UNIVERSITY OF VAASA

SCHOOL OF TECHNOLOGY AND INNOVATIONS

ELECTRICAL ENGINEERING

Konsta Mäenpänen

TESTING COMMUNICATION RELIABILITY WITH FAULT INJECTION

Implementation using Robot Framework and SoC-FPGA

Master's thesis for the degree of Master of Science in Technology submitted for inspection, Vaasa, 4 November 2019.

Supervisor	Timo Vekara
Instructor	Jukka Harjula
Evaluator	Jarmo Alander

PREFACE

I would like to thank Timo Vekara and Jarmo Alander for the guidance both in the thesis work and during my studies. Furthermore, this thesis was made for Danfoss Drives, so I to thank the company for giving me the opportunity by offering me the topic for the thesis. At Danfoss Drives, I would like to give special thanks to my instructor Jukka Harjula as well as Petri Ylirinne and Panu Alho, who have instructed and supported me completing this master's thesis and the preceding bachelor's thesis.

TABLE OF CONTENTS

SYMBOLS AND ABBREVIATIONS

AC	alternating current
ACK	acknowledgement
AMBA	advanced microcontroller bus architecture
API	application programmer interface
ARM	advanced RISC machines
ARQ	automatic repeat request
ASIC	application-specific integrated circuit
AXI	advanced extensible interface
BCC	block check character
CAN	controller area network
CLB	configurable logic block
CPU	central processing unit
CRC	cyclic redundancy check
DC	direct current
FEC	forward error correction
HARQ	hybrid automatic repeat request
HDL	hardware description language
HTML	hypertext markup language
IEEE	Institute of Electrical and Electronics Engineers
IGBT	insulated-gate bipolar transistor
I/O	input/output
IP	internet protocol, intellectual property
IP/UDP	user datagram protocol over internet protocol
IPv4	internet protocol, version 4
JTAG	joint test action group

LCD	liquid-crystal display
LRC	longitudinal redundancy check
LSB	least significant bit or byte
MSB	most significant bit or byte
NAK	negative acknowledgement
NRZ, NRZ-I	no-return to zero, no-return to zero inverted
OSI	open systems interconnection
PC	personal computer
PCB	protocol control block
PL	programmable logic
PS	programmable system
PWM	pulse-width modulation
RF	robot framework
RZ	return to zero
SDK	software development kit
SoC	system on a chip
TCP	transmission control protocol
UDP	user datagram protocol
URL	uniform resource locator
USB	universal serial bus
VRC	vertical redundancy check
XML	extensible markup language

VAASAN YLIOPISTO
Tekniikan ja innovaatiojohtamisen yksikkö
Tekijä: Konsta Mäenpänen
Diplomityön nimi: Kommunikaation luotettavuuden testaus vianinjektoinilla – Robot Framework- ja SoC-FPGA-toteutus
Valvoja: Timo Vekara
Ohjaaja: Jukka Harjula
Tarkastaja: Jarmo Alander
Tutkinto: Diplomi-insinööri
Oppiaine: Sähkötekniikka
Opintojen aloitusvuosi: 2013
Diplomityön valmistumisvuosi: 2019 **Sivumäärä:** 99

TIIVISTELMÄ

Taajuusmuuttajia käytetään teollisuudessa laajasti, sillä merkittävän osan teollisuuden sähkönkulutuksesta muodostavat oikosulkumoottorit, joita ajetaan taajuusmuuttajien avulla. Taajuusmuuttajiin on mahdollista kytkeä optiokortteja, jotka lisäävät taajuusmuuttajaan valvonta-, ohjaus- ym. toiminnallisuuksia. Nämä kortit kommunikoivat sarjaliikenneväylän kautta taajuusmuuttajan pääyksikön kanssa.

Sarjaliikennelinkissä, kuten taajuusmuuttajan väylällä, voi syntyä virheitä, jotka häiritsevät tietoliikennettä. Sen takia sarjaliikenneprotokolliin on luotu virheentunnistus- ja -korjausmekanismeja, joilla pyritään varmistamaan virheetön tiedon kuljettaminen. Luotettavuutta testaamaan voidaan väylälle generoida virheitä siihen tarkoitetulla laitteella.

Tässä diplomityössä luotiin taajuusmuuttajia valmistavan yrityksen, Danfoss Drivesin (aik. Vacon), pyynnöstä häiriögeneraattorijärjestelmä. Järjestelmä koostuu SoC-FPGA-piirillä luodusta virheitä syöttävästä laitteesta, PC-työkalulle luodusta testirajapinnasta sekä Ethernet-kommunikaatiosta niiden välillä. Laite kytketään väylään, ja testirajapinta tekee testaajalle mahdolliseksi luoda mukautettavia testejä ja ajaa testejä käyttäen Robot Framework -testiympäristöä.

Diplomityössä tutkittiin ensin sarjakommunikointiväylien yleisimpiä virheentunnistus- ja korjauskeinoja sekä SoC-FPGA-piirien sekä työssä käytetyn Robot Frameworkin ominaisuuksia. Järjestelmä suunniteltiin ylhäältä-alas-periaatteella ensin tunnistamalla kolmen edellä mainitun komponentin päärakenne päätyen lopulta yksittäisten ohjelmafunktioiden logiikan suunnitteluun. Tämän jälkeen laite ja testirajapinta toteutettiin C- ja Python-ohjelmointikielillä käyttäen suunnitellun kaltaista kommunikaatiota näiden kahden komponentin välillä.

Lopulta järjestelmä testattiin kaikki komponentit yhteen kytkettynä. Varsinainen injektorilogiikka, joka luo virheitä väylään, ei ollut työn loppuun mennessä vielä toimittavan tahon puolelta valmis, joten järjestelmää ei voitu testata todellisessa ympäristössä. Työssä luodut osuudet voidaan kuitenkin myöhemmin kytkeä kokonaiseen järjestelmään.

Työn tärkeimpänä johtopäätöksenä on, että tavoitteiden mukainen järjestelmä saatiin luotua ja testattua toimivaksi mahdollisin osin. Jatkokehityskohteeksi jäi mm. kokonaisen järjestelmän luonti ja testaus oikeaan kommunikaatioväylään kytkettynä.

AVAINSANAT: sarjaliikenne, virheentunnistus ja -korjaus, virheinjektointi, häiriöiden generointi

UNIVERSITY OF VAASA
School of Technology and Innovations
Author: Konsta Mäenpänen
Topic of the Thesis: Testing communication reliability with fault injection – Implementation using Robot Framework and SoC-FPGA
Supervisor: Timo Vekara
Instructor: Jukka Harjula
Evaluator: Jarmo Alander
Degree: Master of Science in Technology
Major of Subject: Electrical Engineering
Year of Entering the University: 2013
Year of Completing the Thesis: 2019 **Pages:** 99

ABSTRACT

Frequency converters are widely used in industry because a notable part of the industrial electricity consumption is by electrical induction motors driven by frequency converters. It is possible to connect option boards into a frequency converter to add monitoring and control features. These option boards communicate with the main control unit of the frequency converter over a serial communication link.

In a serial communication link, e.g. in a frequency converter, it can occur faults that interfere with the transfer. Hence, error handling mechanisms are used to secure transmission of the data without errors. A fault injector device, which generates errors into the data travelling in the link, can be used to test the communication reliability.

In this master's thesis, an error generator system was created for a company, Danfoss Drives (previously Vacon), manufacturing frequency converters. The system consists of a fault injector device created with a SoC-FPGA, a testing interface for a PC tool, and an Ethernet-based communication between these two. The device is connected to a serial communication link, and the testing interface makes it easy for a tester to create and run modifiable fault injection tests using a Robot Framework test environment.

At the beginning of the thesis, the most common error detection and correction mechanisms in serial communication and properties of SoC-FPGAs, and Robot Framework were studied. Following this, the system was designed with top-down approach, first identifying the main structure of the components, and finally ending up in designing the logic of individual functions. After this, the device and the testing interface were implemented in C and Python using the designed Ethernet communication between them.

After the implementation, the system was tested with all the components combined. The actual fault injection logic was not ready by the end of the thesis, so the tests were not run in a real environment. However, the work is done so that the implemented parts can be later used in a complete system.

The most important conclusion is that the system was created and tested to meet the requirements with applicable parts. Further development includes creating a complete system and testing it with a real communication link.

KEYWORDS: serial communication, error detection and correction, fault injection, error generation

1 INTRODUCTION

1.1 Frequency converters

This master's thesis is done for Danfoss Drives which is a manufacturer of frequency converters. Frequency converters have an important role in industry, because a significant share, two thirds of the electrical energy in industry is consumed by electrical motors in industrial applications (Motiva 2006: 14). These motors are mostly induction motors driven by *alternating current* (AC) from the power grid with a frequency converter. The operation speed and torque of an induction motor depend on the voltage and the frequency of the input AC current. Thus, to control the motor, a frequency converter is used to convert the fixed-voltage and frequency AC current from the power grid for the motor energy-efficiently.

1.1.1 Working principle of a frequency converter

A frequency converter is used to control the torque and the speed of an electric motor by adjusting the motor input frequency and voltage. It draws alternating current from the power grid and converts it into alternating current that has a desired voltage and frequency. Typically, the generated sinusoidal AC current is fed into an electric motor. This alternating current generates rotating magnetic fields which eventually rotate the rotor of the electric motor. (Krishnan 2001: 313).

The power conversion part of a frequency converter consist of a rectifier, an inverter and a *direct current* (DC) link circuit between them. This is depicted in Figure 1. The electric power fed to the variable-frequency drive is normally alternating current with three phases (L1, L2 and L3 in Fig. 1). This alternating current is converted into a DC voltage by using a *rectifier* which can consist of a diode bridge or thyristors. This DC link is an intermediate circuit between the AC input and the AC output and it holds a DC voltage. Usually, the DC link circuit contains capacitors that reduce ripple in the DC voltage and thus try to keep the level of the DC link voltage constant. (Krishnan 2001: 314–315).

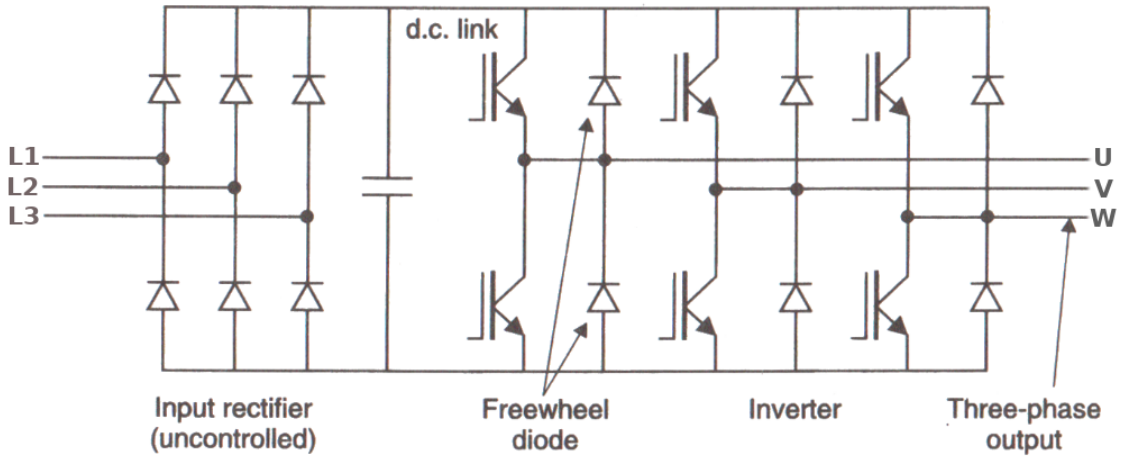

Figure 1. The main parts of the power conversion part of a variable-frequency drive. (Hindmarsh & Renfrew 1996: 259).

The DC voltage provided by the DC link is then converted by an inverter into alternating current. The inverting is done by the inverter (on the right in Figure 1) that chops the DC link voltage with semiconductor switches, usually IGBTs (*insulated-gate bipolar transistor*). These semiconductor switches connect the output of a phase to either the negative or to the positive bus of the DC link (Fisher 1991: 409). The *pulse-width modulation* (PWM) of the variable-frequency drive controls the length of the pulses so that sinusoidal alternating currents are formed at the three-phase output (U, V and W in Fig. 1), which are needed to operate a rotating electric motor (IEEE 1997: 210–216).

1.1.2 Control circuit and option boards of a frequency converter

In addition to the previously described power conversion part, depicted in Fig. 1, the variable-frequency drive also has a control unit (control circuit) that drives the logic of the device. It controls the inverting circuit by switching on and off the semiconductor switches to produce the desired waveform for the output. The control unit is also responsible of receiving inputs such as adjustments in speed or torque or to stop or start the motor, and to react to them accordingly. The main parts of a control unit are shown in Fig. 2. (Danfoss 2000: 52).

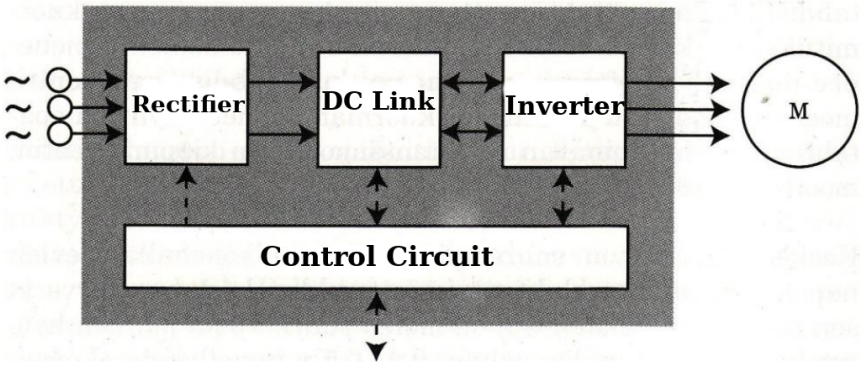

Figure 2. Main parts of a variable-frequency drive, including the power conversion part (rectifier, DC link and inverter) and the control circuit. (Danfoss 2000: 52).

The control unit communicates also with the user interface logic. The user of the drive can perform the operations by a physical operator panel on the front side of the variable-frequency drive, shown in Figure 3, or by a personal computer (PC).

Figure 3. The operator panel of a Vacon NXP frequency converter with an LCD (*liquid crystal display*) and a keypad.

In addition to this, the control unit handles other digital and analogue inputs or outputs which may be connected to an *option board* that performs a special task. These extensions can be inserted into the existing slots inside the variable-frequency drive to add functionality without replacing the variable frequency drive or its control unit. The option boards can be designed to handle some special operation and communicate with the drive itself. (Danfoss 2016: 4–5).

Option boards can be connected into a variable-frequency drive to easily add functionality. Each card has a dedicated functionality, and they communicate with the control unit of the drive through a serial communication link. This communication between the option boards and the control unit must be robust to ensure that the functionality works as intended, which is critical especially in safety-critical applications. Thus, serial communication links need to make sure that instructions are passed between the components without errors.

The robustness of the communication is a vital part of the functionality of a variable-frequency drive. Therefore, verifying how the error handling in the communication is working, cannot be overlooked. To improve testing process effectiveness and the coverage of the tests, proper development procedures such as test-driven development and different testing frameworks can be utilised.

1.2 Objective

The objective of this thesis is to create a system generating errors, an *error generator system*, depicted in Fig. 4. Its task is to inject faults into a serial communication link to test the reliability of the communication. The created error generator system is used in the test automation system of Danfoss. The system consists of the components that are depicted in Figure 4:

1. *A fault injector*, that is an SoC-FPGA (*system-on-a-chip field-programmable gate array*). It generates and injects errors into a communication link connected to it.
2. *Test library*, which provides an interface for a tester to create and run tests with a personal computer (PC) by using the fault injector. It consists of fault injection functions (“*keywords*”) that can be called, and test templates. The test library communicates with the fault injector over Ethernet.
3. A high-speed communication link and the nodes communicating over the link. The fault injector is connected to the link, and the tests that are generated and run with the testing library should generate errors in the data in the link.

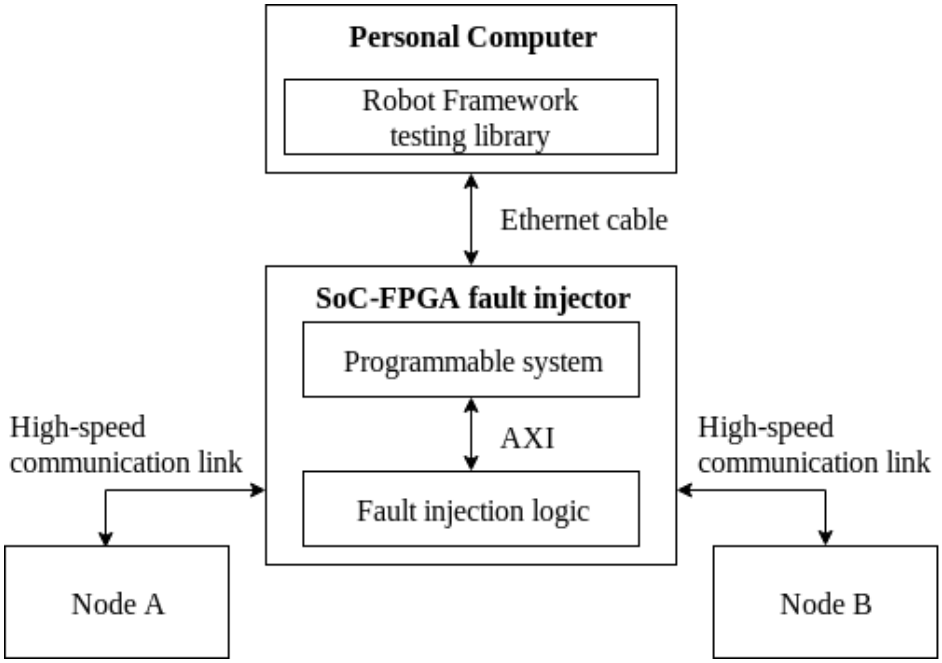

Figure 4. The initial schematic of the error generator system that is designed and implemented in this thesis.

The objective of this thesis is divided into three parts:

1. Designing and implementing the *Robot Framework* test library (Robot Framework is discussed in detail in Subchapter 2.5).
2. Creating the design for the programmable system (the processor) of the SoC-FPGA fault injector. Firstly, it communicates over Ethernet with the test library, as well as with an AXI (*Advanced eXtensible Interface*) protocol with a *fault injection logic block*.
3. Designing the communication between the test library and the SoC-FPGA fault injector over Ethernet.

The fault injection logic block is an intellectual property (IP) FPGA block, that is implemented on the same SoC-FPGA as the fault injector. It does the actual error generating and injection on the communication link, that is connected to the SoC-FPGA. However, the fault injection logic block is designed by Danfoss, and thus it is supplied later for the thesis and designing it is not included in the objective. Also, the high-speed communication link, the nodes on the link and the data transferred in the link are not included in the objective of this thesis. The communication link and the fault injection logic block are shortly discussed more in Subchapters 3.2. and 3.3.

1.3 Structure

The structure of this thesis consists roughly of three larger parts: the theory, the design and implementation and the testing part. In the theory part, essential theory is studied to provide the possibility to continue with the implementation and especially the testing part where the results are reflected with the presented theory. The implementation part describes the creation process of the system, and finally, testing part lists and analyses the test results and other observations made while demonstrating the device.

A significant part of Chapter 2 concentrates on research of the serial communication protocols. They are studied focusing on error detection and error handling techniques. Since the underlying communication bus protocol is implemented by the company, its implementation details are not shared. However, the error handling mechanisms of two other protocols are shortly described to give reference. SoC-FPGAs and the Ethernet communication possibilities are also studied with some extent to give base and helpful reference for the design and implementation phase. Finally, the test framework, *Robot Framework*, is introduced, focusing on what can be achieved with it.

The design and implementation phase starts with the more detailed description of the structure of the error generator system. All the parts of the error generator system that are created (the SoC-FPGA fault injector, the Robot Framework test library and the communication between them) are described, including how the system is integrated as a part of Danfoss test automation system. After this, both the hardware part on the SoC-FPGA and the testing library with Robot Framework are designed. Finally, the design is implemented and the implementation details are given.

Chapter 6 includes the planning of tests, the description of the test environment as well as the test results. The test results are analysed and summarised at the end of the chapter by using reflection from the theory and the objective parts of the thesis. Finally, the previous parts are wrapped up with conclusions and a summary.

2 FAULT INJECTION

In this chapter, required theory is studied to give reference and help for the work in the design and implementation phase as well as the testing for the error generator system. The study tries to flow logically from the communication link, where the errors are generated and injected, to the software where the tests are eventually run, opening all the relative theory needed in between.

First, we start by introducing the serial communication basics and their error detection techniques to understand what is happening inside the error generator system. Then some research is done on fault injection in general as well as faults in digital communication to grasp consequences of faults and what benefits can be achieved by using fault injection in testing and increasing robustness of a system against those faults.

After this research, which is carried out in rather a lecture-like form, the remaining study focuses on the practical part of this thesis. Study is done on how the hardware of the SoC-FPGA fault injector works and how it communicates with other components in the error generator system. Ultimately, introduction of a test automation framework "Robot Framework" which is used in the thesis for the test library, is given.

2.1 Serial communication

Serial communication in general consists of sending data in serial rather than in parallel. In other words, in a serial communication, a sender sends, and a receiver receives one bit after another instead of multiple bits at the same time (Axelson 2007: 11). To understand the effects of fault injection into a serial communication link, the basics of serial communication are studied.

In this subchapter, the focus is on the physical layer and the data link layer, the two bottom-most layers, of the OSI (open systems interconnection) model. OSI model is a conceptual model defined by an ISO (International Organization for Standardization) standard. According to the model, communication is built on layers, each of which provides

defines functionality for the layer above, by encapsulating the functionality from the layer below (ISO 2002).

2.1.1 Serial versus parallel communication

Serial communication is used very widely in different applications, mainly due to its more efficient usage of the data transfer medium than parallel communication methods. When a serial communication is used, in the most optimal situation only two wires (one for the data and one for ground) can be used to transfer the data when an electrical transfer medium is used. There might also be a clock signal if the communication is synchronous. A serial communication without a clock signal is called asynchronous communication. (Dell 2015: 90).

