DIPLOMARBEIT

Xerxes Error Behavior

ausgeführt unter Anleitung von

A.o. Univ-Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Institut für technische Informatik, VLSI-Design 182-2

und

Dipl.-Ing. Oliver Maischberger
Dependable Computer Systems, GmbH.

eingereicht an der Technischen Universität Wien,
Technisch-Naturwissenschaftliche Fakultät

durch

Bernhard Rieder
1200 Wien

Wien, im Mai 2003

Xerxes Error Behavior

Zusammenfassung

In der heutigen Automobilindustrie findet zur Zeit ein Wechsel der Basistechnologie von mechanischen oder pneumatischen Systemen zu elektronischen Kontroll- und Steuersystemen statt. Während bei nicht sicherheitskritischen Systemen dieser Wechsel schon sehr weit fortgeschritten ist, zögern die meisten Hersteller noch beim Einsatz von elektronischen Systemen für sicherheitskritische Anwendungen wie z.B. *x-by-wire*. Ein wichtiges Argument das gegen den Einsatz dieser neuen Technologien spricht ist, daß sich für die meisten Systeme aufgrund ihrer Komplexität wohl kein formaler Nachweis ihrer Sicherheit mehr erbringen läßt. Hier sollen technischen Studien, Labortests und Prototypen diese Sicherheit gewährleisten. Die Wichtigkeit solcher Untersuchungen zeigt sich jetzt beim CAN-Protokoll, das sich in der Automobilindustrie bereits gut etabliert hat und bei dem erst kürzlich eine schwerwiegende Designschwäche entdeckt wurde.

In dieser Arbeit wird das Fehlerverhalten der *Xerxes* Kodierung untersucht. Die *Xerxes* Kodierung ist eine flankengesteuerte und gleichstromfreie Bitkodierung. Nach maximal 2,5 übertragenen Datenbits wird ein Pegelwechsel der Übertragungsleitung garantiert, womit sich *Xerxes*-Kodierte Signale auch über ferromagnetische Übertrager hinweg ausbreiten können.

Der Grund dieser Untersuchung ist eine vor kurzem erschienene Publikation über die Anfälligkeit für Fehlerfortpflanzung im CAN Protokoll welche dazu führt, daß zwei Übertragungsfehler zu sechs oder mehr Bit-Fehlern in den empfangenen Daten führen, die mit dem verwendeten CRC nicht mehr sicher erkannt werden können. Ziel dieser Arbeit ist es, eine umfassende Untersuchung des Fehlerverhaltens der *Xerxes* Kodierung durchzuführen, mögliche Probleme, die sich daraus ergeben, aufzuzeigen und Empfehlungen für die Protokollentwicklung und für den Entwurf integrierter Schaltkreise zu geben.

Die Untersuchung ist in zwei Teile gegliedert. Der erste Teil prüft das statische Verhalten der *Xerxes*-Kodierung, d.h. die Kodierung wird vollständig auf Fehlerfortpflanzung getestet wobei der Einfluss von dynamischen Effekten wie sie bei der Signalabtastung auftreten nicht untersucht wird. Der zweite Teil dieser Arbeit untersucht die dynamischen Effekte der Signalabtastung mit Hauptaugenmerk auf asymmetrische Verzögerungen in den Sende-Empfangseinheiten.

Xerxes Error Behavior

Abstract

In today's automotive industry a fundamental paradigm shift from pure mechanical or hydraulic systems towards electronic control and steering systems is taking place. While electronic communication buses are already widely used for non-safety critical subsystems most manufacturers hesitate to introduce electronic communication systems for safety critical systems such as x-by-wire. The most important reason against the use of these new technologies is that most of the current systems are far too complex to prove their safety analytically. Thus studies, laboratory tests and prototypes are required by the automotive industry to ensure the required safety standards. The importance of these studies gets obvious in the CAN protocol now, a protocol, well established and widely used in the automotive industry. But only recently a grave Design flaw has been discovered.