In a parallel communication, more wires are needed to send the data. For example, sending an eight-bit byte requires at least eight parallel data wires and a ground wire. In general, parallel communications are easy to implement and in addition, because more data can be sent over a parallel communication during the same time, they are usually faster than serial communications. This is also a drawback, because it increases the manufacturing costs of the cables and the connectors and using more signal paths in an integrated circuit reduces the available space in the circuit. Furthermore, due to the large number of conductors, there is more interference and crosstalk in parallel communication than in a serial link. Also, because of this, the transfer distance is limited which is why serial communications are mostly preferred and used especially when data is sent over longer distances, e.g. from one circuit to another using a cable. (Murthy 2009: 119–120).

2.1.2 Synchronous and asynchronous serial communication

In a serial communication – or in digital communication in general – it is vital that the receiver and the transmitter of the data are synchronised, meaning that the received data from the transmitter is sampled at the correct interval at the receiver. If the parties were not synchronised, there would not be a way to determine, where the data would start and end, resulting into corrupted data and inability for the parties to communicate. These synchronisation errors in critical applications, such as in a variable-frequency motor

drive, could lead to fatal consequences. Thus, here, a comparison of synchronisation methods is done to grasp how the synchronisation can be carried out and what are their effects on both the communication robustness and transfer speed (Dell 2015: 90).

As shown in Figure 5, the synchronisation can be carried out in a protocol by a separate clock signal or by adding information in the data to provide the receiver the possibility for synchronisation.

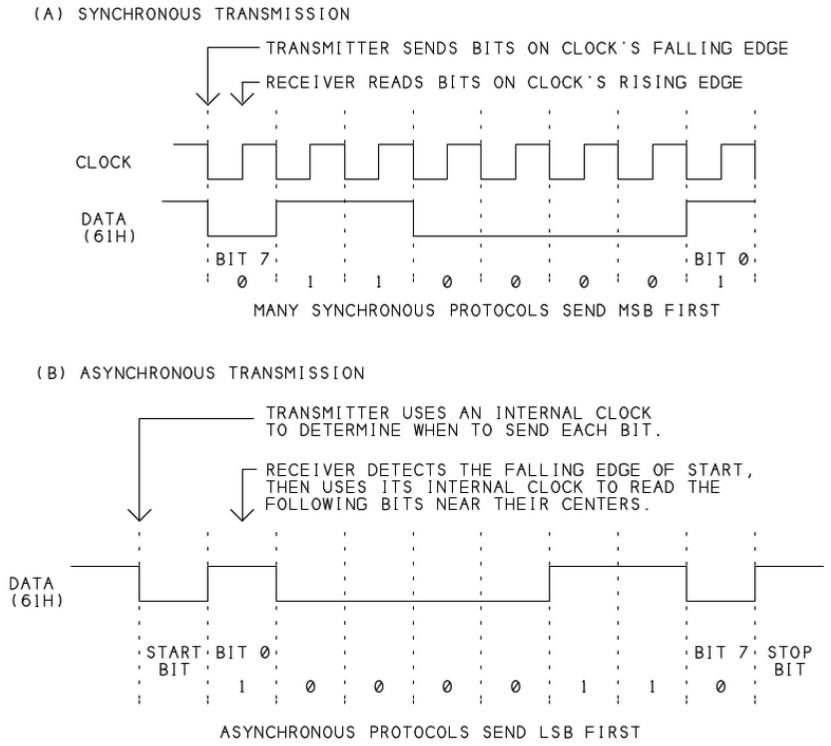

Figure 5. An illustration by Axelson (2007: 13) on how synchronous serial communication requires a separate line for the clock, whereas an asynchronous transmission needs synchronisation bits (here a start bit and a stop bit) to synchronise the data.

In a **synchronous** transfer, a clock signal is transferred alongside with the sent data. This pulsating clock signal states the beginning of each data bit which synchronises the communication. With this setup, high speed rates can be achieved. The drawback is that transferring the clock signal requires an additional wire and the protocol must define who the master is, i.e. which party of the communication is responsible of generating the clock signal. (Murthy 2009: 122).

In **asynchronous** communication, however, no clock signal is being transferred. The transmitter and the receiver must agree beforehand on the speed of the communication, e.g. by a definition in the standard or by configuring the parties. Despite this, because of inaccuracy, the clocks of the receiver and the transmitter may and will probably not be exactly the same. Therefore, by sending synchronisation bits at the beginning of each data packet (a transfer unit of data consisting of bits), the receiver can both acknowledge when the actual data starts and synchronise the clock if any skew has occurred in relation to that of the transmitter. This way with asynchronous communication, one wire for the clock can be saved, as well as the configuration for the clock master. On the other hand, extra data must be sent in form of synchronisation bits which increases the overhead. Furthermore, another drawback is the clock skew which limits the maximum speed of the communication. (Axelson 2007: 11–12).

2.2 Error control in digital communication

In this thesis, an error generator system is created to test the tolerance of the communication link against faults. With a proper error detection and handling in this serial communication protocol, number of errors can be decreased.

As stated, the system that is created, tries to simulate real-life faults and errors by intentionally generating and injecting them into the communication link to test robustness. These real-life faults and errors in digital communication can be caused by multiple factors “in the field”. As stated in Subchapter 2.1.2, inaccurate or flawed clocks can perturb the communication if synchronisation is not implemented properly. In addition, outside factors such as interference and disturbance in the actual transfer medium are simple reasons for an error. Furthermore, the limited capacity of a communication link can cause errors if congestion in the communication occurs. The causes of faults are discussed in more detail and listed in Subchapter 2.2.1.

In general, error control against these faults in any protocol consists roughly of two parts: managing how the receiving end should detect an error in the message (error detection) and the operations to ensure that the correct message can be delivered to the recipient by the protocol should there be an error (error handling). To comprehend the

fault management in the communication link, different error detection and correction mechanisms are studied and compared in Subchapters 2.2.2 and 2.2.3 to provide base for reflection in the eventual testing phase.

2.2.1 Types and causes of errors in digital communication

As stated earlier, the error generator system simulates real-life faults on a serial link. These errors can occur in several ways. Firstly, through the electromagnetic interference (EMI), disturbing currents can be induced in the electric transmission link, modifying the digital signal travelling in it. The electric transfer medium can either absorb electromagnetic interference from other electric devices that are sources of electromagnetic fields, or it can be interfered with by another nearby communication wire, also known as *crosstalk* (Malarić 2010: 69–75). The interference can come from a natural source, too. A lightning strike or electrostatic discharges are typical examples of this (Malarić 2010: 59).

Secondly, all electrical signals contain some amount of noise. It is a random disturbance in a signal, and it is caused by e.g. the thermal motion of the electrons (thermal noise) as well as the randomness in their movement (shot noise). Usually, the noise level is low but if the relative amplitude of the noise in a signal rises too high compared with the actual data signal, the information in the signal cannot be distinguished from the noise. (Malarić 2010: 58).

Furthermore, as already stated in Subchapter 2.1.2, the lack of synchronisation can cause errors, too. If the communication protocol does not define enough means of resynchronisation of the transmitter and the receiver by for instance a clock signal or a self-clocking line coding, clock skew can occur, meaning that the receiver samples the signal at an erroneous point, which can possibly lead to a faulty bit value read by the receiver. Synchronisation errors can occur even if there were no interference or disturbance with the travelling in the communication link.

Errors in the digital communication are realised as flipped bits. This means that the transmitter intended to send a bit but because of an error in the signal, the receiver reads

the value of the bit flipped, so that the value of the bit is of opposite polarity. Single and multiple bit errors are shown in Fig. 6.

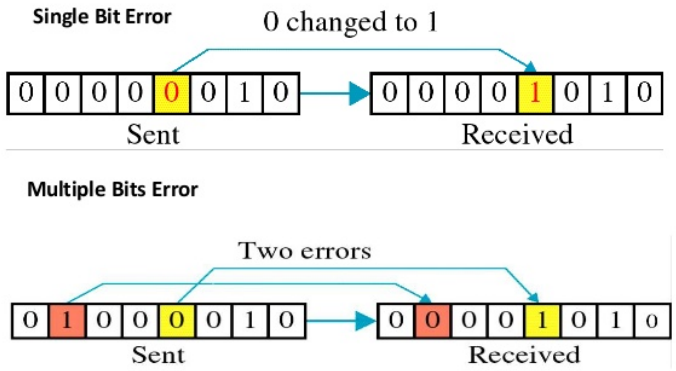

Figure 6. Single bit errors and multiple bit errors.

If only one bit in the data packet changes, a single bit error occurs. However, in the serial communication protocols, the duration to transfer a single bit is very small. Therefore, it is very unlikely that only a single bit is flipped because of an error, so single bit errors are relatively rare in serial communication.

Thus, it is more likely that multiple bits are flipped because of a fault. This is because the erroneous state in the communication link usually lasts for a relatively long time compared to the transfer speed. This is called a *burst error*. Multiple bits – which do not necessarily have to be consecutive – are flipped, causing incorrect data. If the erroneous state lasts very long, the whole packet can be corrupted by flipped bits, causing that the other end will never receive the packet, resulting in a lost packet.

Furthermore, if the communication protocol uses automatic repeat request, ARQ (discussed later in Subchapter 2.2.3), the receiver must send information to the transmitter that it had received the data. If an error occurs while sending this *acknowledgement*, the transmitter would try to resend the same data. In this case, a duplicate packet error takes place, because the given packet has already been sent.

Data packets can be lost because of an error or they can intentionally be dropped, leading into *dropped packets*. This can happen if there is too much traffic on the communic-

ation link. If too many packets would be sent into the communication link, the transmitter has to drop some of the packets to avoid traffic congestion in the communication link.

2.2.2 Detecting errors in transmitted data

Detecting errors in communication protocols is based on sending data with inserted additional or redundant information that can be used to validate data integrity at the receiving end. In a very simple example, the protocol can define a following error correction mechanism: If the sender sends four bits, e.g. 0110, it should add those four bits repeated as a data validation field, thus sending 8 bits (0110 0110). So, if the protocol is followed, when the receiver receives anything different than a packet that consists of two exactly same four-bit bytes, it will detect an error and the communication will begin a procedure to handle the error. The principle is simple. However, because the given example would be very slow and ineffective, much more accurate and effective algorithms are used in practice to calculate the additional error detection field, some the most common of which are presented next.

Parity is a simple way to check that the data is transmitted immutably. Parity check is used to validate data in a byte level by adding a parity bit at the end of the actual data bits. Thus, if a byte contains an even number of 1's, the parity bit should be zero. Conversely, if the number of 1's in the byte is odd, the parity bit is set to one. This is called *even parity*, since an even number of 1's produces a 0 parity bit. Hence, *odd parity* would mean that the value of the parity bit is inverted. By checking if the number of 1's in the data byte is the same as stated by the parity bit, an error can be detected. However, if an error caused an even number of changes in the data bits, the error would go undetected. For example, noise bursts usually cause disintegration to more than only one bit, which is why parity is not very reliable form of detecting errors. (Hioki 2001: 519–522).

Previously mentioned way of calculating parity is usually referred as **vertical redundancy check** (VRC) because the parity bit is calculated for each byte. Parity bits can be computed also at the end of the block of a multiple-byte-message, which is called **longitudinal redundancy check** (LRC). The parity is computed for each bit position from

LSB (least significant bit) to MSB (*most significant bit*) and the possible VRC bit. With the combination of VRC and LRC, the position of the erroneous bit can be found and corrected unlike if only vertical parity was applied. However, like in vertical redundancy check, an even number of errors in the bits cannot be found. (Hioki 2001: 522). Table 1 gives an example how the LRC and VRC are calculated for a multiple word message.

Table 1. A message consisting of four 4-bit data words. Each data word has in addition a parity bit calculated with vertical redundancy check. A parity calculation is then applied to each bit position (from bit 0 to VRC) into the LRC column. Here, even parity is applied.

	Word 1	**Word 2**	**Word 3**	**Word 4**	**LRC**
bit 0	0	1	1	0	0
bit 1	1	1	0	1	1
bit 2	0	1	0	1	0
bit 3	1	0	0	1	0
VRC	0	1	1	1	1

Cyclic redundancy check (CRC) is a more effective way to detect errors. In the CRC, the message block is divided by a *generator polynomial* and the remainder of the division, a *block check character* (BCC), is appended at the end of the message block, which is depicted in Fig. 7. The receiver then checks the integrity of the data by dividing the message block containing the block check character by the same generator polynomial: expected remainder of the division should be 0 if the data was transferred without any errors detected. (Peterson & Brown 1961: 228–235).

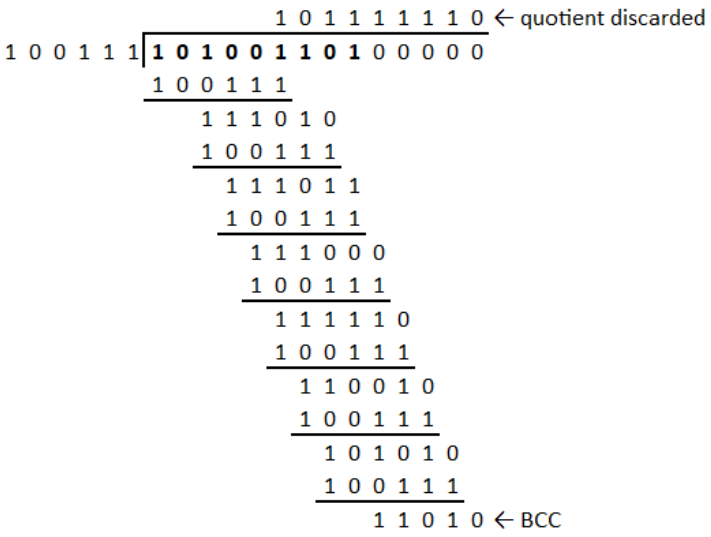

Figure 7. An example of creating the BCC using CRC according to Hioki (2001: 526). Here, the divisor is the generator polynomial $X^5 + X^2 + X + 1$ = 100111 and the dividend is the actual data 101001101 with 5 trailing zeroes. The number of zeros added defines the length of the BCC.

Because CRC is more efficient than the vertical and longitudinal redundancy checks and still does not increase the implementation costs significantly, it is used widely as the main error detection mechanism in many serial communication protocols. (Hioki 2001: 524–525).

Checksums are another popular way to detect errors. As the name implies, the bytes in the data are added up and the sum is appended at the end of the actual message as the block check character (BCC). Depending on the length of the BCC, the checksum can be a single-precision checksum or a double-precision checksum. Here, any overflow or carry is ignored; the accuracy of the error detection can be increased by using either the residue checksum where the overflown value is added back to the checksum or the Honeywell checksum where the checksum is calculated by summing up the bits of two consecutive data words instead of one. (Hioki 2001: 529–531).

Cyclic redundancy check is a very common fault detection mechanism, thanks to its high error detection rate. For example, universal serial bus (USB) which is a popular serial communication protocol commonly used to connect peripherals and other devices with a personal computer, uses CRC to detect corrupt packages. Another popular serial communication protocol, controller area network (CAN), which is used broadly in auto-

motive industry, relies its error detection on CRC by, similarly to USB, appending a CRC field at the end of each data packet sent as shown in Fig. 8. (Compaq & al. 2000: 1999 , Bosch 1991: 59).

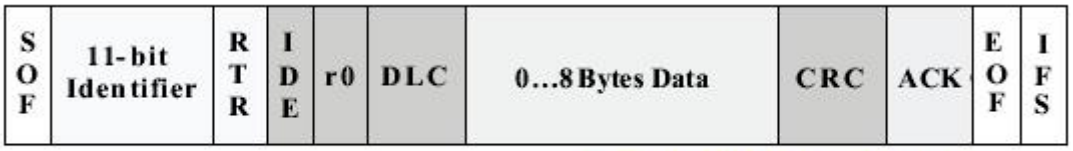

Figure 8. CAN frame format showing the CRC and ACK fields that are vital for error management (Davis & al. 2007: 245).

2.2.3 Error correction in a communication link

After an error is detected, the protocol should handle how to manage it. There are two basic types of correcting the errors in the communication; *automatic repeat request* (ARQ) and *forward error correction* (FEC). In ARQ, the receiver must automatically send a response to the sender acknowledging that the received data is transmitted correctly. If the receiver detects an error by using an error detection mechanism, e.g. a CRC, it should respond the transmitter negatively or not at all so that the transmitter may try to send the data again (Hioki 2001: 532). The example of a sequence of frames sent with an ARQ is shown in Fig. 9.

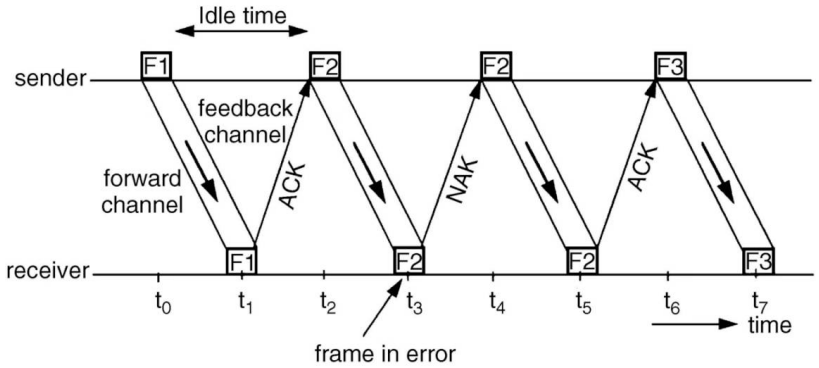

Figure 9. The sender sends three data frames (F1, F2, F3) with an ARQ protocol. The receiver detects an error in frame F2 and sends a *negative acknowledgement* (NAK) to force the sender to resend the frame.

In FEC, however, the sent data includes a correction code which injects redundant bits in the data. The receiver can decode the correction code then to both detect the error and to correct the error (Hioki 2001: 532). This decreases the need to retransmit the data if an error occurs, which is why FEC is preferred in communications where resending the data would be too expensive or not possible. An example of a FEC mechanism is a very widely used *hamming code* (Hamming 1950) but the implementation details will not be presented here.

One of the previous two can be applied for a protocol. In addition, a combination of these two can be used to both request a retransmission and correcting errors by a code. This is called *hybrid automatic repeat request* (HARQ). In HARQ, in a simplified manner, the receiver tries to correct errors with the error correction code. If there are too many errors, an automatic repeat request is sent. (Wicker 1995: 409).

Another possibility is *blind transmission*, or *unacknowledged connectionless transfer*, where neither positive or negative acknowledgement is returned to the transmitter. Hence, the transmitter cannot be sure whether the transmission was error-free or whether the data ever received its destination. Even though data transmission cannot be re-requested, data with faults are ignored (“dropped”) by the receiver e.g. with the CRC, so that only correct data will be handled. (Shinde 2000: 171).

The obvious drawback of blind transmission is the uncertainty in the transmission, but not acknowledging the data reduces the bandwidth usage of the communication link and increases speed, when there is no acknowledgement and retransmission. The method is useful, when the bandwidth is limited, acknowledgement is not necessary or when there is no possibility to acknowledge (lack of return channel). To reduce the number of lost packets, the transmitter can send a packet multiple times and “hope” that at least one of them will receive the destination.

Table 2 shows the comparison of the error correction mechanisms studied here. As can be seen, all the correction mechanisms have some advantages but also drawbacks, meaning that the mechanism should be carefully selected to fit the application.

Table 2. Error correction mechanisms in a nutshell.

Error correction mechanism	Basis of correction	Advantages	Drawbacks
ARQ	Positive or negative response returned depending on the result of error detection	Data is sent reliably	ACKs use bandwidth and retransmissions cause delay
FEC	Correction code included in data; receiver detects and corrects the error	No need to return acknowledgement	Additional correction code must be inserted into the message
HARQ	Combination of ARQ and FEC	Best of both ARQ and FEC	Those of ARQ & FEC
Blind transmission	Data with errors is dropped but no repeat request	Speed and low use of communication link. No additional codes.	Lost data cannot be recovered

The previously presented serial communication protocol examples, USB and CAN, both rely on ARQ as their error correction mechanism. In USB, the client should send an acknowledgement packet (ACK) back to the host after each data packet that is sent, if no errors were detected. If the host does not receive an ACK packet within a certain time limit, a frame, it will assume that the transmission failed and will resend the same data packet. In CAN, however, because of its distributed nature, if none of the receiving devices detects that the transmission was error-free, an acknowledgement error occurs, and the sender will try to retransmit the packet. Furthermore, if a device detects an error, it will send an error frame, which corrupts the packet that is being transmitted and forces the sender to retransmit.

With the acknowledgement mechanism in the ARQ based error correction, the corrupted packets in USB (or frames in CAN) that are detected by the CRC can be resent properly to the receiver. Furthermore, acknowledging each packet ensures protection against lost packets; if the packet is lost, the transmitter will not get an ACK response and is forced to resend the packet.

The ARQ must be extended to detect duplicate packets. In USB, for example, detecting a duplicate packet is implemented by data toggling, i.e. adding a packet identifier, that is

toggled between *DATA0* and *DATA1* for each sent data packet (Axelson 2015: 51–53). A duplicate packet would be detected by the USB device if the packet identifier did not change between consecutive received data packets. On the other hand, because the transmitter in a CAN bus doesn't communicate directly with the receiver but – unlike in USB – it broadcasts the frame to all the receivers, managing duplicate packets is not done by the protocol itself but should be implemented by the receiving nodes.

2.3 Fault injection testing

In this thesis, an error generator system is created to test communication reliability with fault injection. In fault injection in general, a set of selected faults is injected into a system. The system is then monitored to validate that the response of the system matches the defined specifications during faulty conditions. In this way, test coverage can be improved when also more seldom error conditions can be tested, too.

According to Benso & Brinetto (2003: 28–30), the faults in a hardware system can be divided into three main categories:

- hardware faults caused by wearing out or breaking of a hardware component,
- intermittent (or periodic) faults because of the instability of the system, and
- transient faults from e.g. EMI and noise.