This thesis examines the error behaviour of the *Xerxes* coding scheme. The *Xerxes* coding scheme is an edge triggered and DC free bit coding scheme. The use of the *Xerxes* bit coding scheme grants the maximum time between two edges to be less than 2.5 bit cell durations, allowing the use of ferromagnetic isolating transformers.

The main reason for this study was a recent publication about multi bit error vulnerabilities in the bit stuffing encoding used in the CAN protocol, where two transmission errors can lead to six or more bit errors which cannot be securely detected by the used CRC. The aim of this study is to investigate in detail the error behaviour of the *Xerxes* encoding scheme, identify possible problems and give recommendations for protocol- and chip-design.

The examination of the *Xerxes* encoding scheme is divided in two parts. The first part examines the static behaviour of the *Xerxes* coding scheme. This is done by checking the coding completely for error propagation without any dynamic influences like the effects of bit sampling. The second part covers the dynamic effects of bit sampling with the main focus set on asymmetric delays such as caused by most transceivers.

To my parents and all my friends

and to all my past, my present and my future rats.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

In the automotive industry there is an increasing demand for "*x-by-wire*" systems. "By-wire" is a term that describes the replacement of mechanical or hydraulic connections between individual subsystems by electronic connections (wires). The "x" denotes the possible fields of applications that shall be implemented, like *steer-by-wire* or *break-by-wire*. This makes it also possible or easier to introduce driver-independent assistance systems to increase the road safety. The assistant systems currently in use like *Anti Lock Breaking Systems* (`ABS`), *Anti Slip Regulations* (`ASR`) or *Electronic Stability Program* (`ESP`) cause a high technical effort, which can be significantly reduced by *x-by-wire* systems.

Before mechanical or hydraulic functions can be replaced by electronic communication it has to be ensured that this will not introduce a new safety risk but in contrary result in an additional gain of safety and comfort. Therefore a high degree of reliability and deterministic behaviour is required for this electronic signalling. Last but not least this connections are supposed to transfer a maximum number of signals to reduce the necessary cabling.

With conventional bus systems the necessary safety and performance requirements cannot fully be met at the same time. The upcoming *Local Interconnect Network* (`LIN`) protocol is too slow, non deterministic and to unreliable. The *Controller Area Network* (`CAN`) protocol is able to meet the necessary speed requirements but the bus access is *event triggered* and therefore `CAN` is not deterministic. With increasing network traffic, the maximum delay time until which messages are sent cannot be predicted. Such a non deterministic behaviour is not acceptable for safety critical applications like *break-by-wire* or *steer-by-wire*. As consequence *time triggered* transmission protocols and applications become necessary.

1.1.1 The Need for Safety

Some of the major players in the automotive and supplier industry have joined forces in developing the FlexRay™ protocol which is supposed to meet the high requirements of *drive-by-wire* systems.

All stages of the protocol development have so far been accompanied by *Field Programmable Gate Array* (`FPGA`) devices to test the behaviour of the transmission protocol in real time and on real hardware (although many functions have been tested in software simulations before). At the time of this writing, a major car manufacturer is starting to build a prototype car based on the FlexRay™ bus architecture. All these actions are taken to create a protocol that is safe by design. An other idea behind this approach is openness. Once the protocol is finished any manufacturer can join the FlexRay™ consortium, obtain the specifications and develop FlexRay™ applications royalty free. This will guarantee appropriate prices and independence from a single supplier.

In *drive-by-wire* systems, even a short disruption of service can become the cause of an accident. Assuming the same failure rate as in the aerospace industry, which is 0.1 *fit*[1], about four accidents would be caused alone in the USA [FRpre 01] within one year. This is acceptable considering the fact that *x-by-wire* systems contribute to the overall safety on the road.