So, in our error generating system, it can be validated that the communication link (*the system*) manages the corrupted or lost packets (*the injected faults*) by detecting the errors and starting the error management procedure defined in the protocol (*responding by the definition*). (Koren & Krishna 2007: 355, Benso & Brinetto 2003: 35).

The permanent faults are relatively infrequent and easy to detect, and the hardware system can be recovered from them by repairing or replacing the component that causes the fault. Also, the system can be recovered from intermittent faults by either component replacement or redesigning the system if the intermittent fault is caused by a design error (a "bug") (Benso & Brinetto 2003: 28–29).

On the other hand, the transient faults are the most common faults and they are more difficult to detect than permanent or intermittent faults. Usually, the transient faults do not cause significant damage to the hardware system itself but rather leave the system into a faulty state for a short time which is likely to cause unexpected behaviour in the system. Thus, fault injection tests on hardware, e.g. on a communication link, should have their focus on the transient faults. (Benso & Brinetto 2003: 11–12).

It is possible to generate transient errors that resemble the effects of a real-life fault directly in the hardware. However, direct hardware fault injection can damage the system and it is very device-specific and thus not a portable solution (Benso & Brinetto 2003: 30–31). A more popular way is to connect a software-based fault injector device into the hardware system. The software in the fault injector manages changing the bits in the data, producing a fault with same effects as an actual physical fault (Koren & Krishna 2007: 357).

So, the SoC-FPGA fault injector in this thesis can be connected on the communication link between the parties. It manipulates the data sent by the transmitter before its transmitted to the receiver, and so it injects a fault in the communication system that reminds of a real hardware fault, such as excessive electromagnetic interference on the communication link. With the error generator system, the following faults are able to be injected into the communication link with the use of the Robot Framework test library:

1. CRC fault: bits in the message data are flipped so that the CRC calculation is violated. CRC field is not modified.
2. Change target address: The message contains a “target address” field. Its contents are manipulated so that the message is received by an undesired receiver.
3. Increasing/decreasing the length of the message.
4. Duplicate: The message is sent twice with the same contents.
5. Removing the message: The message is lost and thus not received.

The error detection and handling, and the contents of a message in the serial communication link in this thesis is discussed in more detail later, in Subchapter 3.2.

2.4 SoC-FPGAs

As the fault injector device in this thesis, a f*ield-programmable gate-array* (FPGA) is used. It is a circuit that is field-programmable or reconfigurable by the designer. An FPGA can be programmed by designing and specifying the logic with a *hardware description language* (HDL). An FPGA consists of a high number of c*onfigurable logic blocks* (CLBs) which usually implement a simple logical function or memory. When the FPGA is programmed, a configuration tool connects these multiple configurable logic cells of the FPGA together so that a circuit that implements the specified logic is created. Thus, an FPGA can implement a logic of an *application-specific integrated circuit* (ASIC) but has the advantage of being reconfigurable which increases available development time and reduces costs. (Shannon 2012: 127).

Because an FPGA does not contain any logic itself, designing the logic must be done from the beginning. Thanks to the possibility to fully customise the logic, FPGAs are faster in signal processing than a CPU (central processing unit) but because of the lack of overhead such as predefined processor instructions and peripherals like memory units, the development is very time consuming.

So, usually it is more reasonable in an application to use a microcontroller or a processor where some of the more complex calculations are done, and use an FPGA where data processing that requires low latency and speed is needed. To create this kind of a system, a stand-alone CPU and an FPGA alongside with other peripherals can be used. A more economically sound choice is to use a soft core, i.e. to implement the CPU logic on the FPGA itself – since any logic, even a CPU can be programmed on an FPGA. One alternative is to use a SoC FPGA.

A SoC FPGA is a single integrated circuit that has a full processor architecture including a central processing unit, memory and inputs and outputs (I/O) and other peripherals such as networking interfaces on a single chip. The chip is relatively small, so it can be used in applications where circuit size is limited, and low power consumption is needed. This processor is integrated with an FPGA onto a single chip to utilise the best parts of the both. The advantage of this is that, unlike in a system consisting of a separate micro-

processor and an FPGA, the SoC FPGA is on a single silicon, decreasing the distance between the signals that are transmitted between the two, thus decreasing power consumption, response time and manufacturing costs. (Altera 2013: 1–4).

2.4.1 Advanced eXtensible Interface

The Advanced eXtensible Interface (AXI) is an interface protocol defined in ARM (*advanced RISC machines*) *advanced microcontroller bus architecture* (AMBA) specification. It is widely used protocol for developing applications for SoC-FPGAs. The AXI protocol defines the rules how the data is exchanged between intellectual property (IP) cores. (Xilinx 2012: 5).

Thus, the AXI protocol can, for instance, be used to set up the communication between the programmable system (PS), i.e. the microcontroller and the programmable logic, the FPGA. In the use case of the thesis, the programmable system can receive data over network from the PC tool, process it, and write the processed data with AXI protocol to the fault injection logic block, which is an FPGA IP block.

The communication with an AXI protocol starts with a handshake process during which the master of the AXI communication and the slave indicate that they both are ready for the data transmission. After a successful handshake, the data is transferred over the selected channel, depending what kind of action is desired. There are five different channels: write address channel, write data channel, write response channel, read address channel and read data channel. All the channels have their own handshake channels, and the handshake is executed using the handshake channel of the corresponding action. For example, if the AXI master intends to read data from the AXI slave, the address is transferred over the read address channel, and the read data is transferred to the master with the read data channel, as shown in Fig. 10. (Xilinx 2012: 5–6).

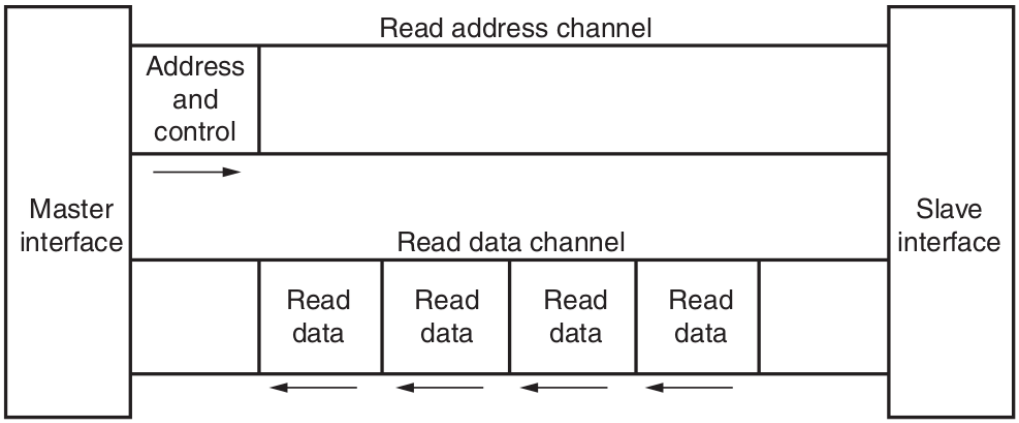

Figure 10. The read transaction uses the read address and the read data channels to read data (Xilinx 2012: 5).

Figure 11 illustrates the channels and the data used during a write transaction over the AXI protocol. For example, when the master requests to write data, a handshake is done on the *write address* channel, after which the data, the address where the data write should occur, is sent by the master. After this, another handshake is executed on the *write data* channel, and the master sends the actual data which should be written to the address. Finally, the slave sends an acknowledgement for successful data write by using the *write response* channel.

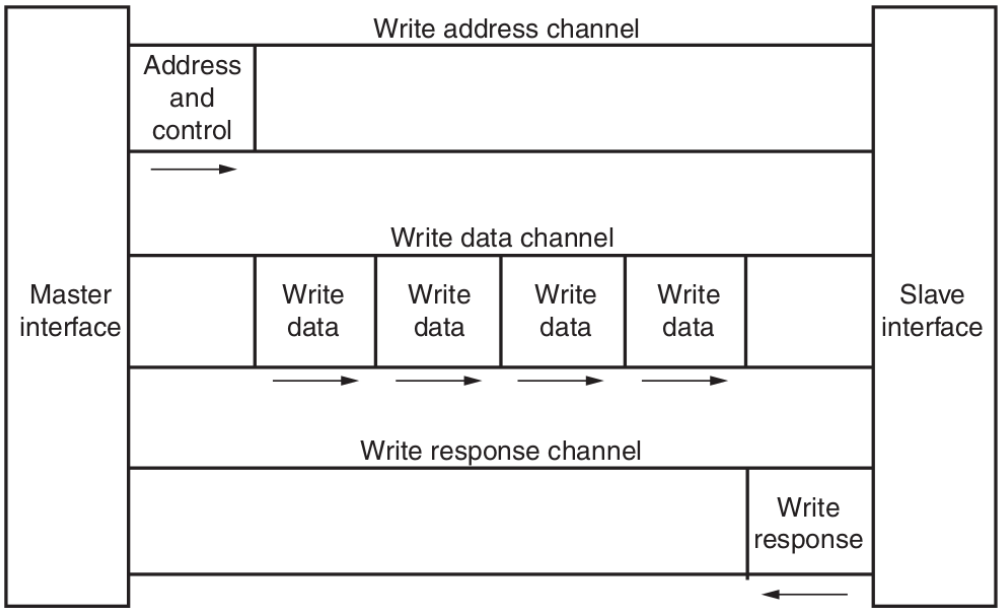

Figure 11. Write channels (write address, write data & write response) in the AXI interface and the flow during the write transaction (Xilinx 2012: 6).

2.4.2 Internet protocol and user datagram protocol

The fault injector device which is implemented in the thesis, will communicate with the PC tool over Ethernet – this should not be confused with the serial communication link where the errors are generated. Thus, implementing the communication requires knowledge also on the protocols that are used to transfer data over Ethernet. On the transport layer, the *user datagram protocol* (UDP) is used together with the *internet protocol* (IP) underneath it on the internet layer and, as stated, Ethernet on the link layer. An alternative for UDP could be transmission control protocol (TCP).

The internet protocol is the protocol that takes responsibility to route the transferred data between the networks. The actual data is in a packet of a higher-level protocol (such as UDP or TCP) and when sending such a packet over the Internet Protocol, it is encapsulated in a IP packet. The IP packet adds additional headers into the data, such as IP address which is used to route the package into the correct network and to the correct recipient. Furthermore, the IP packet is wrapped into a frame in the link layer protocol (e.g. Ethernet) and this data is passed into the transfer medium. An example with the data encapsulation hierarchy (with UDP/IP) can be seen in Fig. 12. (RFC 791: 1–11).

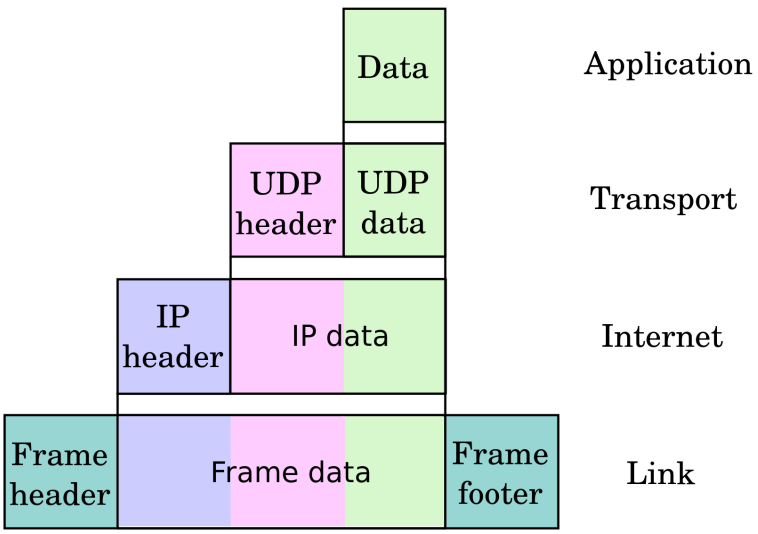

Figure 12. Data encapsulation with UDP and IP.

Also, the Internet Protocol takes responsibility of fragmentation and reassembly of the data. This means that if the link layer below the IP layer has restricted length for the

frame it can transfer, the Internet Protocol will slice the data in smaller fragments to divide it into several frames. Upon the receiving, the receiver will then reassembly the fragmented data. (RFC 791: 7–8).

With the UDP protocol, applications can send *datagrams*, i.e. data packets, over an IP network. As depicted in Fig. 13, a UDP datagram consists of a header field which has the destination address (IP address) for the datagram, the data length and optionally a source port and a checksum to validate the data. (RFC 768: 1).

When the IP protocol is used, also a *pseudo header* is added before the UDP packet's own header and it consists of the source and destination IP addresses as well as the protocol identifier and the packet length. This pseudo header has no functional meaning; the IP addresses here are not used to route the data, but they are only a part of the checksum calculation. (RFC 768: 2).

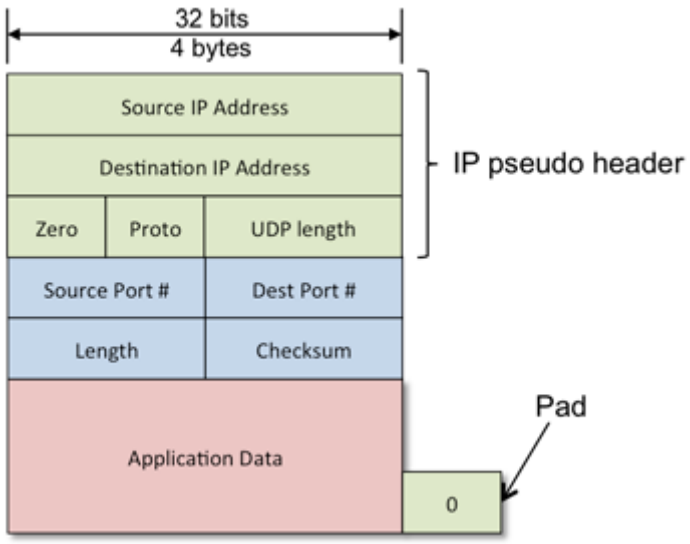

Figure 13. UDP datagram with IP (version 4, IPv4) pseudo header in green, the UDP datagram header in blue and the actual information in red. The pseudo header takes up 3×4 bytes and the UDP header 2×4 bytes so the total size of the header is 20 bytes.

In UDP, no connection between the transmitter and the receiver needs to be established to transfer data. Thus, without a connection, the application cannot make sure if the datagrams have been delivered successfully or that in which order they are received. This may restrict the use of UDP in some applications because lack of error handling, but it

also has advantages: unlike in the TCP, which would be an alternative protocol for communication over internet, there is no handshake process to establish the connection or a disconnecting process between the parties, but only direct delivery of the datagrams which decreases the delay time.

Also, because of the lack of acknowledgement on successfully transmitted datagrams, the data that is not delivered, is just dropped out instead of requesting the retransmission of the data. This is advantageous in real-time applications such as audio or video streaming to decrease the number of interruptions in the data because of error correction and retransmission of the data. If the communication robustness in the application is critical, TCP can be used instead of UDP – or the application that is using UDP can implement own error handling or connection mechanisms in the application protocol over the UDP.

The two transport layer protocols are compared in Table 3. In the comparison, it can be seen that UDP excels over TCP with high speed and low load but is less robust.

Table 3. Comparison of the basic attributes of the UDP and TCP protocols. (Wikibooks 2019).

Feature	UDP	TCP
Description	Simple high speed low functionality wrapper that interface applications to the network layer and does little else.	Full-featured protocol that allows applications to send data reliably without worrying about network layer issues.
Connection setup	Connectionless data is sent without setup.	Connection-oriented; connection must be established prior to transmission.
Data interface to the application	Message base -based; data is sent in discrete packages by the application.	Stream-based; data is sent by the application with no particular structure.
Reliability and acknowledgements	Unreliable best-effort delivery without acknowledgements.	Reliable delivery of message, all data is acknowledged.
Retransmissions	Not performed. Applications must detect lost data and retransmit if needed.	Delivery of all data is managed, and lost data is retransmitted automatically.
Features provided to manage flow of data	None	Flow control using sliding windows, window size adjustment heuristics and congestion avoidance algorithms.
Overhead	Very low	Low, but higher than UDP
Transmission speed	Very high	High, but not as high as UDP
Data quantity suitability	From small to moderate amounts of data.	From small to very large amounts of data.

2.5 Robot Framework test automation framework

Robot Framework is a generic automation framework for testing written in Python programming language. In this thesis, Robot Framework is utilised to generate test functionality and to run tests. With Robot Framework, first of all, the user can use a tabular based syntax to create a test case and start its execution from the command line.

Secondly, Robot Framework is extendable; the user can create own keywords – which effectively behave like functions – each of which performs a user-defined action on the system under test. Thirdly, the user gets reports from a test that is run by Robot Framework. (Robot Framework 2019).

As stated, Robot Framework is modular, as it provides the framework for the test data syntax to pass the test data and the *application programmer interface* (API) to implement the test actions for the system under test, the communication link, as shown in Fig. 14.

So, the user can extend the framework by creating own keywords (functions); there are some built-in keywords exported by the Robot Framework itself but Python and Java programming languages are used to create test libraries, i.e. a unit containing multiple keywords. The keywords contain implementations that execute actions that test or validate the system under test either directly or by using a test tool. These keywords can then be called by a test case file that uses and has the access to the defined test library. (Robot Framework 2019).

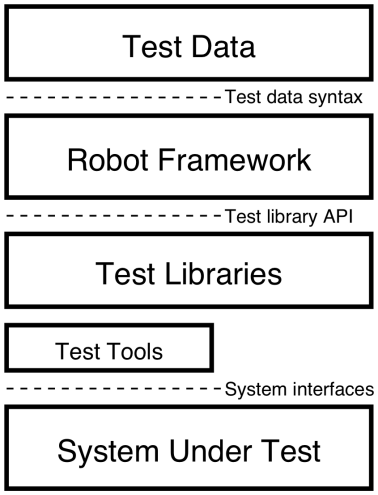

Figure 14. Robot Framework provides the test data syntax and the application interface (API) to create tests for the system under test. (Robot Framework 2019).

Robot Framework provides also reporting and logging tools to gather data about the test. While running a test, the user gets informed of the test execution by log messages in the terminal. After the test is completed, an XML (*eXtensible Markup Language*) and

an HTML (*Hypertext Markup Language*) files are generated which the user can view on a web browser to easily take a look how the system under test behaves. (Robot Framework 2019).

Following is an example of a Robot Framework test file, which tests a calculator application. Here, the example uses a custom made test library called *CalculatorLibrary*. The *test suite* contains three test cases called *Additions*, *Subtractions*, and *Calculation errors*, the first two of which have two steps and the last test case, *Calculation errors*, only one test step. These test cases are run in order when the test suite is executed.

As shown in Algorithm 1, a user can define keywords not only in the test library with Python or Java but also in the test file, under the "*** *Keywords* ***" notation (here, "*Calculate*" and "*Calculation should fail*"). The user can define the arguments that can be passed for the keywords and then the test steps that are executed when they are called. As can be seen, the keyword *Calculate* takes two arguments and calls the *Push buttons* keyword from the *CalculatorLibrary* which operates the action with the calculator application and also the *Result should be* keyword which validates the result.

```
*** Settings ***
Library           CalculatorLibrary

*** Test Cases ***
Additions
    Calculate    12 + 2 + 2    16
    Calculate    2 + -3        -1
Subtractions
    Calculate    12 - 2 - 2    8
    Calculate    2 - -3        5

Calculation errors
    Calculation should fail    1 / 0         Division by zero.

*** Keywords ***
Calculate
    [Arguments]    ${expression}    ${expected}
    Push buttons    C${expression}=
    Result should be    ${expected}
Calculation should fail
    [Arguments]    ${expression}    ${expected}
    ${error} =    Should fail    C${expression}=
    Should be equal    ${expected}    ${error}
```

Algorithm 1. Example of a Robot Framework test sequence, containing a test case and custom keywords.

The test library, CalculatorLibrary, can be implemented with Python to actually interact with the system under test, the calculator application, and it is implemented as a Python class, as shown in Algorithm 2. When a test is run by Robot Framework, it searches for the source files of the libraries that are defined in the test sequence, and calls the Python (or Java) function, when that step occurs in the test sequence.

```
from calculator import Calculator, CalculationError

class CalculatorLibrary(object):
    def __init__(self):
        self._calc = Calculator()
        self._result = ''

    def push_button(self, button):
        self._result = self._calc.push(button)

    def push_buttons(self, buttons):
        for button in buttons.replace(' ',''):
            self.push_button(button)

    def result_should_be(self, expected):
        if self._result != expected:
             raise AssertionError('%s != %s' % (self._result, ex-
pected))

    def should_fail(self, expression):
        try:
            self.push_buttons(expression)
        except CalculationError, err:
            return str(err)
        else:
             raise AssertionError("'%s' should have failed" % ex-
pression)
```

Algorithm 2. An example of implementing a custom library which provides functions that can be used in Robot Framework tests.

If the test sequence (in Algorithm 1) and the test library (Algorithm 2) are located in the current working directory, and Python and Robot Framework are installed, the previously given calculator test example could be run by executing a Python command

```
python -m robot calculator_test.robot
```

in the command line, where “-m robot” as a parameter means that a Python module called “robot” (Robot Framework) is used, and “calculator_test.robot” is the file name of the test sequence. The framework runs the test by using the keywords in the test file and in the test library, and after the execution, test result is printed on the console and

the reports from the test are generated for more in-detail inspection. As told, it is also possible to use Java programming language to implement the test libraries. However, because in this thesis, the test library is implemented in Python, any examples of how test libraries can be written and tests can be run with Java, are not described here.

In the previously described style, such Python keywords (functions) will be designed and implemented in the following chapters for the error generator system to inject faults. Of course, instead of pushing buttons of a calculator, they will send fault injection commands to the SoC-FPGA fault injector which will use the fault injection FPGA block to inject the faults.

Now that all the relevant theory for the error generator system is studied, the details of the system are discussed. After this, the design and the implementation of the system can begin.