Unfortunately, traditional fault tolerant system design experience is not directly applicable to mass consumer products. A failure that occurs with a probability of 1 *fit* would occur every 73 years in the American aerospace fleet; the same failure rate will result in a failure every 4.5 days in the automotive fleet [Koopman 98]. Thus, small, acceptable fault tolerances in traditional fault tolerance applications may not be small enough to ensure safety in consumer applications.

1.2 Overview

The analysis is split into two parts. The first part examines the stability of the Xerxes encoding scheme without the influence of bit sampling to find out, if n bit errors can produce $n + m$ errors in the decoded bit stream. The second part examines the influence of bit sampling on the error behaviour by applying various kinds of errors on the transmitted signal and feeding the generated signal into different sampling logics.

The study is structured as follows: The second chapter gives an introduction to the Xerxes encoding scheme and a short overview of two other related publications. The third chapter describes the internal architecture and usage of the simulator used in the presented study. The fourth chapter describes the simulations run for code coverage and the two possible fault models. Even if error propagation can occur with one failure model but not with the other both fault models give similar results when faults are not injected systematically but with a specified error rate. The fifth chapter describes the simulation runs for resynchronisation. Various faults are injected in the time domain before the signal is sampled. The impact of the individual faults on different sampling methods is evaluated. The last chapter provides a short conclusion and gives suggestions for further research.

[1]one fit equals one failure per trillion operating hours or a failure rate of $10^{-9}h^{-1}$

Chapter 2

Background

2.1 Xerxes Coding Scheme

The Xerxes coding scheme is defined in US Patent 4,234,897 [Miller 80] with the main goal to provide a DC free data transmission and is based on MFM (Modified Frequency Modulation). For a more detailed description of the Xerxes encoding scheme see also Wolfgang Forster´s diploma thesis [Forster 01].

2.1.1 Definitions

Xerxes is an edge triggered encoding scheme. Edges may occur on two distinct points:

- the begin and end of each bit cell, which is referred to as clock point
- the middle of each bit cell, which is referred to as data point

2.1.2 Decomposition of Binary Data

The Xerxes coding scheme decomposes the bit stream into four distinct sub sequences:

A any number of 1s without 0s (1. . . 1)
B any even number of 1s with a leading and a closing 0 (011. . . 110)
C any odd number of 1s with a leading and a closing 0 (011. . . 10)
D a pair of 0s (00)

2.1.3 Encoding of Sequences

The sub sequences of type A to D are encoded according to the following rules:

A transitions at the data point of each bit cell (like MFM)
B transitions at the clock point of the begin of each pair of 1s and the end of the last pair of 1s
C transitions at the clock point of the begin of each pair of 1s and the data point of the last (odd) 1
D transitions at the clock point at the end of each bit cell

2.1.4 Look Ahead

For correct encoding and decoding an incoming ahead of 1 bit is required, as the encoding and decoding state machines show. From figure 2.1 one can see that the next bit or edge has influence on the en/decoding of the current bit.

2.1.5 Error Conditions

The Xerxes encoding scheme defines the following assertions on Xerxes-encoded signals:

- the minimum delay between two subsequent edges lasts at least one full BCD
- the maximum delay between two subsequent edges is shorter than or equal to 2.5 BCD units
- the Running Digital Sum RDS[1] of the signal is always less or equal than 1.5

For signals not compliant with these assertions a code violation must be recognised by the Xeres Decoder. The third assertion may not be very important when decoding the signal but it is important when designing isolating transformers for galvanic isolation.

2.1.6 Xerxes Basic Decoding Rules

There are three basic rules when decoding Xerxes encoded signals:

- bit cells containing transitions at the data point are decoded to “1”
- bit cells with no transition between them are decoded to “11”
- all other bit cells are decoded to “0”

While these restrictions allow correct decoding of a Xerxes stream they do not imply that the received signal is correct. With these basic rules a signal without any transition would be decoded as a series of “11”s (second rule) even if this is a code violation according to chapter 2.1.5.

2.1.7 Xerxes Encoding Example

Figure 2.1 shows a sequence of bits encoded using NRZ and Xerxes. The sub-sequences as described in chapter 2.1.2 are shown below the Xerxes encoded signal.

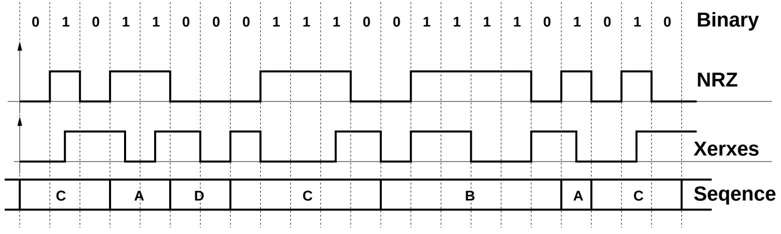

Figure 2.1: Xerxes Coding Sequence Example

2.1.8 Encoding State Machine

The encoding state machine shown in figure 2.2 is based on Wolfgang Forster’s diploma thesis [Forster 01]. The designation of state transitions is changed from the original to notations like “1,0/D”. This signifies that the transition is taken, if the current bit is “1” and the next bit is “0” and a edge at the data point is generated. Two or more values on one side of the colon represent more possibilities and a dash represents all possible values. An edge can have more than one designations meaning that the corresponding transition is taken, when either one of the conditions is fulfilled.

Please note that in this state machine the states for generating the Frame Completion Bit (FCB) and Code Violation Avoidance Bit (CVAB) [FR_PS 01] are missing. The reason is that the generated sequence depends on the state of the transmission line, which would be difficult to track using the state machine as it would double the necessary states. There the frame completion sequence is created in an own state machine.

[1]the RDS can be calculated as $\int u dt$ with $1\,\texttt{BCD} = 1$ and $u = \begin{cases} 1 & for \text{ “1”} \\ -1 & for \text{ “0”} \end{cases}$

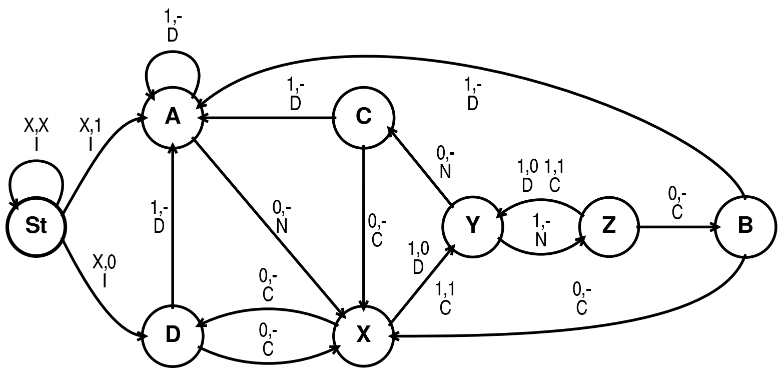

Figure 2.2: Xerxes Encoding State Machine

2.1.9 Decoding State Machine

For the notation of the transitions the same notation as in chapter 2.1.8 is used. The difference is that the machine reads "C", "D" or "N" Symbols and writes bits. "C", "D", and "N" are notations for edge at clock point, edge at data point and no edge. The state machine can be generated by reverse traversal of the encoding state machine. To distinguish between "B" and "C" sequences an additional state "W" has to be introduced.