3 ERROR GENERATOR SYSTEM

Figure 15 depicts the proposed structure of the error generator system. This chapter describes each part of the error generator system in more detail in the subchapters.

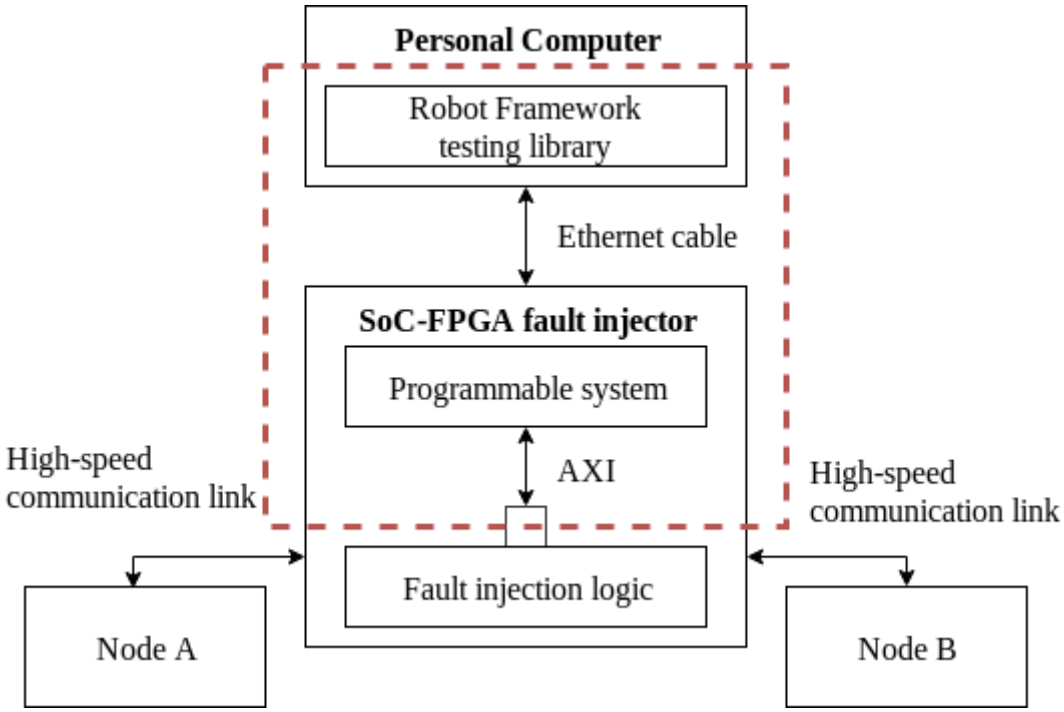

Figure 15. The parts in the proposed solution of the error generator system. The figure is already introduced in Subchapter 1.2 about the objective of the thesis. The objective contains creating the components of the system that are inside the red dashed-line rectangle in the figure, thus excluding the fault injection logic block and the communication link or the nodes.

The focus of the description is more on those parts of the error generator system that are in the objective of the thesis. As a recall, these include the parts inside the red dashed-line rectangle in Figure 15:

1. *Robot Framework test library (RF test library),* which will provide the logic that makes possible to create and run fault injection tests. Manual test templates are also provided. They can be used as reference when running the tests.
2. *Communication between the test library on a personal computer (PC) and the SoC-FPGA fault injector.* The communication is carried out over Ethernet with the UDP/IP protocol and is shown in the Figure 15 as *"Ethernet cable"*.

3. *Communication between the programmable system and the fault injection logic*. The programmable system will receive injection requests from the Robot Framework test library over Ethernet and pass them to the injection logic by using the provided AXI interface. Possible responses from the fault injector should also be handled properly.

The other parts of the system (high-speed communication, the communicating nodes and especially the injection logic block) are also described shortly but they are not implemented in this thesis. They are delivered either during the thesis or later by Danfoss. These parts are not needed for the implementation phase because the previously described parts can be implemented and tested separately.

Creating the software for the programmable system depends on the fault injection logic IP (intellectual property) block. However, this does not block the development process. It can be implemented even if the block were not delivered as long as the description for the AXI interface is provided. This way the communication over the AXI interface between the programmable system and the injection logic FPGA block can be designed and implemented.

3.1 Danfoss test automation system

The fault injector will be a part of the Danfoss test automation system as shown in Fig. 16. With the error generator system, it can be validated that the communication between the nodes works expectedly.

All the three parts designed and implemented (the SoC-FPGA fault injector, the Robot Framework test library and the communication between them) are used in the test automation system. Firstly, the Robot Framework test library is downloaded into a server machine which will automatically execute tests. Secondly, as can be seen in Fig. 16, the SoC-FPGA fault injector will be connected into the test automation server (instead of a personal computer) when it is integrated in the test automation system. Finally, the implemented Ethernet connection will be used in the test automation system to enable communication between the RF test library and the SoC-FPGA fault injector.

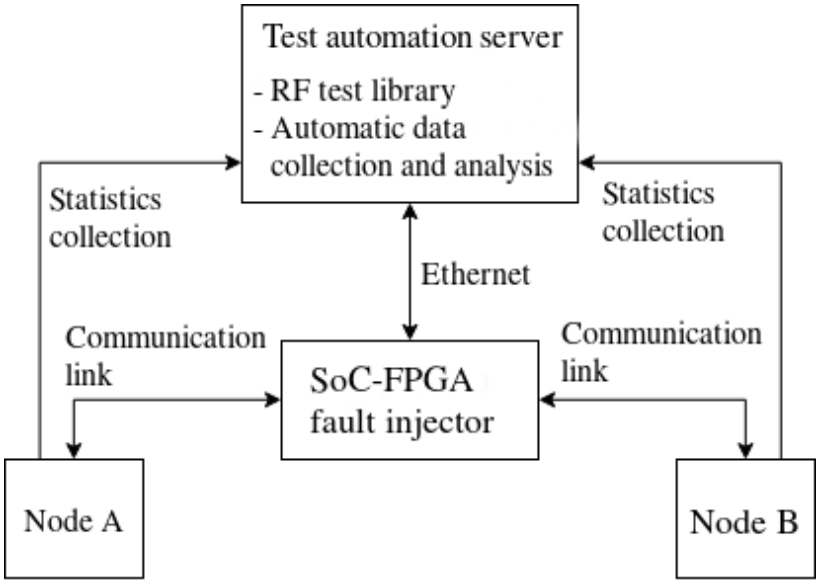

Figure 16. The schematic view of how the test automation system looks like. The error generator system designed (RF test library, Ethernet communication and SoC-FPGA fault injector) in this thesis will be a part of it.

In the test automation system, the tests are run and the results are collected automatically. Automatic tests with comparison with manual tests are discussed more in Subchapters 3.4.2 and 3.4.3.

3.2 High-speed communication link

A serial communication link will be connected to the fault injector for fault injection testing. The parties in the communication link are FPGA IP blocks inside the nodes, e.g. option boards of a frequency converter. The nodes transmit real-time data and configuration data between each other over this serial communication link. However, the contents of the data are irrelevant from the viewpoint of the thesis.

The high-speed communication link uses a proprietary communication protocol developed in Danfoss. It uses a *8b/10b encoding* which is an extended line coding. It maps 8 bit words into 10 bit words which decreases the noise-to-signal ratio and reduces the DC offset because, in average, the number of 0 and 1 bits in the data are in balance after additional bits are inserted. Furthermore, with the increased number of bits, there are enough state transitions for synchronisation without significant increase in bandwidth. (Widmer & Franaszek 1983: 450).

The packet, or the message, in the protocol consists of a message target address, data and a checksum. With the target address, the receiving IP block can route the message internally to the correct recipient within the block. The data integrity is validated with a CRC checksum and the most critical messages also include a sequence number.

In the protocol, the error management relies on the error detection by the CRC. The receiver drops a message if it is detected invalid by the CRC block. However, the receiver does respond with neither a positive or a negative acknowledgement message upon receiving. This is because the transfer speed is extremely important in the communication. To increase the speed and to limit the bandwidth usage of the communication link, automatic repeat requests (ACK messages) are not sent, or invalid messages are not retransmitted. Nevertheless, to be robust, the transmitter should send a critical messages twice.

3.3 SoC-FPGA fault injector

The fault injector itself is a SoC-FPGA, as shortly described in the objectives of the thesis. The SoC-FPGA fault injector will contain the programmable system (a processor) and the injection logic IP block (FPGA) developed by Danfoss. The communication link, where the faults will eventually be injected, is connected to the latter, the injection logic block.

3.3.1 Fault injection logic of the SoC-FPGA fault injector

The fault injector logic block is an FPGA module that will be developed by Danfoss and will be integrated into the SoC-FPGA fault injector. It takes the serial data in, injects a fault into the correct package depending on the values passed from the programmable system over the AXI interface, and passes either unmodified or modified serial data out.

The injection logic will be connected to the programmable system as an IP block by using the provided AXI interface. The AXI interface of the injector logic block will (at least) contain the following data input: *the fault type*, *the target address* and *the number of repetitions*, i.e. how many times the error will be injected. With the target address input, the injection logic finds a message which contains the given target address and injects the fault to the message. The fault type is one of the following:

1. CRC fault: The injection logic flips bits in the message data so that the CRC calculation is violated. The CRC field is kept as is.
2. Change target address: The injection logic changes the target address of the message. The receiving node would route the message according to the address to a wrong block inside the node.
3. Increasing/decreasing the length of the message: The injector removes or adds data in the message.
4. Duplicate: The message is sent twice with same contents.
5. Removing the message: The message is lost by the injection logic and thus not received.

The nodes should detect the errors (e.g. by using the CRC sum) in the messages and react accordingly. The response to an invalid message depends on i.a. how critical the message and the fault are and thus cannot be generalised. In extreme cases, precautionary measures, i.e. motor shutdown, may be carried out by the receiver if an error is detected.

The injection logic provides data output for the programmable system, too. The injection logic acknowledges a successful fault injection to the communication link by passing an acknowledgement over the AXI interface to the programmable system. The acknowledgement contains the type of the fault that was injected. The programmable system should handle these responses properly and pass the acknowledgement forward to the Robot Framework test library.

However, the injection logic can only return data about the fault injection process itself. It can give information if the fault could not be injected but it does not know whether the receiving node processed the fault correctly. This should be managed by the enclosing framework, the test automation system (see Subchapter 3.4.3 Automatic tests).

3.3.2 Communication between the fault injector and the test library

The fault injector is connected to Robot Framework test library on a PC over Ethernet. The communication is built on UDP/IP. Thus, the processing system of the fault injector must be able to handle UDP datagrams and parse the request of the datagram sent by the

Robot Framework test library. Furthermore, the fault injector system has to be able to send data in the other direction, back to the PC, too.

Even though TCP/IP alternative would have been a more robust alternative for communication between the fault injector and the test library, UDP was selected because it is more lightweight and faster as it was discussed and compared in Subchapter 2.4.2.

Furthermore, and more importantly, the SoC-FPGA board of the fault injector will be used for other operations (outside this thesis) where UDP will be used. So for better maintainability and extensibility, using UDP was included in the specifications so that only one protocol would be used in the SoC-FPGA fault injector on the transport layer. As a conclusion, UDP was selected mainly for the latter reason, and no additional comparison is made in this thesis on the speed, the bandwidth-usage and the reliability between TCP and UDP.

In this thesis, a simple application protocol is built on top of the UDP to send data between the RF test library and the fault injector. Without it, the contents of datagrams sent by UDP are meaningless themselves. The protocol should describe the format in which the requests to the fault injector and the responses from it are represented in a UDP datagram. If the fault injector needs setup data, the communication protocol should also define the format for it.

3.4 Test library with Robot Framework

3.4.1 Robot Framework keywords for fault injection

Robot Framework keywords are written in Python programming language as Python functions. There is one keyword for each of the five fault types described in Subchapter 3.3.1. Each keyword takes also a *repeat number* and a *target address* as their parameters, so that the number of injected faults and the target address of the message can be configured. The flow of the keyword is shown in Fig. 17.

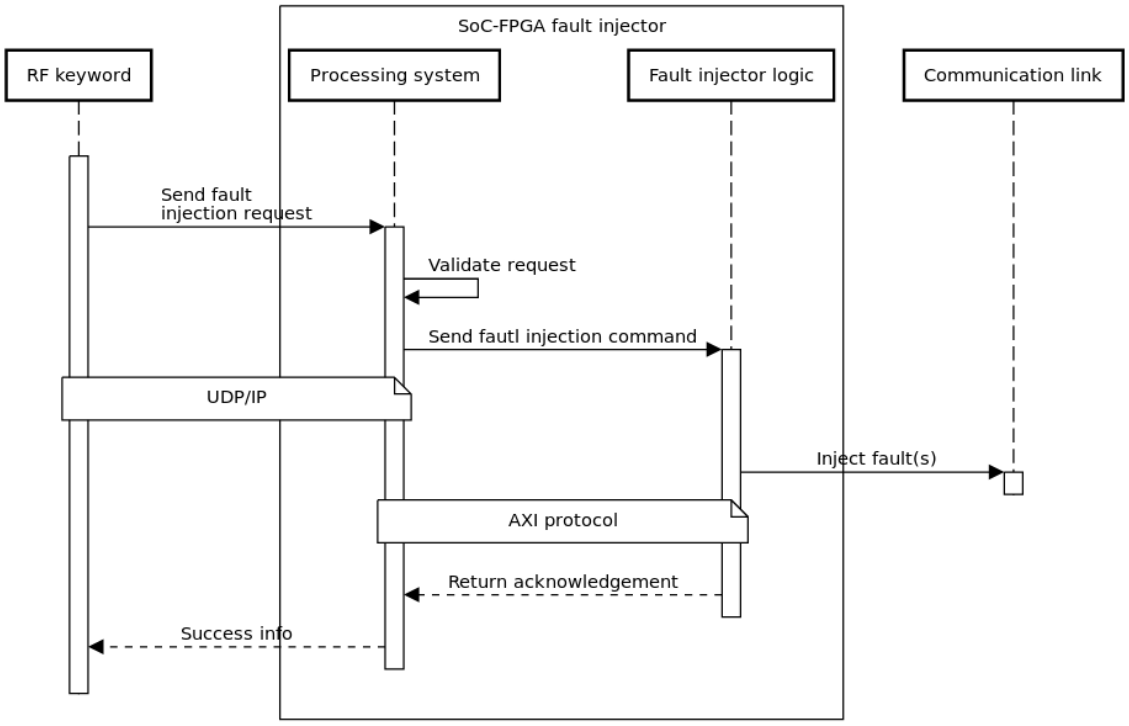

Figure 17. A sequence diagram describing how the data flows from a Robot Framework keyword (RF keyword) to the fault injection logic and eventually into the communication link as a generated error.

All the keywords have somewhat same functionality. When the keyword is executed, a UDP datagram containing the fault type, repeat number and address is generated and sent over Ethernet to the fault injector. Before sending the data, the keywords wrap the information in a datagram that has a predefined format so that the injector can read and understand the data. This format will be specified in Subchapter 4.2.

After the datagram is sent, each keyword should wait for a UDP response datagram from the injector. The fault injector validates the request that is sent. It also gets the response from the fault injection logic that the fault was injected successfully. If the response was positive, the injector sends a UDP datagram containing a positive acknowledgement, otherwise a negative acknowledgement. Thus, if the keyword receives a negative acknowledgement or does not receive any acknowledgement within a defined time period, it should mark the keyword execution as failed. This is done by raising an exception; the built-in features of Robot Framework then stop the test execution and mark the test as failed with a message that is given by the keyword.

In addition to the five keywords which are used to inject faults, there will be additional keywords to execute setup and utilisation functions on the fault injector. Setup keywords can include changing IP addresses and ports of the fault injector or fetching version information.

3.4.2 Manual (targeted) tests

In manual tests in general, the person running a test, a tester, manually runs a test by following a predefined test plan. Furthermore, the tester reads manually the output of the test program and validates the output to determine if the test was successful or not.

In the fault injection testing, a Robot Framework test file is started manually by the tester on a PC. The test sequence in the test file contains a list of fault-injection keywords that are to be executed. Then, after the test has finished, statistics are collected from nodes of the communication link, and they are manually read to validate that the fault detection and management behaviour of the nodes worked expectedly. Also, during the test execution, the tester should monitor log messages of Robot Framework to see whether the actual fault injection operations are executed successfully.

With targeted test, the tester can specify which faults should be injected. This can be useful, when only certain fault types or a specific target address needs to be tested. However, unlike in automatic tests, the tester needs to be present during the test and has to read the test output manually. Also, unless the test results are in a clear format, the tester must know how to interpret the read test output correctly to define the result of the test.

3.4.3 Automatic tests

With automatic tests, a test process is done completely automatically. Tests execution are started and executed automatically either by a trigger or at a specific point of time, e.g. once a day. The results of the tests are read automatically and stored in a human-readable format. Usually, if the automatic tests are run periodically, the operator of the automatic test system is notified in case the test were not completed successfully, so that proper actions can be carried out.

The fault injection tests will be executed in the test automation system. The test automation system will consist of a server machine, which has a script that will run a Robot Framework test sequence automatically. The sequence contains all the fault injection tests so that all faults will be injected and tested automatically. After the test has run, the test automation system is capable of collecting the statistical data from the nodes to detect, whether the faults were detected and handled correctly in the receiver. Collecting statistics about the communication and analysing them is done outside of the error generator system and thus is outside the scope of the thesis and not discussed here in detail.

The advantage is that the tests could be automatically run when a new software binary is downloaded into a node. Because a tester does not have to be present, tests with longer duration and a wide range of target addresses and repetition count can be automatically executed overnight and results can be read next day. However, running the automatic tests here requires that the development of the test automation system – which this error generator system will eventually be part of – is finished. Because that development process is not the objective of the thesis, it is possible that results of automatic tests cannot be included in this report.

The components of the error generator system are now described. As a recall, the error generator system consists of

1. Robot Framework test library, which will be used to create and run tests and which communicates with the SoC-FPGA fault injector over Ethernet,
2. SoC-FPGA fault injector, which has a processing system (a processor) and a fault injection FPGA block, latter of which is not created in this thesis, and
3. the serial communication link and the nodes, e.g. option boards, which communicate over the link, and which are not created here but provided for the testing phase.

Now that the system is described, it is designed and implemented for those parts that are included in the objective. Those include the RF test library and the SoC-FPGA fault injector and the UDP/IP communication between them.

4 DESIGNING THE ERROR GENERATOR SYSTEM

This chapter describes how the error generator system is designed. The design process proceeds hierarchically, starting from an abstract structure description of the system, eventually ending into designing the detailed flow of a single algorithm. Design tools such as component diagrams, class graphs and pseudo code are utilised in the design process.

The design starts with defining the top-level components and their responsibilities for the error generator system. From the requirements in the Subchapter 1.2 describing the objectives of the thesis and the description of the system in the previous chapter, three main components of the design could be identified easily. The initial component hierarchy is shown in Fig. 18. All these three main components are then designed individually and the design is described in the following subchapters.

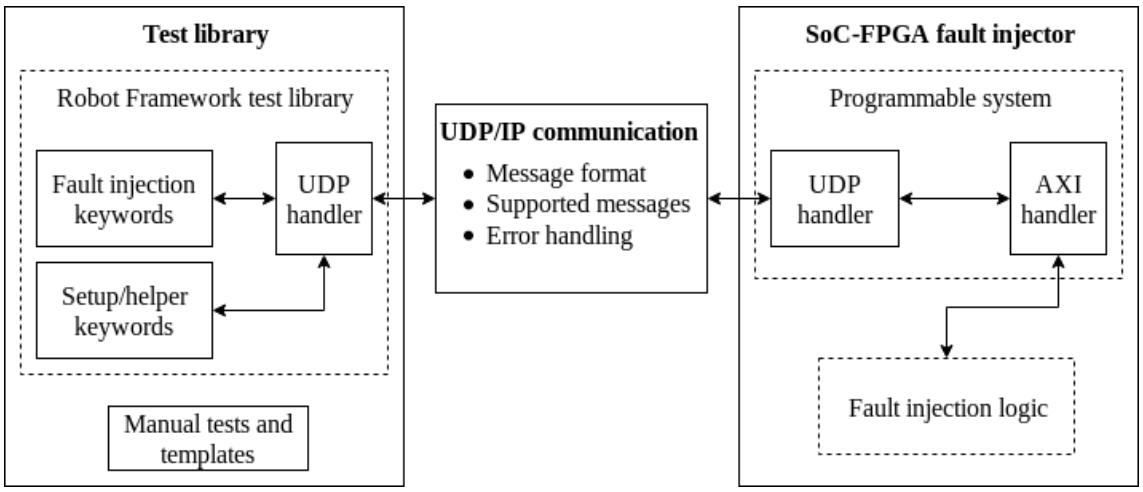

Figure 18. The objective consists of designing the test library, (the programmable system of) the fault injector and the UDP/IP communication between them. As stated, the fault injection logic block (the dashed rectangle on the right) is not in the objective, but will be used in the ready system.

The UDP/IP communication is the backbone of the whole system which is why it is designed first. It defines the message format and the supported messages for the communication to make the communication between the test library and the fault injector possible. To increase the robustness, an error handling mechanism can be designed, too.

After the UDP/IP communication is designed, the test library and fault injector design process can begin.

The Robot Framework test library has a UDP handler block, which supports the previously mentioned UDP/IP communication and its message formats. It is responsible of sending the messages and receiving them. Eventually, the messages that are handled by the UDP handler, are sent and received by the Robot Framework keywords. They are split into two sets; the fault injection keywords execute a specific fault injection test and the setup and helper keywords change settings in the fault injector or provide additional functionality. These all are contained in a Robot Framework Test Library, implemented as classes and functions in the Python programming language.

The test library also has example test sequences implemented as Robot Framework test files. The test files are templates to show how the fault injection tests and setup functions in the Robot Framework test library can be called in a Robot Framework test file. This template can then be used to create and execute manual tests in the testing phase.

Like the RF test library, the programmable system of the fault injector contains a UDP handler module which handles the incoming and outgoing UDP datagrams. The UDP handler of the PS then parses the request and calls the AXI handler. The AXI handler sends the fault injection request to the fault injection logic block over the AXI interface, gets the response and returns it back to the UDP handler.

4.1 Design of the UDP/IP communication

Here, the design of the communication with UDP between the RF test library and the SoC-FPGA fault injector is given. First, to ensure reliable UDP communication, a method to increase robustness is selected and fine-tuned. Then, the actual format of a datagram is designed by first recalling the requirements and then defining the format.

4.1.1 UDP communication data flow

As already stated, the UDP protocol is a connectionless protocol. This means that to send a datagram, a communication between the sender and the receiver does not need to

be established. This increases speed but the sender cannot be sure whether the receiver has got the datagram. Our protocol over the UDP communication must address this issue somehow.

A simple way to ensure a successful transmission is to use acknowledgements. This, however, would require that acknowledgement and the retransmission procedures are designed, which causes extra work. This would also use extra bandwidth and cause delay which decreases the advantage of using the UDP. Furthermore, and most importantly, defining an acknowledgement procedure in our protocol would be reinventing the wheel since such mechanism is already implemented successfully in TCP.

To understand the effects of the errors on the communication link, different error handling mechanisms were studied in Subchapter 2.2.3. This information, however, turns out to be useful when designing the UDP communication. For blind transmission, it was given that "*to reduce the number of lost packets, the transmitter can send a packet multiple times and 'hope' that at least one of them will receive the destination"*.

So in our UDP protocol, all the messages are sent three times so that at least one datagram will be received properly. However, to avoid duplicates, the message must contain an increasing message identifier number, so that only one message is handled and the remaining messages with the same identifier are ignored. This is to avoid accidentally generating a same fault multiple times in the fault injection logic block; the error generator system could provide ambiguous results in testing, when a successful transmission of a package was expected in the link, but an error occurs unexpectedly because of a duplicate fault injection command processed by the SoC-FPGA fault injector.

4.1.2 UDP datagram structure

All the data is sent as characters in the datagram. This way, the implementation can be made faster when the datagram contents are in a more human-readable format in strings of characters instead of plain bytes. The most of the datagrams are created in the Robot Framework test library on Python where manipulating character sequences (strings) is fast and easy, though manipulating bytes is possible, too. Sending the data as characters

significantly increases the datagram size compared to sending as “raw” bytes, so they must be kept as short as possible.

However, recalling from Subchapter 2.4.2, Figure 13, that the headers of a UDP datagram already take up 20 bytes of the total size. Using the binary format does compress the size of the payload but the relative decrease in the total size of the datagram is not significant, if the length of the data in the string format would be around the length of the header, as it is in this use case.

As previously stated in Subchapter 4.1.1, the datagram should contain an identifier field to avoid duplicates. This is done by starting the datagram with a one-character (byte) field containing a number from 0 to 9 (the message identifier field). The message should start with this identifier and the receiver should drop a datagram with an identifier as the same as that of the previous datagram. No action is needed when the message identifier shifts by more than one value between the consecutive datagrams.

To identify, what kind of message is sent by the datagram, the datagram type field is appended to the datagram after the message identifier separated by a space character. Four characters are reserved for this field for extensibility purposes. The message type can be for example a request to inject a fault or it can be an acknowledgement from the fault injector that the fault injection was successful. These types are discussed more in Subchapter 4.1.3.

Furthermore, some messages need additional information, i.e. parameters. These are added to the end of the datagram, again after a space character separating them from the previous field. If more than one parameter is sent, they are separated by a space.

The datagram format is now complete. The full datagram format is shown in Table 4. As can be seen, the minimum length of the datagram is six (6) characters if the message type does not require parameters. Thus, if the receiver receives a datagram shorter than that, it can safely ignore it. The maximum length of the datagram is defined later, when the message types are defined, because the maximum length of a datagram depends on the maximum length of the combined parameters at the end of the datagram.

Table 4. The datagram format with the field names and the length for each field.

Field	Message identifier (0–9)	Space	Message Type	Space	Parameters
Length (chars)	1	1	4	1	1–…
Required	yes	yes	yes	no	no

4.1.3 Message types for a datagram

After the datagram format is designed and defined, the contents of a datagram are defined. These include defining the message types and the possible parameters of a message.

The fault injection keywords of the test library send the fault injection requests to the fault injector. Thus, because there are five possible types of a fault, each type for a fault injection must have its own message type. The fault injection needs also the target address and the number of repeats to be successful, as stated in the description of the fault injection logic, so these two values must be sent as the parameters of a fault injection datagram. Again, the type identifier is defined as a string of characters to make it easier to develop the communication handling in the test library and in the SoC-FPGA, and also, when later testing the UDP communication, to make debugging the received and sent datagrams easier.

Furthermore, from the *setup and helper keywords*, the following data can be sent to the fault injector: set IP address, set UDP port, set communication link type and get the version information of the fault injection logic IP block. Here, setting the IP address and the port refers to the address and port of the PC running the Robot Framework, so that the fault injection logic knows in which address send the data back. Naturally, these two take an IP address or a port as their parameter. The message type to set the communication link type has one numeric parameter that describes the communication link type.

Finally, there should be a possibility to send a “test started” message from the RF test library to the SoC-FPGA fault injector at the beginning of a test. With this, the fault injector can reset its message identifier number, so that if the very first message of the test sent from the Robot Framework is the same as that of the last test, the datagram is not accidentally ignored as a duplicate.

All the possible messages that can be sent from the RF test library to the SoC-FPGA fault injector are defined. The message types with their four-character identifiers and the possible parameters are listed in Table 5.

Table 5. The proposed message types and parameters from the RF test library to the SoC-FPGA fault injector.

Message description	Message type string	Parameters
(Inject) CRC fault	`fcrc`	target address, repeats
Address change fault	`fadr`	target address, repeats
Length change fault	`flen`	target address, repeats
Duplicate message fault	`fdup`	target address, repeats
Drop message fault	`fdrp`	target address, repeats
Set IP address	`sipa`	IP address
Set UDP port	`spor`	UDP port
Set communication link type	`sclt`	communication link type
Get injection logic version	`gilv`	
Start test sequence	`strt`	

Also, the fault injector sends data via UDP back to the Robot Framework test library. These messages follow the previously defined datagram format. The fault injector returns the positive or negative acknowledgement about the injected fault, and the fault injection logic block version, when requested. The message types with their parameters are shown in Table 6. To keep the consistency with the four-character message type definition, an underscore is prefixed at the beginning of the three-letter message type identifiers.

Table 6. The proposed message types and parameters from the fault injector to the RF test library.

Message description	Message type string	Parameters
Positive acknowledgement	`_ack`	
Negative acknowledgement	`_nak`	
Injection logic version information	`_ilv`	injection logic version

The UDP handler modules of the RF test library and the programmable system on the SoC-FPGA fault injector should follow these datagram format definitions. For example, when a request to inject a CRC fault 5 times to address “12345678” is sent from the RF test library, a following datagram will be sent three times

```
0 fcrc 12345678 5
```

and if the fault was injected successfully, the fault injector would respond with

```
0 _ack
```

4.2 Design of the Robot Framework test library

The Robot Framework test library consists of the test libraries and a template test sequence or sequences. Furthermore, the test library will have fault injection keywords (Robot Framework’s “functions”) and set-up or configuration keywords. A UDP handler will be included in this Robot Framework test library to provide communication, but it will a separate Python module or class and not directly in the test library.

The test sequences should contain templates to show how the keywords are used when running a test with Robot Framework. There should be templates and examples how to create and run a fault injection test and how to use the setup functions. These test design templates can be designed either here on in the test preparation phase. This hierarchy and the relationships the parts is depicted in Figure 19.

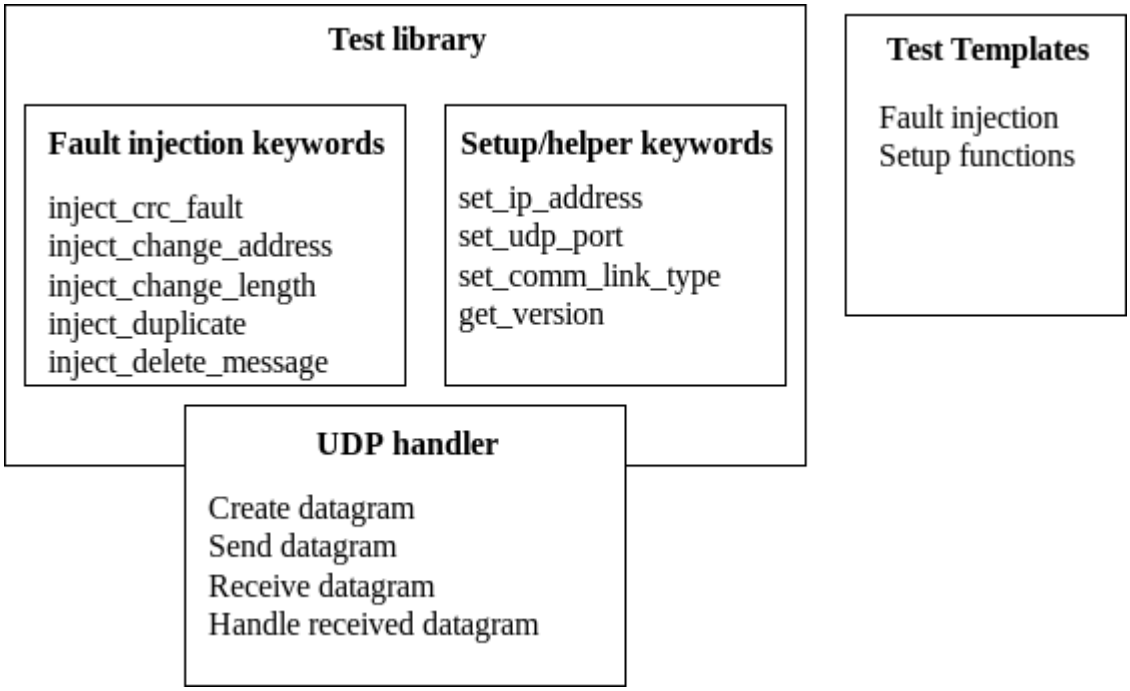

Figure 19. Components of the test library. The test library utilises the UDP handler component to execute the keywords. The test templates are test sequences with examples how to call the keywords.

4.2.1 Design of the fault injection keywords

Effectively, all the fault injection keywords are the same meaning that the body of a Python function is almost identical for all the functions. Only the type of the fault injection changes, which, as described in Subchapter 4.1.2, means that the message type is different. Thus, this nuance can be ignored when designing all the keywords at once.

The flow diagram describing how the process in a fault injection keyword proceeds, is depicted in Fig. 20. The elements that are on the right side of the dashed vertical line represent actions of which logic is encapsulated in the UDP handler component, and, thus, are designed later.

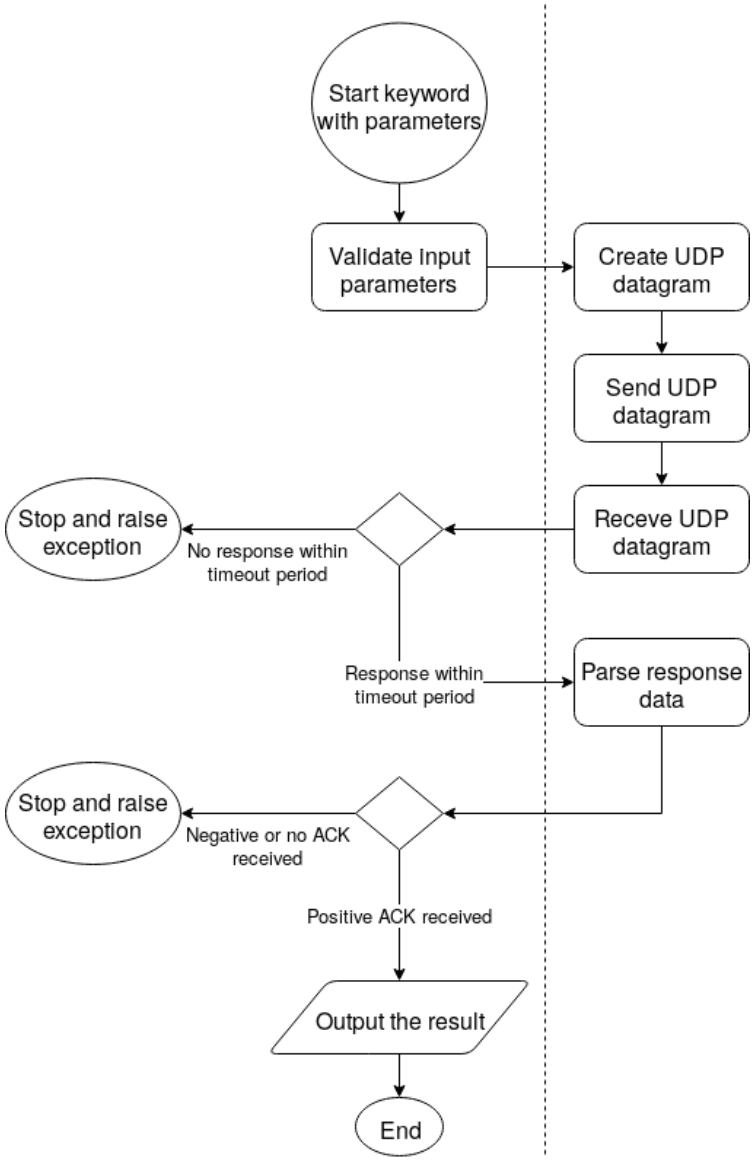

Figure 20. The flow diagram of a fault injection keyword. The steps located on the right side of the dashed line represent the actions that happen in the UDP handler which will be designed later.

Firstly, each fault injection keyword takes a target address and the number of repeats as their parameters. First, the parameters should be validated when executing a keyword. Currently, no requirements are specified for the target address so any non-empty value should be accepted. Nevertheless, the number of repeats should have a numeric value that is greater than 0. Maximum value is not specified – unless more specifications are provided.

After the parameter validation, the UDP handler is utilised, as can be seen in Fig. 20. The parameters, and the fault injection type which is included in the keyword itself, are

passed to the UDP handler to create a datagram with correct format. Then, this datagram is sent by the UDP handler, which takes the responsibility to send the datagram as it is designed. All the errors in this phase should be considered fatal and lead to stopping the execution of the keyword.

The keyword waits for the acknowledgement after it has sent the fault injection request. Yet again, the UDP handler receives the datagram and returns it to the keyword. If the datagram was received within a configurable timeout, its contents are read by the UDP handler. If the response is a positive acknowledgement, the error injection was successful and this is output to the user. However, if anything else is received, the injection is assumed to be failed. In case of fail (timeout or a non-positive acknowledgement), an error should be thrown so that the execution of the whole test case will fail.

4.2.2 Design of the setup keywords

As already shown in Figure 18 and Figure 19, there are setup and helper keywords modify or fetch the values of the error generator system. Firstly, the *set_comm_link_type* keyword should send a datagram to the fault injector to configure the type of the communication link that is connected to the fault injection logic block. The flow of this keyword follows the logic of the fault injection keywords. The input parameter, communication link type, is validated and a datagram is created and sent by the UDP handler. However, nothing is returned from the fault injector after the link type is set. So, after the “Send UDP datagram” step in Fig. 20, this keyword will reach the end.

The *get_version*, of which logical flow is depicted in Figure 21, will fetch the version of the fault injection logic IP block from the fault injector. Thus, it needs to make a request by sending an UDP datagram and handle the response. However, because it does not have any parameters, no validation step is included before sending the datagram. After the response has been got, it is validated and printed to the user if it was received successfully within a timeout period.

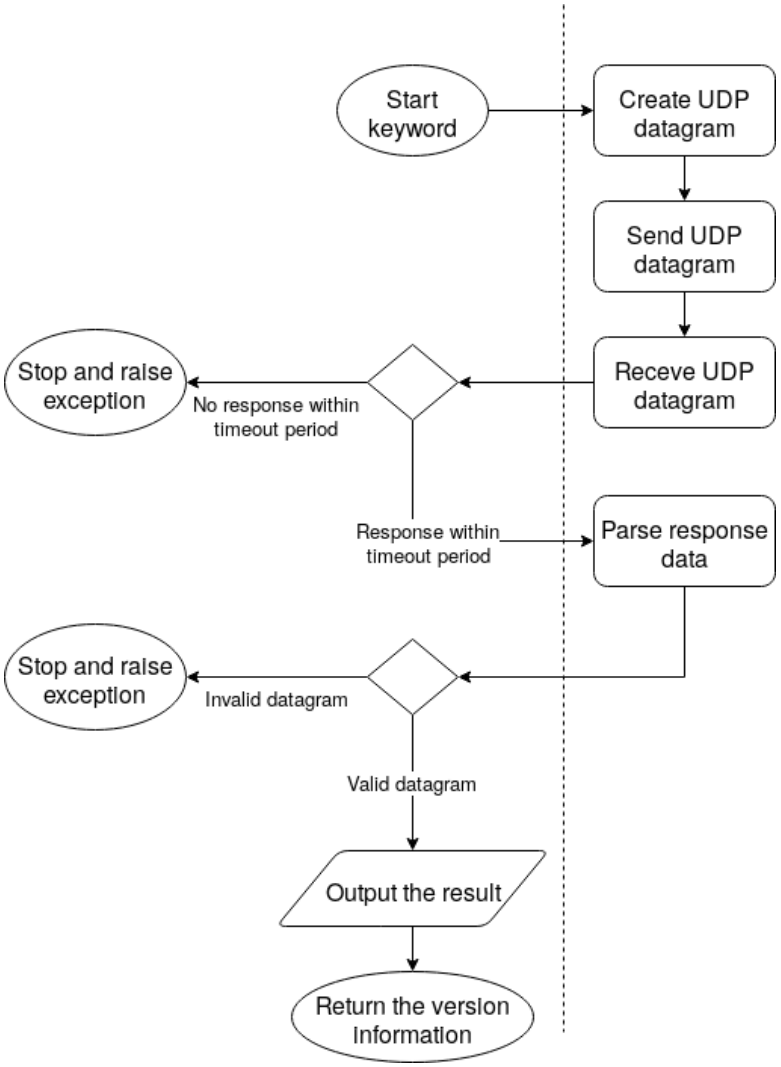

Figure 21. Flow diagram of the keyword that will get and return the version of the fault injection logic IP block. Here, "valid datagram" means that the received datagram had the type "*_ilv*" which indicates that the datagram contains the injection logic version.

However, as can be seen in Fig. 21, because the keyword fetches data, the version information should be returned from the keyword. Returning the value from the keyword makes it possible to handle this value inside a Robot Framework test sequence and create conditional tests depending on the version of the fault injection logic block. For example, a following Robot Framework test sequence could be created with which reads the version of the injection logic block into a variable called *version* and executes a test step depending on the version

```
${version}=  Get Version
Run Keyword If
    ${version} == '0.2' Inject CRC Fault  1234  2
```

This snippet of a Robot Framework example test sequence would inject a CRC fault twice to address “1234” if the fault injector version was 0.2. Otherwise it would skip the CRC fault injection. Robot Framework’s built-in function “Run keyword if” is used here and it takes a condition as its first parameter and the keyword which will be executed if the condition was true as the second parameter.

Two setting keywords, set IP address and set UDP port, are yet to be designed. To design the two, it must be clear that setting an IP address and port can be done for both “upstream” and “downstream”. Here, the upstream address and port refer to the destination of the datagrams from the RF test library to the fault injector. Thus, the upstream address and port should be that of the fault injector. In contrary, setting the downstream address and port defines, where the fault injector will forward its datagrams. So, both the keywords have two parameters; the IP address or UDP port, and an additional parameter to define whether the setting configuration is done for “upstream” or “downstream” values.

Setting the upstream values do nothing more than modify only specific variables and no data is sent over UDP. The IP address or port value is first validated, and then the value is stored in a variable, so that the UDP handler will send the next datagram using the defined destination. However, configuring the downstream address and port values require sending a UDP datagram to the fault injector, which means that the upstream values must be configured before. Furthermore, when the downstream port or address is changed, the UDP handler must start to listen for UDP datagrams on that endpoint instead and close the previous UDP socket. This logic is more comprehensible in the flow diagram in Figure 22.

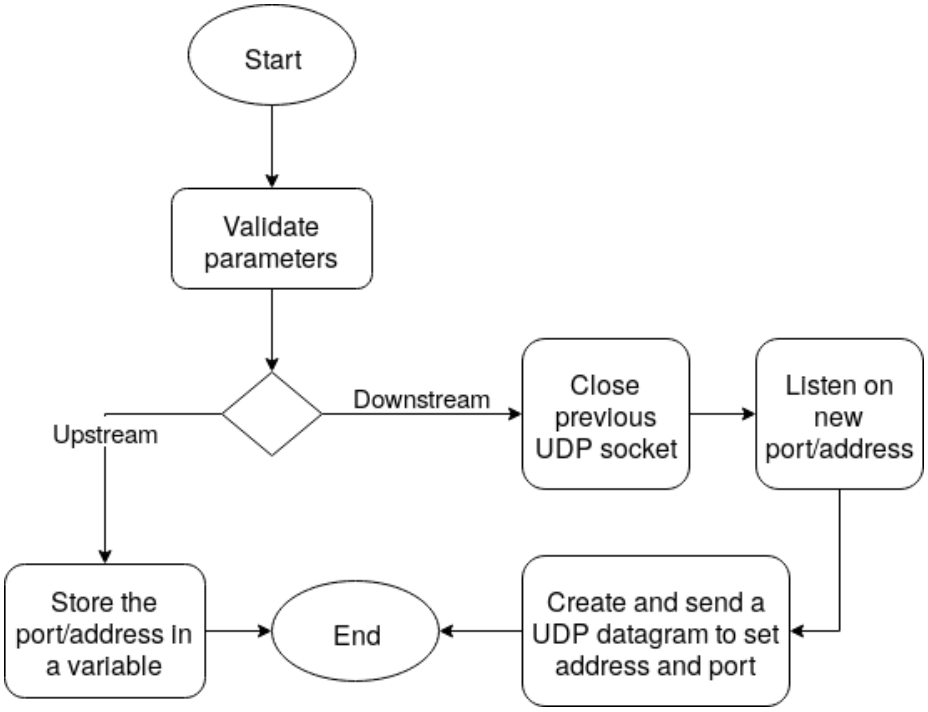

Figure 22. The flow diagram of ”set_ip_address” and ”set_udp_port”. For clarity, creating and sending a UDP datagram processes are compressed into a one block. The operations ”Close previous UDP socket” and ”Listen on new port/address” are encapsulated inside the UDP handler module.