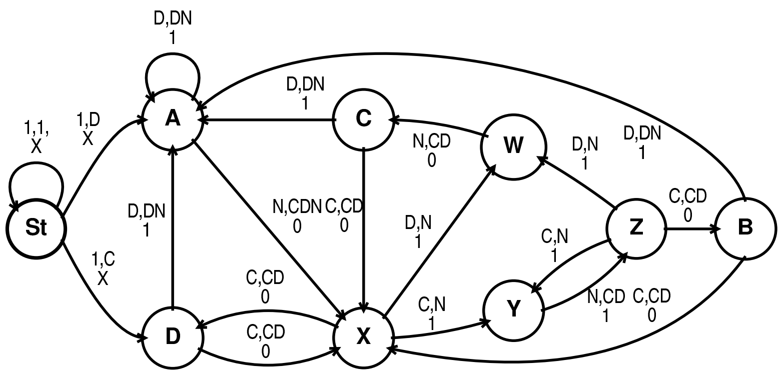

Figure 2.3: Xerxes Decoding State Machine

2.2 Related Work

There are two papers closely related to this study. In the following an overview of the related chapters is given.

2.2.1 Multi-Bit Error Vulnerabilities in the CAN Protocol [Tran 99]

In [Tran 99] the CAN protocol is examined and shown that the effective hamming distance of CAN drops to two by using a weakness, that is specific to the bit stuffing encoding.

The bit stuffing algorithm used in the CAN protocol works as follows: If five subsequent bits of the same level are transmitted, then one complementary bit is inserted into the data stream. The receiver detects five bits of the same level, checks the next bit if it has a complementary level and removes it afterwards. Figure 2.4 shows the stuffing- and destuffing process.

With bit stuffing used for transmission, all single bit errors can be detected either by the CRC or by the message length. Problems arise, when two transmission errors occur like shown in figure 2.5.