4.2.3 Designing the UDP handler

Sending and receiving UDP datagrams to and from the fault injector is encapsulated in a module called UDP handler. It is responsible of sending datagrams to the correct address to the SoC-FPGA fault injector. It also receives the datagrams sent by the fault injector. The datagrams must follow the designed UDP/IP communication paradigm: the UDP handler sends datagrams according to this specification and validates that the received datagrams follow the definition. Finally, there must be an access point, an interface to the UDP handler, so that the Robot Framework test library can use the methods and consume information that is handled by the UDP handler without caring about how the logic is carried out internally.

The design starts by identifying the main components of the UDP handler. Since the responsibility of the UDP handler is to send and receive datagrams, it is natural that the module is split into separate components for receiving and sending. The UDP handler

module will contain a UDP handler class, which is the main class and the interface to the module for the Robot Framework keywords. Thus, it has simple functions for both sending a datagram and getting the last received datagram. Furthermore, it has setup functions to configure the upstream and downstream endpoints, and it will contain the later-described UDP receiver and sender instances in its properties to send and receive the data. Upon initialisation of this class, it will also initialise the later-designed UDP sender and receiver for starting the communication. Full contents of the class can be seen in Fig. 23.

As can be seen in Fig. 23, this module also has a UDP datagram class, of which instances represent a single datagram. An instance encapsulates the needed information for a datagram, and because of the object-oriented format, the contents of a datagram are more accessible by the Robot Framework keywords that if it would be a plain string of characters. With this, unnecessary and redundant parsing of a datagram is avoided. Also, creating a new datagram for sending is more convenient than manually creating the contents of a datagram for each sent item.

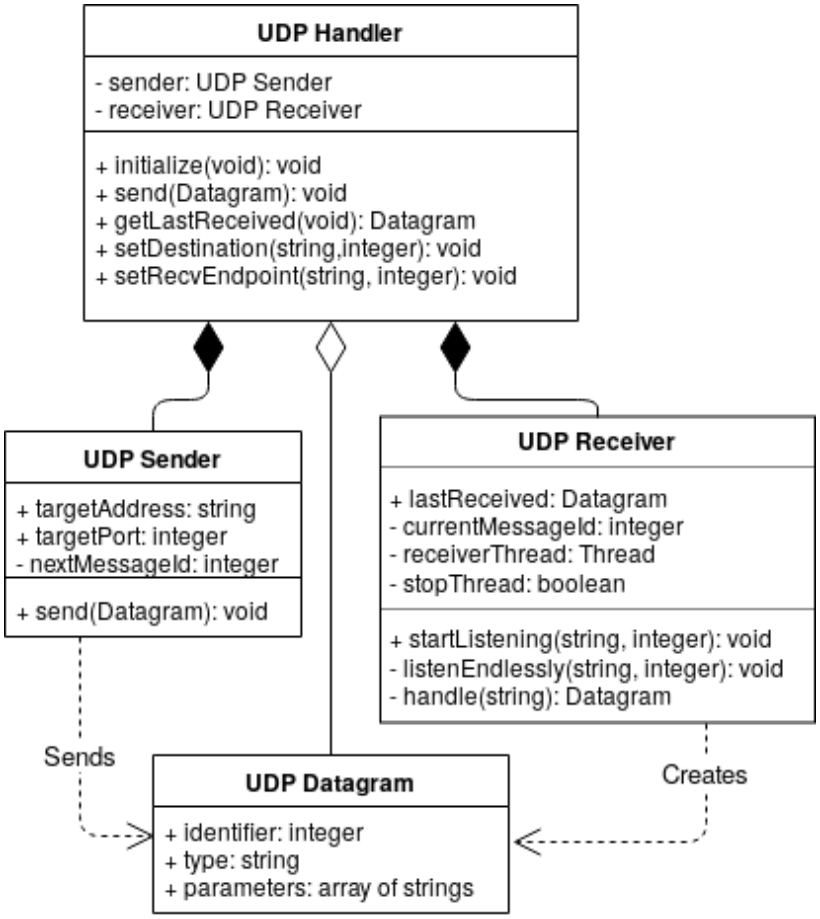

Figure 23. The class diagram of the UDP handler module. The UDP handler class is the top-level class and the entry point for UDP-related operations for the test library; it contains a sender and a receiver, which have functions of handling datagrams in one direction. The methods and the fields of the classes are described in this subchapter.

The UDP sender has the target address and target port for the destination which can be changed. It also contains the message id that will be increased for each message sent. When the send function is called, the sender will convert the given UDP datagram into a string of characters and finally sends it multiple times to the destination, as previously described in the Subchapter 4.1 about the UDP communication. It also makes sure that the identifier of the datagram is increased according to the specification after each sent datagram, so the receiving end will distinguish new datagrams from duplicates.

The design of the UDP receiver class is more complex than the sender class. This is because it is designed to run in a separate thread, i.e. in parallel at the same time with other functions. This way, the receiver can constantly listen incoming datagrams. The other alternative would be sending a datagram and then starting to listen for incoming

datagrams. This is also possible because it is the Robot Framework libraries who are always initiating the UDP communication. This means there cannot be a received datagram, a response, from the fault injector unless a datagram, a request, has been sent to the fault injector first. However, running the receiver in a separate thread allows more modular design when sending and receiving functionalities are divided in different classes. Also, constantly listening in parallel means that the main thread is not blocked because of receiving a datagram. Anyhow, if any problems with this design are encountered during the implementation phase, the alternative design (first send and then listen) can be used.

The “startListening” function of the UDP listener will begin a new thread and start listening packets after checking that the previous thread is not running. The target port and address are passed as parameters to the function as described in the following pseudo code function:

```
Check if already listening.
If already listening,
    request to stop listening and wait for one second.
    If still listening, throw an error and exit.

Create a new thread.
If IP address and UDP port not defined, use default values.
Listen endlessly in the another thread with the address &
port.
Start the thread and store it in receiverThread variable.
```

The “listenEndlessly” will be executed in a separate thread, as described in the previous pseudo code design. It will create a socket and constantly listen for incoming packets. If a datagram is received, it is passed to a *handle* function, which does the conversion from a plain string to a UDP datagram object, and also validates that the contents of the datagram for correct format and for duplicates. As pseudo code, the logic of the function can be described with the next pseudo code snippet:

```
Reset "stop thread" status
Reset the last received datagram identifier number
Create a UDP Socket and bind it to the given endpoint
Until stopping the thread is requested,
    wait for incoming UDP datagram,
    and send the datagram contents to the handle function.
```

The handle function, as said, will parse and validate the datagram. Then, if the datagram is valid, it will be stored in a variable *lastReceivedDatagram* by handle function. To there, the UDP handler, the "main" class of the module, will provide an interface to fetch the last received datagram by the receiver. When the *getLastReceived* function of the UDP handler is called, it also clears the value of the said variable for the last received datagram. So, when the function is called next time, an empty value will indicate that no new datagrams were received by the receiver. Thus, the storage for the last received datagram will act as a one-slot stack or queue for incoming datagrams. The logic in pseudo code of the *lastReceivedDatagram* function will be just the following:

```
Get the last received datagram from the UDP receiver.
If the datagram was not empty,
    reset the last received datagram of the UDP receiver to
'empty'.
Return the last received datagram.
```

As can be seen in Fig. 19, the part "Robot Framework test library" is now designed, except the test templates. Because they are strongly dependent on the implementation and are rather trivial, they will be only introduced in the testing chapter. Thus, this means that only designing the software for the SoC-FPGA is remaining for the design phase.

4.3 Design of the SoC-FPGA fault injector

In this subchapter, the remaining component of the error generator system, the SoC-FPGA fault injector is designed. Roughly, it can be considered as a UDP listener or server, that waits a request in incoming UDP datagrams, executes an action and if required, returns a result over UDP to the Robot Framework. Usually, "the action" is sending a command to the fault injection logic block over the AXI interface to inject a fault in the communication link.

As it is already described, the action can also be configuring some settings or fetching the version information. However, the main functionality of the fault injector can be divided into two parts; the UDP handler that deals with the incoming datagrams, and the AXI module or "AXI handler", which sends a command to the programmable logic – the fault injection logic – over the AXI interface. To avoid ambiguity between the UDP

handler of the fault injector and that of the RF test library, the UDP handler will be called “UDP server” from now on. These two modules as parts of the fault injector are depicted in a component diagram in Fig. 24.

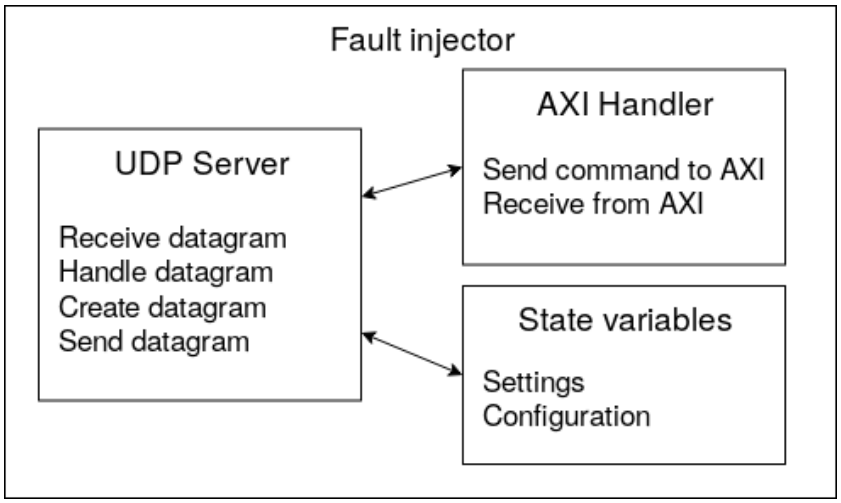

Figure 24. Component diagram of the SoC-FPGA fault injector. The UDP server is the core of the fault injector and the relationship between it and the AXI handler as well as the state variables which contain needed configuration data.

As shown in Fig. 24, the UDP server contains functionality to receive, handle, create and send a datagram. The fault injector, after the initialisation, will start to wait for incoming datagrams by receiving them. The received datagram is handled, and a proper action is executed by invoking a function in the AXI handler or writing and reading configuration data, depending on the received request. Also, depending on the request, a datagram can be created and sent back to Robot Framework test library. The design of these components are described more in the following subchapters.

4.3.1 Designing the UDP server

The UDP server receives and handles the incoming datagrams. Unlike the UDP handler in the Robot Framework test library, the receiving can be *blocking*, meaning that the program execution is blocked until a datagram is received. This is because actions in the fault injector are always triggered by sending a datagram from the test library.

After the SoC-FPGA fault injector is started, it will open a UDP socket and start listening for datagrams. The address and the port cannot be changed when the fault injector is

running, meaning that the address of the fault injector must be configured correctly in the RF test library. After a datagram is received, a correct action is executed, as stated in the following diagram in Fig. 25.

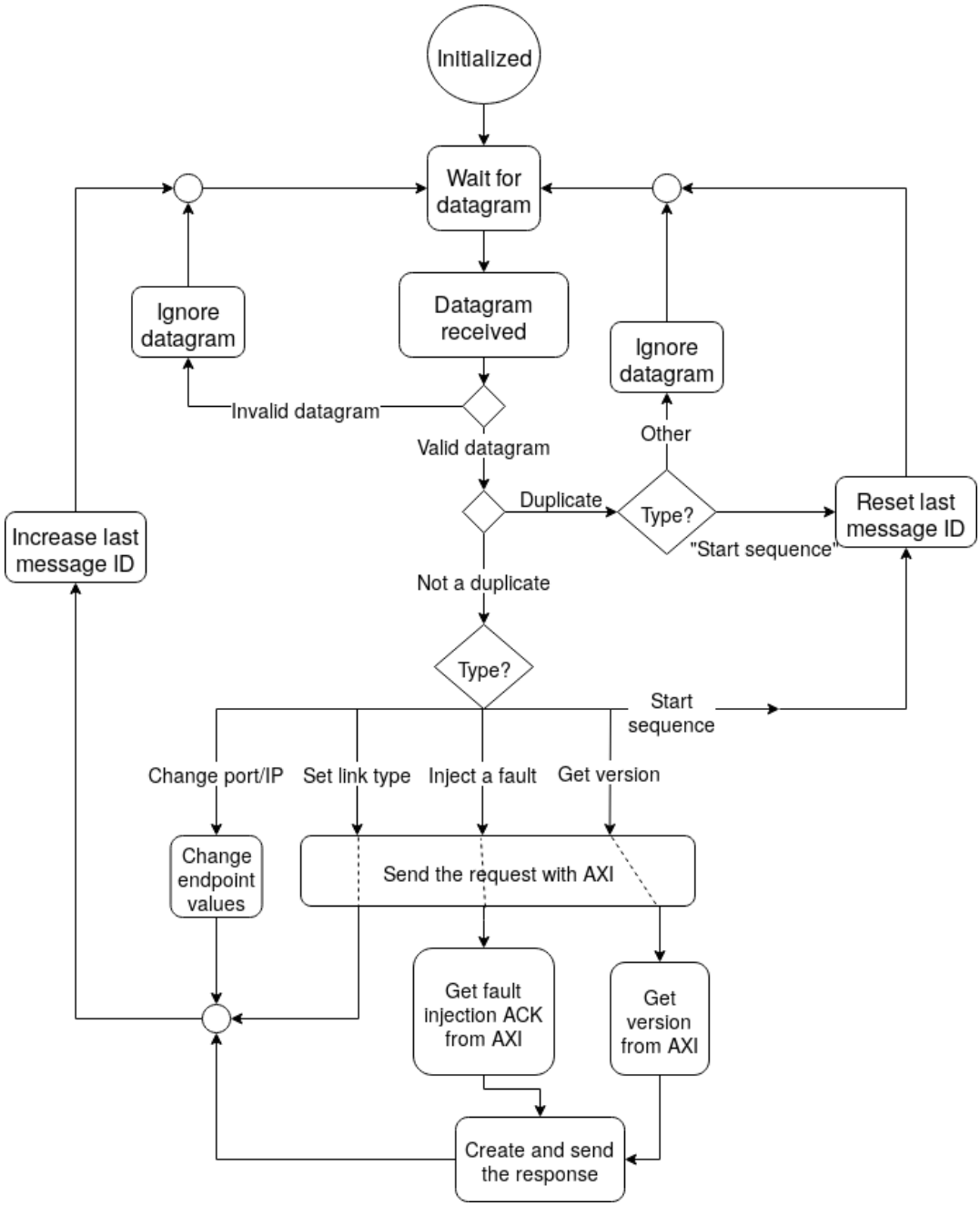

Figure 25. The flow diagram of the UDP server. In the second decision, after validating a datagram, a duplicate means that the message identifier of the datagram is the same as that of the previously received datagram. The type refers to the message types defined in Table 4.

From Figure 25, one can recognise the required functionality. This functionality can be separated in individual procedures, functions, which will be designed and implemented each on their own. Thus, the UDP handler consists of the following functions:

1. *Wait and receive datagram*, which receives a datagram and stores its contents in a shared variable.
2. *Validate datagram*, which checks that the datagram consists of earlier defined fields; a message identifier (a number), message type (a four-character string) and parameters. If the datagram is valid, each field is stored in a corresponding variable or a field in a structure or an object.
3. *Check duplicate*, which compares the message identifier of the datagram to the previous one. The datagram is ignored if the identifier is the same and the message type is not “start sequence”, in which case the identifier variable is reset so that next datagram won’t be treated as a duplicate.
4. *Execute action*, which will check the message type and execute the corresponding action. After the action has been executed, the variable containing the identifier of the last message is increased for duplicate validation.
5. *Change endpoint*, which is executed when the received datagram requests a change in the IP address or port. A new value is stored in a variable and next datagram will be sent to the newly configured endpoint. Thus, this function changes only the destination of the outgoing datagrams, since the IP address and port of the device are hard-coded and cannot be changed after booting.
6. *Set link type*, which sends a command with the later-described AXI handler to change the type of the communication link. This is needed because the fault injection logic block must have a configurable (numeric) communication link type.
7. *Inject a fault*, which injects the requested fault to the communication link by sending a command to the fault injection logic with the AXI handler. It will also wait for a response from the AXI for a short time and send the acknowledgement with UDP, if response was received. The “Get version info” will work in the same way; the request and the response to and from the AXI are just different, but the response is sent afterwards to Robot Framework over UDP.

8. *Create and send datagram*, which creates a datagram with the required elements and send it three times. It also handles increasing the message identifier for outgoing datagrams after each sent datagram, so that the receiving end will notice and drop the duplicates.

4.3.2 Designing the AXI handler

The AXI handler manages the communication between the fault injection logic on the FPGA and the processing system (the processor) of the SoC-FPGA over the AXI. It provides an interface for the programmable system so that it can call functions of the AXI handler to inject faults and set the communication link type and get the version information of the fault injection logic block.

Fault injection function sends a fault injection request over the AXI to the fault injection logic. It takes fault type, number of repeats and the target address as parameters. The number of repeats and target address are numeric parameters. Also, the fault type is a number in range 0–4 that enumerates the type of the fault: 0 = CRC fault, 1 = change address, 2 = change length, 3 = duplicate message and 4 = remove message. Firstly, the function validates that the input parameter values are in valid range, then converts them into an AXI suitable message and finally sends the message to the fault injection logic with AXI. After injecting the fault, the AXI handler reads data from the fault injection logic; the logic will return the type of the fault if the fault was successfully injected. Thus, a positive value is returned from the function if the fault injection acknowledgement was correct. Otherwise, if the acknowledgement does not match the type of the fault or cannot be read, or if the input parameters were invalid, an error code is returned. Hereby, the UDP server can then send the correct return value back to the Robot Framework.

Get version info works somewhat similarly as the previous function. However, it does not need any input parameters. So, when the function is called, the AXI handler makes an AXI read request on the correct address to get the version information. After reading, depending on the format in which the value is read, the AXI handler makes the necessary parsing and type conversion for the value, and returns it to the caller.

Set link type sets the type of the communication link that is connected to the fault injection logic. Because the fault injection logic is in the development phase while the thesis is being done, it is not yet sure what kind of values can be set for the communication link type. However, for extensibility purposes, this function is designed to take a numeric argument and write it over AXI to the fault injection logic. No return value is read afterwards.

5 IMPLEMENTATION OF THE ERROR GENERATOR SYSTEM

After designing the error generator system, the system was implemented. During the implementation phase, some flaws were found in the design, which required re-designing some parts of the system. These modifications are discussed first, and then the implementation details of the two modules (Robot Framework test library and SoC-FPGA fault injector) are given.

5.1 Redesign and general notes on the implementation

5.1.1 Redesigning the message identifier of the datagram

While making initial tests for the first implementation, erroneous behaviour was detected which was traced to be caused by the message identifier of datagrams. The initial design suggested that the datagram has a one-number identifier, which will roll back to 0 after the maximum value, 9, is reached.

There, however, were some problems when UDP datagrams were kept hanging in the network. This caused them to be handled as non-duplicate datagrams, when they finally after multiple seconds, or even minutes, reached the destination, the fault injector. This is illustrated in Figure 26.

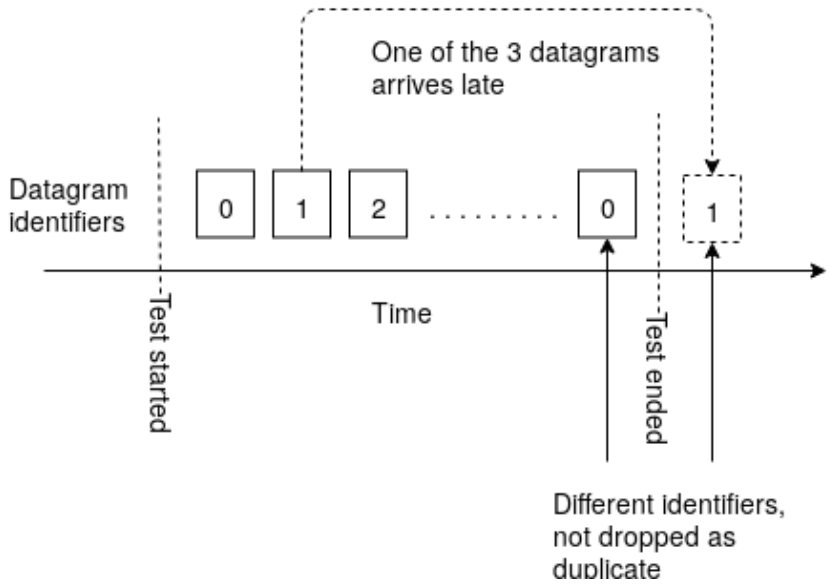

Figure 26. Datagram with identifier 1 arrives late. It is treated as not duplicate, because the identifier does not equal with the last received datagram.

Recalling that every datagram is sent three times, it is possible that one (or more) of the datagrams does not reach the destination or arrives very late. Finally, if the datagram arrives and it has a different identifier than the last received datagram has, the fault injector sees it as a correct (not a duplicate) datagram and executes the specified action.

This is harmful, if the datagram contained a fault injection request, causing the fault injector accidentally inject a fault at a wrong time, either during a test or even after the test. As already discussed in the theory part, in Subchapter 2.3, “in fault injection, faults are injected into a system which is then monitored to validate that the response matches the defined specifications during faulty conditions”. So, fault injection test results would be ambiguous if, for example, an error of wrong type was generated in the communication link, when some other response was expected.

Other possible consequence of an incorrect duplicate handling is if the datagram contained the “start sequence”. It resets the last received datagram id in the fault injector, so the next datagram is definitely handled and the duplicate check for the next datagram is ignored.

The problem was tackled by changing the message format. The identifier for the datagrams from Robot Framework test library to fault injector was made time-based to con-

tain a unique identifier which depends on the milliseconds elapsed, a *timestamp*. Thus, when receiving a datagram, checking the timestamp is used to detect late datagrams; a new datagram must have a larger timestamp value than the previously received datagram. Otherwise it is a duplicate if the identifier is the same, or a late datagram, if the identifier was smaller.

The new identifier is a ten-character hexadecimal number in string format. This increases the length of a datagram but is not considered a problem because there is relatively little data travelling in the Ethernet link.

The identifier is generated by calculating the number of milliseconds elapsed from an epoch, the noon on the 29th of August 2019. Thus, the maximum value being ffffffffff in hexadecimal is 2^{40} - 1 milliseconds from the epoch. This occurs on the 1st of January 2054, which should be enough. The accuracy of the timestamp is one millisecond which is enough. When executing a test sequence with Robot Framework, the consecutive calls to the fault injection keywords take at least tens of milliseconds because of the time to execute a single Python function. Thus, there will not be a collision of two different datagrams sent with the same timestamp. Unique timestamps are even ensured by adding a small, few-millisecond delay before the timestamp is generated. It does increase the execution time of a keyword but is relatively small, given that the execution of a keyword can take even a hundred milliseconds.

Another alternative for the timestamp is to use a dynamic epoch, so that it is reset e.g. at the midnight. This decreases notably the length of the timestamp. However, this approach requires synchronisation of timestamps. It can be done programmatically by configuring a clock (the current time) in the processing system of the SoC-FPGA so that the receiving end is aware when the reset of the epoch occurs. Alternatively, it can be done manually, by resetting (powering off and on) the SoC-FPGA fault injector so the last timestamp value is reset at start-up. The former needs that the time in the clocks of the RF test library and the SoC-FPGA are the same, so some sort of “clock synchronisation procedure” should be introduced which adds to the complexity of the system. The latter (resetting the device) seems a cumbersome solution and would not be suitable if the system is used in automatic testing. Thus, a fixed-epoch timestamp is used.

For the datagrams leaving from the SoC-FPGA fault injector to Robot Framework test library, the identifier is also a 10-character long hexadecimal string, but the value is incremented by one after each sent datagram. Furthermore, the “start sequence” datagram was abandoned, because if it arrives late, there was no chance to detect if it was correct. This makes the time-based identifier for the incoming datagrams to the fault injector the most suitable.

Because the last received identifier in Robot Framework test library is reset for each test execution, the datagrams coming from the fault injector don’t need to have such an identifier; the first received datagram in a test sequence is always accepted. This makes implementation easier. However, the identifier must always be increased between consecutive datagrams sent from the fault injector, too, otherwise the receiver in the RF test library end drops the message as a duplicate or late.

Now a datagram, that is sent from the Robot Framework test library to the fault injector at midnight on the 1st of September 2019 would have the format, when a CRC fault was requested

```
000cdfe600 fcrc 12345678 3 .
```

Here, 000cdfe600 is the identifier, which represents the 216,000,000 milliseconds elapsed from the epoch in hexadecimal format. Other parameters are the same as it was previously described in the design phase.

5.1.2 Redesigning the setup of the fault injection library

The initial design requested, that the RF test library should have keywords for configuring the IP address and port for both incoming and outgoing datagrams. These keywords were not implemented, because it is more convenient to set up the communication when the library object is initialised, and it is most likely unnecessary to change the configuration while the test is already running.

The IP and port for both directions are passed as parameters when configuring the library in a Robot Framework test sequence, for example:

```
*** Settings ***
Library  FaultInjectionLibrary    192.168.1.16    4398
  192.168.1.10 26000 ,
```

where the first two parameters, 192.168.1.16 and 4398 are the IP address and port for incoming datagrams, and the last two parameters, 192.168.1.10 and 26000 are the values for outgoing datagrams, i.e. the address of the fault injector device.

The constructor of the FaultInjectionLibrary class is called with those parameters. Eventually, it calls an initialisation function with the same parameters to set up the endpoint for the UDP sender object, and start listening with the UDP receiver object at the given endpoint. It also sends the address to the fault injector so the fault injector will transmit its datagrams to the correct address. Also, as seen in Algorithm 3, an additional datagram “ping” is sent to the fault injector after configuration; the fault injector should respond with an “_ack” datagram. This process ensures that the communication is configured correctly and the test itself can be run.

```
self.handler.setDestination(str(ipOut), int(portOut))
self.handler.setRecvEndpoint(str(ipIn), int(portIn))
self.handler.initialize()

# Send the IP of this machine to the fault injector
self.handler.send(UDPDatagram.UDPDatagram('sipa',
[str(ipIn)]))
self.handler.send(UDPDatagram.UDPDatagram('spor',
[str(portIn)]))

# Erase any last received datagram first
dgram = self.handler.getLastReceived()
self.handler.send(UDPDatagram.UDPDatagram('ping'))
dgram = self._getWithTimeout() # wait max 5 secs.

if dgram is None or dgram.msgType != '_ack':
  self.close_library()
  raise Exception('...')
```

Algorithm 3. The library initialisation function. Handler initialisation function will start to listen for incoming datagrams. With the helper function ”_getWithTimeout”, the call waits 5 seconds for receiving a datagram.

5.1.3 The incomplete implementation of the AXI handler

The AXI handler is designed to be responsible of making read and write calls over the AXI interface to the fault injection logic IP block. The IP block in question, as already mentioned in this thesis, is implemented and designed independently at the company. However, the IP block is not ready before the thesis is finished, so the fault injector system and the AXI handler have to be implemented without it.

The AXI handler contains the functions

```
int send_fault_injection(int fault_type, char*
address_and_repeats);
int get_fault_injection_ack(int fault_type);
void set_link_type(int link_type);
void get_fault_injection_version(char* buffer);,
```

where “send_fault_injection” and “set_link_type” functions should execute an AXI write operation and the other two a read operation. However, because of missing specifications – i.a. on the interface of the injection logic block and the AXI version that is used – the AXI call is not made, but a placeholder for the AXI call is written as a comment so that the code can be extended when the required IP block is ready. For example

```
void set_link_type(int link_type) {
  /* TODO: AXI write to set the link type */
}
```

is an AXI handler function to set the communication link type. It contains a comment in the code where the actual AXI write function call should be inserted. Furthermore, because the block is not ready, it cannot be configured for the programmable logic; a task that would have been included in the implementation part of this thesis. AXI handler implementation is discussed more in detail in Subchapter 5.3.

5.2 Implementation of the Robot Framework test library

The Robot Framework test library is implemented with Python programming language's version 3.6.8 for Robot Frameworks version 3.1.1. The test library contains five files: UDPDatagram.py, UDPSender.py, UDPReceiver.py, UDPHandler.py and FaultInjectionLibrary.py. Each file represents a single Python module which means that the code in one file can be used, i.e. imported, in another module. This allows modularisation of the logic, so that code that is relative for a specific task is located in a single module, a file, and not scattered in pieces between the files. This helps maintaining and further development. The hierarchy between the Python files is depicted in Figure 27.

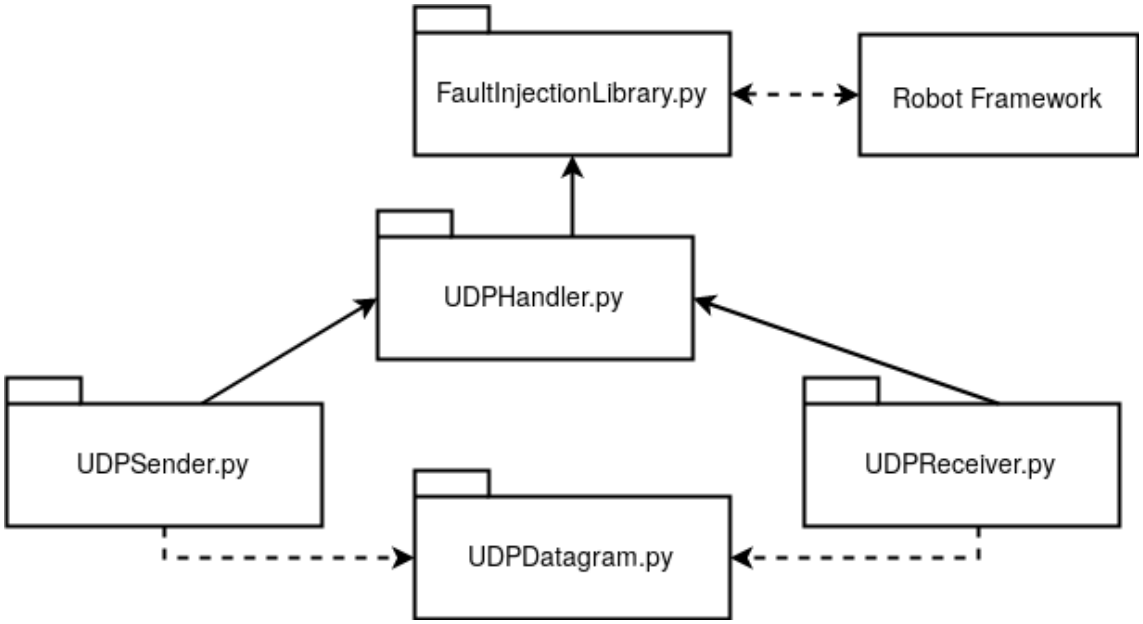

Figure 27. The modules (files) in the Robot Framework test library and the relationship between them. Robot Framework accesses the codebase with the test library module, FaultInjectionLibrary.py, by loading it at the beginning of a test.

As it was designed, UDPDatagram.py contains a class representation of a single datagram. The interface (declarations of the functions and the class) is given in Algorithm 4.

```
class UDPDatagram:

    # Initializes id, messageType and parameters
    def __init__(self, messageType = '', parameters = []):
        # -- Lines removed --

    # Validates that the datagram is correct
    def validate(self):
        # -- Lines removed --

    # This converts this UDP datagram object into a string
    def getString(self):
        # -- Lines removed --

# Creates a datagram object from the contents (a string)
def creteDatagram(contentString):
    # -- Lines removed --
```

Algorithm 4. Function declarations of the UDPDatagram.py module.

As shown in Algorithm 4, three additional functions were added to this module. After a datagram is created and its identifier, message type and parameters are changed, the module that uses the datagram should call “validate” function on the object before proceeding further. This function verifiers that the type is one of the allowed types and the identifier as well as the parameters are in allowed range. If any criteria is not fulfilled, an exception is raised so the caller of the validate function is informed.

The second of the additional functions in this module are “getString” which converts a datagram object to a string representation so it is serialised and can be encoded into bytes before sending. The last, the third additional function is “createDatagram” which takes a string representation of a datagram and converts it to a UDPDatagram object. Before returning the object instance, it also calls the previously described validate function.

The file UDPSender.py is a module that contains a UDPSender class that is responsible for sending datagrams. Its declaration is shown in Algorithm 5.

```
class UDPSender:

    # The datagram identifier is created
    # with the milliseconds elapsed from the EPOCH
    # Epoch on 29th of August 2019 12:00 (24h clock)
    EPOCH = datetime(2019, 8, 29, 12, 0, 0)

    # Initialise values and create a sending socket
    def __init__(self):
        # -- Lines removed --

    # Close the sending socket
    def close(self):
        # -- Lines removed --

    # Get the identifier (as integer) for the datagram
    def getIdentifier(self):
        # -- Lines removed --

    # Send a datagram
    def send(self, datagram):
        # -- Lines removed –
```

Algorithm 5. Function declaration of the UDPSender.py module.

It is implemented as designed, except the identifier has to match the new specification. Before sending a datagram, the “getIdentifier” function is called, which calculates the number of milliseconds elapsed from the epoch. The epoch is defined as a class variable called “EPOCH”, as shown in the previous code in Algorithm 5.

When sending, the UDPSender creates a UDPDatagram object and gets its string representation with “getString” of that class. That function does the conversion for the timestamp identifier, which is in integer format into a 10-digit hexadecimal format.

The sending socket is created in the constructor of the class (in Python it is called “__init__”). To free any resources in the system, the UDPSender class has an additional “close” function which closes the UDP socket. It should always be called at the end of the program.

UDPReceiver.py contains the UDPReceiver class that listens for incoming datagrams and handles them. The declaration of this class is given in Algorithm 6.

```
class UDPReceiver:

    def __init__(self):
        # -- Lines removed --

    # Start listening the datagrams in a separate thread.
    # This should be called in UDP Handler's initialize
function.
    def startListening(self, address = None, port = None):
        # -- Lines removed --

    # Listen indefinitely for incoming datagrams
    def listenEndlessly(self):
        # -- Lines removed --

    # Handle the datagram: validate and ignore duplicates
    def handle(self, data, addr):
        # -- Lines removed --

    # This closes the receiver when test quits or the ad-
dress is changed
    def close(self, socketEndpoint = None):
        # -- Lines removed –
```

Algorithm 6. Declaration of the functions in UDPReceiver.py module.

In the design phase, it was planned that there should be a boolean variable that indicates when the socket should be closed. This is designed for the case when the address for incoming datagrams would be changed, so the previous socket and thread must be closed before a new one can be opened.

Instead, the closing and re-opening process is done by sending a special datagram. This is because the function “recvfrom” in Python’s socket library blocks the execution of the code until a datagram is received. So, when a new thread should be started, the receiver creates a sending UDP socket which sends a “DATAGRAM_RECV_STOP” datagram to the receiving socket as shown in Algorithm 7.

```
if self.sock is not None:
    # Socket is closed by sending a stop command to it
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    for _ in range(3): # Send 3 times
        s.sendto('DATAGRAM_RECV_STOP'.encode(), ep)
    s.close()
```

Algorithm 7. Inside the close function, a special stop datagram is sent with a temporary socket ”s” to the receiving socket (its address is defined with ”ep”).

The receiving socket listens for this message and closes itself and the thread stops if such message was received.

Furthermore, because of the redesigned datagram identifier specification, any received datagram is ignored and dropped not only if the identifier is the same than the previous one but also if the value is smaller than the previous. Also, similarly with the UDPSender class, the receiver contains a simple “close” function that executes the previously described closing process, so the socket is freed and the listening thread finishes when the program quits.

UDPHandler.py file contains the UDPHandler class. As it was designed, it is only an interface for the Robot Framework test library. It has one additional function called “cleanUp”. It closes the sockets of the sender and receiver with the respective “close” functions of those two classes.

FaultInjectionLibrary.py is the file that contains the test library class which will be imported by Robot Framework when a test is executed. The declaration of all the functions in this class is shown in Algorithm 8.

```
class FaultInjectionLibrary(object):

    # Only one instance is created for the test execution
    ROBOT_LIBRARY_SCOPE = 'GLOBAL'

    def __init__(self, ipIn = RECV_IP, portIn = RECV_PORT,
ipOut = ZYNQ_IP, portOut = ZYNQ_PORT):

    def _initializeLibrary(self, ipIn, portIn, ipOut,
portOut):

    # General method to send a fault injection command with
UDP and receive an acknowledgement
    def _injectFault(self, faultType, address, count):

    # Get the received datagram but with a 5 sec timeout
    def _getWithTimeout(self):

    # The keywords that are available in the library:
    def close_library(self):
    def inject_CRC_fault(self, address, count):
    def inject_address_change_fault(self, address, count):
    def inject_length_change_fault(self, address, count):
    def inject_duplicate_message_fault(self, address,
count):
    def inject_drop_message_fault(self, address, count):
    def get_fault_injection_version(self):
    def set_communication_link_type(self, linkType):
```

Algorithm 8. Function declarations in the FaultInjectionLibrary class. For simplicity, function bodies are removed without explicitly telling so.

FaultInjectionLibrary is initialised as it was discussed in Subchapter 5.1.2 and shown in Algorithm 3. The library has also a “close_library” function, which eventually calls the cleanup function from the UDPHandler. Thus, all the reserved resources (sockets and threads) are finished and closed when the function is called. Hence, this keyword must be called at the tear down of a test.

All the functions in the class are available as Robot Framework keywords in the test sequence, unless they start with an underscore (_). Because all the fault injection keywords have the same logic – except the fault type – the common logic is included in “_injectFault” function, which takes the fault type, the address and the number of repeats as parameter. As designed, it injects the selected fault by sending a fault injection

request with UDP to the fault injector and waits for the acknowledgement. Then, for example the definition of the function “inject_crc_fault” can be simplified into

```
def inject_CRC_fault(self, address, count):
    self._injectFault('fcrc', address, count).
```

Also, there is additional helper function called “_getWithTimeout”. This is used e.g. in the previous fault injection function to get the acknowledgement of the fault injection. This function gets the last received datagram from the UDPReceiver with a timeout; if there wasn’t any datagram received in a timeout of 5 seconds, an empty value “None” is returned to indicate that the timeout was reached.

Furthermore, the FaultInjectionLibrary class contains a class member definition

```
ROBOT_LIBRARY_SCOPE = 'GLOBAL',
```

which is a built-in feature of Robot Framework. When the value of this variable is set to “GLOBAL”, Robot Framework knows that it should only create one single instance of this test library during the whole test execution. If this was not defined, or if it had a different value, Robot Framework would create a new instance of the test library for each test. This would initialise the test library and create a new listening socket for each test inside the test suite, which is unwanted behaviour.

5.3 Implementation of the SoC-FPGA fault injector

The fault injector is implemented by programming the processing system with C language. The project template for the Xilinx SDK (*software development kit*) and FPGA configuration tool is generated automatically by a script. It includes the needed configuration to first program the programmable logic (FPGA) and it generates the necessary files to program the processing system. In addition to the automatically generated files, it has three custom made files; main.c, udp_server.c and axi_manager.c. The latter two have also header files called udp_server.h and axi_manager.h.

The file main.c does the initialisation of the UDP PCBs (*protocol control blocks*) and the networking for incoming datagrams and calls for a function in udp_server.c to start listening for incoming datagrams. The initialisation logic code is modified from an example UDP code provided by Xilinx.

The file udp_server.c contains the designed UDP server. The header file contains the function declarations:

```
void set_fwd_addr(char* ip_addr, int port);
void receive_datagram(void *arg, struct udp_pcb *pcb, struct
pbuf *p, struct ip_addr *addr, u16_t port);
int validate_datagram(const char* datagram, long long* id,
char* msg_type_buffer, char* parameters_buffer);
int check_duplicate(long long* id, const char*
message_type);
void execute_action(const char* message_type, char*
parameters);
void send_data(char data[]);
```

It sets up the PCB for outgoing UDP datagrams with “set_fwd_addr” and executes a function called “receive_datagram” when a new datagram is received. This function then calls validation and duplicate detection functions to verify that the received datagram is valid before calling “execute_action” function. This is the function that finally executes an action, depending on the message type. For example, it can send a fault injection request and getting the acknowledgement from the AXI handler or change the forwarding address again.lar

If the action required sending a response datagram back to Robot Framework, it is sent by calling “send_data” function. This function adds a proper identifier to the datagram and sends it three times to the configured upstream endpoint. The contents for the datagram are passed as a plain string because unlike the datagrams sent from Robot Framework, they are very simple by structure, and thus no special structure handling is needed.

AXI handler is implemented in axi_manager.c. As stated, it cannot contain much logic, since the specification of the AXI interface to the fault injection logic is not provided.

For testing purposes, “get_fault_injection_version” function returns a dummy fault injection version “0.1” and “get_fault_injection_ack” returns 1 which is code for a successful operation.

Sending fault injection is done in “send_fault_injection” function. For it, the target address of the fault and the repeat count are passed in a single string, where the two are separated by a space character. Splitting and converting the two into separate variables is done inside the function. If the repeat count was not numeric or if one of the two were missing, the function returns ‘0’ which incidates an unsuccessful operation. Otherwise, ‘1’ is returned. Other functions do not do validation of input data of any kind.

6 TESTING THE FAULT INJECTOR SYSTEM

In this chapter, the fault injector is tested. In the initial plan, it was supposed that the fault injector has the fault injection logic block available and it is connected to the communication link. The injected faults would have been seen in the communication link by reading statistics and other data and the fault detection and correction for the injected faults could have been reflected with the theory given in Chapter 2.

However, because the fault injection logic block is not available yet, the nature of the testing phase changes. It concentrates on testing the UDP/IP communication and the Robot Framework test library.

6.1 Designing the tests

All the tests are done by executing Robot Framework test files that use the implemented RF test library to send and receive datagrams. All the test files are based on the following test file template

```
*** Settings ***
Library  FaultInjectionLibrary  [ad1] [p1] [ad2] [p2]
Suite Teardown Close Library

*** Test Cases ***
First test
    # Test keywords here
Nth test
    # More keyword calls here,
```

which shows how to initialise the fault injection library, and how to close it after the test suite (the collection of multiple tests) is completed, whether it was successful or not. As already discussed, the Library keyword initialises a library (FaultInjectionLibrary) with given parameters. The parameters for our library are the endpoint values (address and port) for incoming datagrams and for outgoing datagrams, in this order.

When the test is designed, the test file can be saved and run by Robot Framework with

```
python -m robot test_file.robot,
```

assuming that Python 3 is installed with Robot Framework and available with “python” command, and the test file is called “test_file.robot” in the current directory.

After the test is run, Robot Framework generates three files. Summary of all the test executed in the test suite is in “result.html” HTML file. The details of the tests are in a HTML log file “log.html” which contains all the steps of the tests and the same data in a more machine readable XML format in a file “output.xml”. For the tests here, log.html file is read with a web browser to show the result and details of a test. If there were any errors, this file also gives a description of errors. An example of how the log.html looks like is shown in Figure 28.