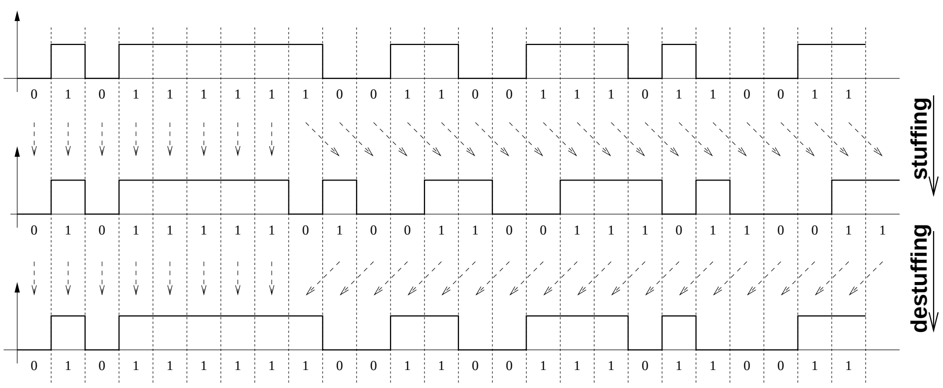

Figure 2.4: Bit Stuffing without Transmission Errors

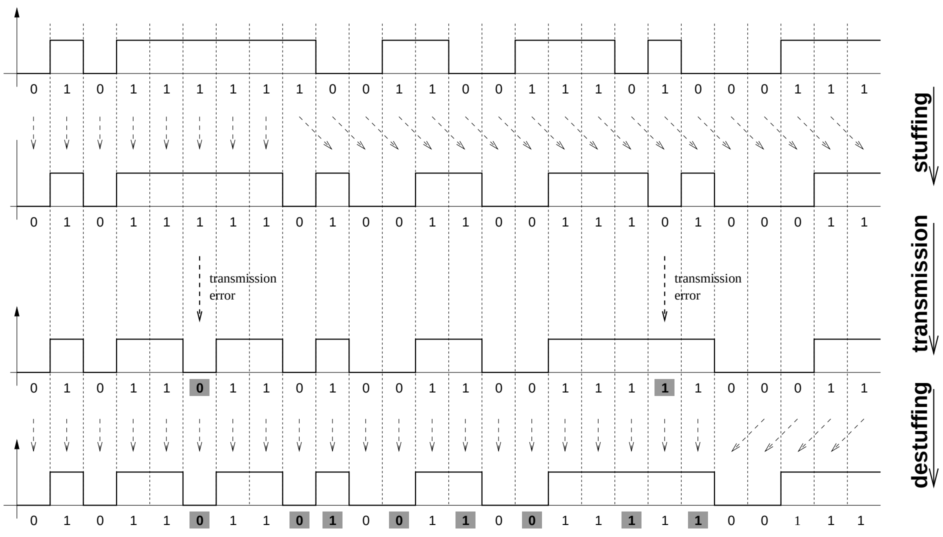

Figure 2.5: Bit Stuffing with two Transmission Errors

In figure 2.5 two bit errors occur during transmission. The first bit breaks a sequence of five equal bits, preventing the stuffing bit from being removed and delaying each following bit by one bit cell. The second error generates a sequence of five equal bits, causing the next bit falsely to be recognised as stuffing bit. The removal of this bit brings all following bits back to their original position keeping the correct frame length. All bits between both failures may be modified and since this may be more than five the errors cannot be recognised by the CRC. The effective Hamming distance dropped to two.

2.2.2 A Preliminary Analysis of the FlexRay Protocol [FRpre 01]

In [FRpre 01] design issues of current implementations of real time systems are compared to the FlexRay™ protocol. This paper does not give an in-depth analysis of Xerxes encoding because it examines all components of the real time system.

The survey of the signalling, as required by [Freq 02], leads to two conclusions that are directly related to this study:

- an end delimiter could significantly improve the error detection rates
- Miller-style encoding schemes like Xerxes are more vulnerable to small noise pulses because one distorted semi-bit results in an inverted decoded bit

Furthermore, the authors of this paper question the need for an additional DC free signalling scheme for the FlexRay™ protocol.

Chapter 3

Simulation Approach

3.1 Simulation Tool

The simulator used for this study is written in `C++` and runs under Linux. It conforms to ISO `C++` and should also compile and run on other platforms. `C++` was preferred to VHDL because it is easier to implement, runs faster and gives more freedom on modifying signals and filter parameters. As a result of basic optimisation for speed the simulator can simulate more than 1.000.000 frames of 64 bits in one minute on an AthlonXP 1900, a value that is hard to reach with a VHDL simulation.

The simulator can perform timed and non timed simulations. Non-timed simulations are used to check, if n transmission errors may produce n+m bit errors without the influence of bit sampling. Errors are modelled by changing the occurrence of edges within discrete values, moving from one sampling point to another. The lowest protocol level in code coverage simulations is the logical line interface. Timed simulations are used to examine the impact of different transmission errors on the physical line on various bit sampling methods an the bit errors in the resulting decoded bit stream. Errors are modelled by changing the occurrence of edges with analogue values before bit sampling. The lowest protocol level considered by timed simulations is the physical line.

The simulator consists of three basic building blocks: data structures, en-/decoders and filters. Data structures store different kind of information (timed signal, sampled signal, binary data) and are connected by en-/decoders, that convert between two different data representations, or filters, modifying the stored data within on representation form.

3.1.1 Data Structures

The data structures used by the simulator are defiend as following:

1 binary data (class `BinStream`) consists of a sequence of bits plus the symbol X for “don’t care”. No timing information is stored in this class.

2 code data (class `CodeStream`) represents edges within a bit cell and consists of the Symbols C (clock point), D (data point), N (none), B (both, clock and data), I (inactive, recessive state) and E (error). There is also no timing information needed, since one bit cell contains exactly one of these symbols.

3 edge data (`vector<double>`) represents edges in the transmitted signal. Each value in the vector represents the time an edge occurs in *Bit Cell Duration* (`BCD`) units.