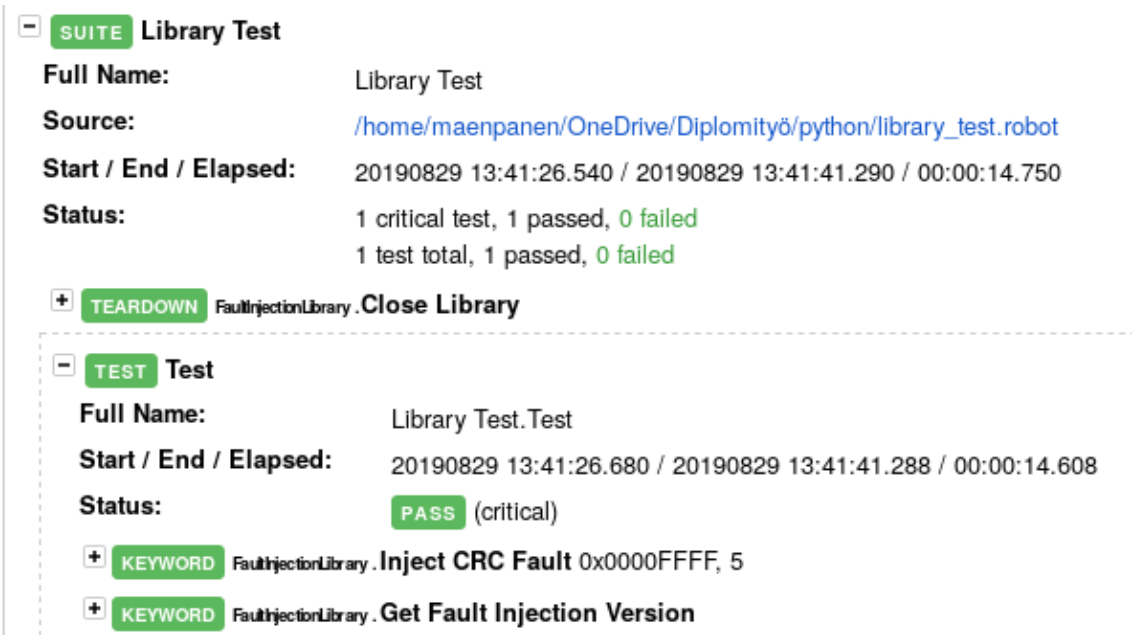

Figure 28. A partial screenshot from the log.html HTML file. It shows information about a completed test suite as well as keyword-level details on the test execution.

The fault injector board also gives output of received and sent datagrams, as well as other data about execution. This data is also read but it has to be done manually for each test to validate that the fault injector functions properly. Thus, the tests should cover all the cases but a single test should be short enough so that there is not too much data to be read manually, so the output from the fault injector can easily be validated.

Thus, the following test cases should be created and run with Robot Framework by using the given test file template:

1. **Fault injection tests**, which contain all the fault injection types. The fault injections should be successful. Different parameters (target address and repeat count) are used.
2. **Failed fault injections**, which try unsuccessfully to inject a fault with a fault injector by passing invalid parameters. The built-in keyword “Run keyword and expect error” of Robot Framework can be utilised here. The errors should caught by data validation in the keywords of the RF test library.
3. **Timeout test**, which tests that a fault injection keyword should fail if receiving acknowledgement lasts too long. Because the fault injector returns always an acknowledgement, its code has to be tweaked for this test so that in some cases the fault injector does not return an ACK datagram to the RF test library.
4. **Negative acknowledgement test**, which tests that the fault injection fails if a negative acknowledgement is received from the fault injector. Like in timeout testing, additional code for testing purposes must be added so that negative ACK can be returned from the fault injector.
5. **Test other functions**, which are setting the communication link type and reading the fault injector version. Testing with invalid values is included here, too.
6. **Multiple tests**, which mean that a test suite (a file) has multiple tests. This tests validates, that only one instance of the fault injection library is generated for all the tests.
7. **Library initialisation**, which initialises the library with invalid parameters. The initialisation function test for successful communication with a “ping” datagram to the fault injector which should fail and a test should not start.
8. **All functions test**, which contains a portion of each of the previous tests. Here, it is validated that there are no unexpected cross-effects between the functions.

All the eight tests should be successfully executed. This is validated by both reading the log file from Robot Framework and the output from the fault injector. If a test fails, the cause will be found out and needed actions are executed, either fixing a small programming error or making a larger change in design, depending on the error.

6.2 Preparing the testing

The SoC-FPGA fault injection board is powered up and connected to the PC. It has three connections to the PC as shown in Figure 29. The JTAG (joint test action group) port is connected with a USB adapter to the computer which allows programming the board. An Ethernet cable is connected to the corresponding ports in both PC and the board and it is used for the UDP/IP communication between the two. Furthermore, a second cable is connected to the PC with a USB adapter and it uses a serial protocol to send e.g. the output of the print statements from the board to the PC. This output can be read by using Minicom serial communications program on the PC, as depicted in Figure 30.

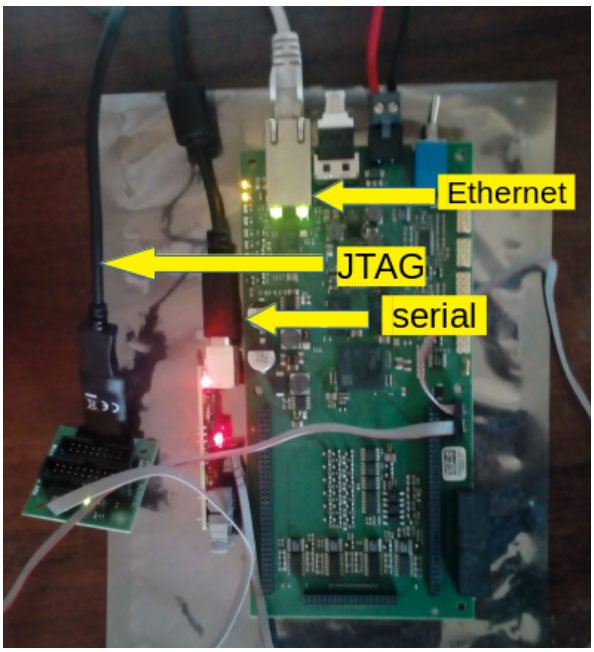

Figure 29. The fault injector SoC-FPGA board with a JTAG USB cable, a serial cable and an Ethernet cable connected to it.

```
Minicom 2.7.1

VALITSIMET: I18n
Käännetty  Aug 13 2017, 15:25:34.
Portti /dev/ttyUSB0, 12:51:52

Komentonäppäinten ohjeet: CTRL-A Z

Starting...
The MAC address of this device is 50:51:52:53:54:55
Ip address of this device is: 192.168.1.10
Start Micrel PHY autonegotiation
autonegotiation complete
link speed for phy address 0: 1000
Changing IP Address...
UDP block created
changing the port...
Forwarding to 192.168.1.16 at port 4398
UDP server started.
```

Figure 30. A screenshot from Ubuntu terminal that is running Minicom. At the top, there is configuration information and help in Finnish followed by the actual output from the fault injector.

After setting up the fault injector board, the FPGA on it is programmed by Xilinx SDK. After programming the FPGA, the software for the fault injector is programmed on the processing system of the SoC-FPGA and run. Finally, a static IP address is assigned on the PC so that the communication over Ethernet to the fault injector is possible. The fault injector is ready for testing.

Because the fault injection logic block is not yet available, testing timeout and negative acknowledgements are not possible as such. To be able to test them, the source code in fault injector is changed a little. A piece of code shown in Algorithm 9 is injected in a proper place for testing purposes in “execute_action” function in the fault injector. Thus, if the address for a fault injection is a string “timeout” and repeat count is 1, no acknowledgement is sent. Additionally, if the address is “negative_ack” and the count is 1, a negative acknowledgement is sent. The keyword call in Robot Framework test should fail in this case.

```
// Simulates that no ACK is sent
if (!strcmp(parameters, "timeout 1")) {
    return;
}

// Simulates a negative acknowledgement from the AXI
if (!strcmp(parameters, "negative_ack 1")) {
     send_data("_nak");
     return;
 }
```

Algorithm 9. Fault injection process is stopped with return before sending an ACK if the parameter string matches with ”timeout 1”. Furthermore, it is stopped and a negative acknowledgement (_nak) is sent if the parameter string is ”negative_ack 1”.

The test files are then created. The contents of the files are included in the Subchapter 6.3 together with the test log and fault injector output.

6.3 Running the tests

The test are run by launching a test from the command line. After the test has run, the fault injector output is read in Minicom window and validated to match the test sequence. Also, the Robot Framework log file is used to validate a successfully executed test.

In Algorithm 10, there is an example of one of the tests, the first test called “fault injection tests”. Setup and tear down of the library are omitted here at the beginning of the file.

```
Fault Injection Tests
  Inject CRC Fault                    abcdffff   5
  Inject Address Change Fault         12345678   2
  Inject Length Change Fault          abcdef01   100
  Inject Duplicate Message Fault      abcdef01   1
  Inject Drop Message Fault           50000aaa   1000
```

Algorithm 10. Example of a fault inject test sequence in the tests.

The output is collected from the fault injector. Example of the output from the execution of the previously given “fault injection tests” test sequence is

```
<-- "0007034d02 fcrc abcdffff 5"
Injecting fault
"0000000001 _ack" --> 192.168.1.16:4398
<-- "0007034d69 fadr 12345678 2"
Injecting fault
"0000000002 _ack" --> 192.168.1.16:4398
<-- "0007034dd0 flen abcdef01 100"
Injecting fault
"0000000003 _ack" --> 192.168.1.16:4398
<-- "0007034e38 fdup abcdef01 1"
Injecting fault
"0000000004 _ack" --> 192.168.1.16:4398
<-- "0007034e9f fdrp 50000aaa 1000"
Injecting fault
"0000000005 _ack" --> 192.168.1.16:4398,
```

where the arrows represent incoming (<--) and outgoing (-->) datagrams to and from the fault injector. As can be seen from the output, all the five faults are injected and an acknowledgement is returned. Also, the datagram format is correct. The Robot Frameworks log HTML file is shown in Figure 31.

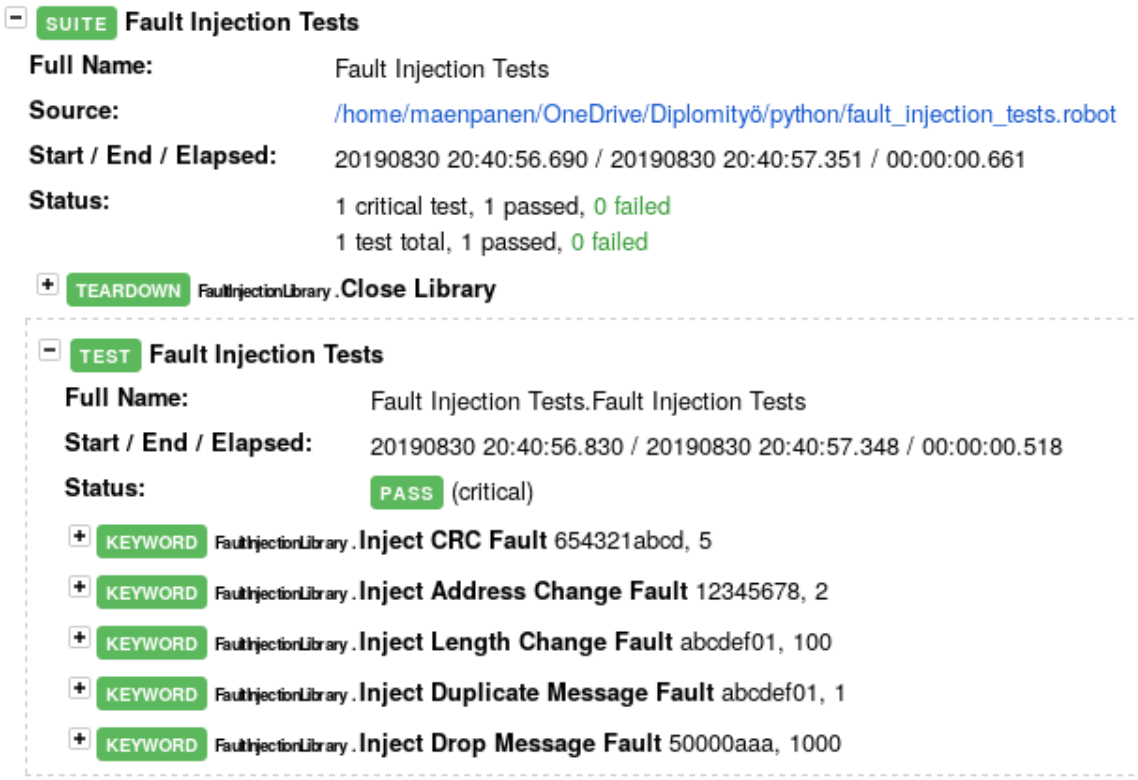

Figure 31. The screenshot of the log file of the ”fault injection tests” test.

As can be seen in Table 7, library initialisation test failed expectedly. It is because the execution was stopped at the first step, when an CRC fault was tried to be injected. The error message was "*No keyword with name 'Inject CRC Fault' found*". This is because the library initialisation threw an error when sending the "ping" datagram to the fault injector did not result in an acknowledgement response because of incorrect endpoint values typed in the initialisation of the library. Thus, the library was not initialised and the keywords were not available in the test sequence.

Table 7. The tests and their results.

Test name	Result
Fault injection tests	success
Failed fault injections	success
Timeout test	success
Negative acknowledgement test	success
Other functions	success
Test with multiple tests	success
Library initialisation	failed (expectedly)
Test all functions	success

6.4 Summary and analysis of the test results

As can be seen in Table 7, all the tests were executed successfully. In addition, when validating the fault injector output files, it was seen that there were no errors in the communication. From these, it can be concluded that the error generator system works expectedly.

Unfortunately, as discussed earlier, the fault injection logic block was not available during the tests. This means that only the UDP/IP communication and the Robot Framework test library were tested. Also, as it was noted in the test of "Library initialisation", the test failed and stopped at the first test step and not in the initialisation process. Stopping the test could be done earlier, before the first step, if Robot Framework supports it.

7 CONCLUSIONS AND FUTURE

In this thesis, a fault-injecting device and a test library for an error generator system is created. They are used to inject faults into a serial communication link to test the error handling of the link. The created device will be used as a part of Danfoss test automation system. The objective consists of designing and implementing a SoC-FPGA fault injector that can communicate with a Robot Framework test library over Ethernet and with the actual fault injection logic on the FPGA, too. Furthermore, the objective includes designing and implementing the Robot Framework test library which is used to produce targeted and automatic fault injection tests for a communication link.

From the remarks made during the thesis as well as analysing the test results, the following conclusions are made:

- Working principles of error detection and handling in a serial communication links were studied.
- A system with the given specifications was designed and implemented.
- The system was tested without the missing fault injector block from the supplier and it was found out that the system worked expectedly.
- It will be possible to create and run customisable fault injection tests on a communication link after the missing component is added and configured to the system.

As stated in the thesis, the system cannot be tested as a whole because the fault injection block is not yet implemented. Thus the objective of this thesis is reduced. Some of the other remarks are not discussed in full detail in the work, but postponed for later development and further research. These include

- adding the fault injection logic block into the system,
- testing the system on a real communication link,
- a more detailed study on the advantages and disadvantages of using a UDP-based communication instead of TCP and
- implementing the UDP protocol with binary messages instead of plain-text and studying the advantages of that improvement.

8 SUMMARY

This thesis is made for Danfoss Drives that manufactures frequency converters. In frequency converters, avoiding errors in transferred data is critical, which is why an error generator system is created in this thesis to test the reliability of the communication in for example a frequency converter.

The fault injection logic block is aimed to be included in the SoC-FPGA fault injector that was created in this thesis. The logic block in question is a circuit that actually does injecting the faults into a communication link. However, the block is not designed within this thesis but by the Danfoss. For this, fault detection and correction techniques in serial communication protocols are studied. This information is useful when eventually running the fault injection tests when the fault injector is connected into a serial communication link.

However, the thesis has a limited timespan where it should be carried out. The logic block in question could not yet be delivered for the thesis within that time so it had to be left out. This means that most of the theory chapter has information that could not be utilised in the reflection and analysis of the test results. However, after studying serial communication protocols, to help the next phases, the design and implementation, the reader was acquainted with SoC-FPGAs, Robot Framework and UDP.

The design and the implementation phases are carried out by separating the error generator system into three main parts; the UDP communication between the SoC-FPGA fault injector and the PC that runs Robot Framework, the RF test library that provides functions to execute fault injection tests and the logic for the programmable system of the fault injector.

After this phase, the functionality is tested by designing and running tests for each functionality. Unlike it was thought at the start of the thesis, the system could not be tested with a real communication link where faults would have been injected. Thus, testing was done only for the ready components. The test results indicated that the implemented components work as expected and they fulfil the requirements.

From the design and implementation part, the main conclusion is that the work on the UDP communication took most of the time. Missing datagrams and datagrams arriving in a scrambled order caused trouble when designing the communication. Despite studying the UDP protocol in the theory phase, the flow control of the communication was not designed robust enough. Because of this, some iterative design had to be done when flaws in the design were found only when already entering the implementation phase.

In the future, the UDP communication could be replaced by a TCP communication. UDP is by default faster and lighter than TCP. Thus, it could be compared if any advantage can be achieved when using one over the other. This study can be interesting, because, to increase robustness, additional flow control with identifiers and sending multiple packets had to be implemented on top of the UDP which increased its bandwidth usage – however, TCP does this automatically.

Most importantly, the fault injection logic block should be connected into the fault injector device to actually inject faults. This allows testing the whole system and can reveal problems in the implementation or in the design.

In the design phase, it was assumed that the block will be ready and connected to the system, so the design was made a complete system in mind. Only in the implementation the respective parts had to be left out. In the best case, the block has just to be configured in the system and a few lines of code to be inserted in the code base of the fault injector to get the complete system working. Thus, finalising the error generator system, testing the complete system and analysing the test results by reflecting them to the theory part of this thesis is the most likely relevant way to expand the work and do further research.

REFERENCES

Altera (2014). *What is an SoC FPGA – Architecture Brief*. [online document]. [Available at: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ab/ab1_soc_fpga.pdf]. [cited on 24 July 2019]

Axelson, Jan (2007). *Serial Port Complete: COM Ports, USB Virtual COM Ports, and Ports for Embedded Systems* (2nd ed.). Madison, Michigan: Lakeview Research. 400 p.

Axelson, Jan (2015). *USB Complete: The Developer's Guide* (5th ed.). Madison, Michigan: Lakeview Research. 545 p.

Bosch (1991). *CAN Specification*. Version 2.0. 72 p.

Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC & Philips (2000). *Universal Serial Bus Specification*. Revision 2.0. 622 p.

Danfoss (2000). *Tietämisen arvoista asiaa taajuudenmuuttajista*. 172 p.

Danfoss (2016). *Vacon NX IO Boards User Manual* [web document]. [Available at: http://files.danfoss.com/download/Drives/Vacon-NX-IO-boards-User-Manual-DPD00884B-UK.pdf]. [cited on 1 July 2019].

Davis, R. I., A. Burns, R. J. Bril, J. J. Lukkien (2007). Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised. *Real-Time Systems* 35: 3. 239–272.

Dell, J. A. (2015). *Digital Interface Design and Application*. Wiley. 206 p.

Fisher, Marvin J. (1991). *Power Electronics*. Boston: PWS-Kent. 491 p.

Hamming, R. W. (1950). Error detecting and error correcting codes. *The Bell System Technical Journal* 29: 2. 147–160.

Hindmarsh, John. & A. Renfrew (1996). *Electrical Machines and Drive Systems*. 3rd edition. Oxford: Butterforth-Heinemann. 368 p.

Hioki, Warren (2001). *Telecommunications* (4th ed.). Upper Saddle River, New Jersey: Prentice Hall. 644 p.

IEEE (1996). Editor: Bimal K. Bose. *Power Electronics and Variable Frequency Drives: Technology and Applications*. 640 p.

ISO (2002). *Open Systems Interconnection – Network Service Definition.* ISO/IEC Standard 8348:2002. 68 p.

Koren, Israel & C. M. Krishna (2007). *Fault-Tolerant Systems*. Burlington: Morgan Kaufmann. 399 p.

Krishnan, R. (2001). *Electric Motor Drives: Modelling, Analysis and Control*. New Jersey: Prentice Hall. 626 p.

Malarić, Krešimir 2010. *EMI Protection for Communication Systems*. Boston: Artech House. 273 p.

Motiva (2002). *Energiansäästäminen Suomessa – energiatehokkuudella kilpailukykyä.* [online] [Available at: http://www.motiva.fi/files/8001/ Energiansaastaminen_Suomessa_Energiatehokkuudella_kilpailukykya.pdf] [cited on 19 July 2019].

Murthy, C. S. V. (2009). *Data Communication and Networking*. Global Media. 506 p.

Peterson, W. W., D. T. Brown (1961). Cyclic Codes for Error Detection. *Proceedings of the IRE* 49: 1, 228–235.

RFC 768 (1980). *User Datagram Protocol*. Protocol Specification.

RFC 791 (1981). *Internet Protocol*. Protocol Specification.

Robot Framework (2019). Introduction and Examples. [web page]. [Available at: https://www.robotframework.org/] [cited on 1 July 2019].

Shannon, Lesley (2012). Embedded Computing Systems on FPGAs. In: *Embedded Systems: Hardware, Design and Implementation,* p. 127–138. Ed. Winieski, Krzystof. Hoboken, New Jersey: John Wiley & Sons.

Shinde, S.S. (2000). *Computer Network*. Delhi: New Age International. 419 p.

Wicker, S. B. (1995). *Error Control Systems for Digital Communication and Storage*. Prentice Hall. 512 p.

Widmer, A. X., P. A. Franaszek (1983). A DC-Balanced, Partitioned-Block, 8B / 10 B Transmission Code. *IBM Journal of Research and Development* 27: 5. 440–451.

Wikibooks (2019). *Communication Networks / TCP and UDP Protocols*. [web page]. [cited on 1 July 2019]. [Available at: https://en.wikibooks.org/wiki/ Communication_Networks/TCP_and_UDP_Protocols].

Xilinx (2012). UG761 – *AXI Reference Guide*. Version 14.3. [online] [available at: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/ latest/ug1037-vivado-axi-reference-guide.pdf] [cited on 1 July 2019].