3.1.2 De- and Encoders

The conversion between these codes is done by de- and encoders that are implemented as classes and derived from an abstract super class. The used de- and encoders (XerxesDecoder and XerxesEncoder) are created at program start and stored as global variable pointers. This makes the simulator usable for other encoding schemes by simply creating new de/encoder classes and assigning them to the global de/encoder in the main function. The existing decoder and encoder implementations are:

1 Encoder (XerxesEncoder): BinStream → CodeStream
2 Decoder (XerxesDecoder): CodeStream → BinStream
3 TimeEncoder: CodeStream → vector<double>
4 TimeDecoder: vector<double>→ CodeStream

3.1.3 Filters

The Filter classes provide mechanisms for manipulating the time when edges occur or for filtering out edges. All filters are sub-classed of the abstract class `Filter` that provides a common interface for filtering, supplying help information and making the filter available to the user. Complex chains of Filters can be specified on the command line interface. Available Filters are: Shift, Scale, Jitter, Asym, High and Low. For detailed explanation of the filters please refer to chapter 5.4. Most of the filters can be applied to a given range of edges or in a specified time interval. The range may be given in `BCD` units (absolute from 0 or from frame start) or relative, by the edge number. A modifier like “`start=@2:stop=@2`” can be used to modify only the second edge in a frame.

3.2 Usage of the Simulator

The simulator is command line driven. Coverage simulations produce only one line output and can be evaluated manually. Timed simulations are started from a shell script and logged to a text file. After the completion of the simulations the log file is parsed by a Perl script and the results are stored in a `SQL` database (PostgreSQL). The powerful `SQL` language allows a fast evaluation of the results. An alternative storage method would be an Excel file. The disadvantages of an Excel file are the stability (Excel seems to behave unpredictable for tables with 267.300 rows), the usability (only very basic commands) and the performance. Rational databases like PostgreSQL are optimised to give the user a very powerful set of commands. These commands are passed through a query optimiser and fed to the database engine resulting in high performance while in Excel, using auto filters, no optimisation is done. A description of the used database including table definitions and query examples can be found in the chapter 3.4.

The command line interface of the simulator is designed to support all features and settings of the simulators. There are no parts of the code that have to be enabled or disabled during compilation except for debug messages, therefore it can easily be used on small CD bootable Linux distributions without a `C++` compiler when compiled statically. All output is written to `stdout` and can easily piped into other commands. This is necessary when the simulator is run over secure shell on different (faster!) machines. Listing 3.1 shows the commands and options available on the command line.

```
> ./Xerxes --help

Xerxes C++ Simulation
*********************
```

```

usage: Xerxes OPTIONS
test selection:
-c, --coverage                             Xerxes symbols coverage
-f, --frame                                test with CRC checking over random frames
-t, --timed                                timed simulation (without crc)
-b, --ber_frame                            simulate bit error rate
    --test-encoder [01][01]...             test encoder binary -> code
    --test-decoder [NCD]...                test decoder code -> binary
    --test-timeenc [NCD]...                test time encoder code -> double array
    --test-timedec v1,v2,v3,..             test time decoder double array -> code
    --test-filter  v1,v2,v3,..             test filter chain on v1,v2,...
    --help                                 this help message
    --helpfilter                           help: filters (fault injection)
    --helptd                               help: bit sampling parameters

test parameter modification     (applies to):
-s, --seed          N            (all) random generator seed (0)
    --half                       (c,b) half bit fault model
    --full                       (c,b) full bit fault model
    --start         encoderstate (ch)  start symbol for xerxes encoder
-e, --errors        N            (f)   N errors in each frame, -1=>1..8 (-1)
-p, --framecount    N            (f,t) run N tests per frame (10.000)
-n, --count         N            (f,t) run N frames in following test (10.000)
-l, --framelen      N            (f,t) length of frames (32)
-a, --add-filter    Filter       (t)   add filter to chain (--helpfilter)
-d, --del-filter                 (t)   delete filter chain
-r, --timedec       tdstring     (t)   set receiver values (--helptd)
    --ber           N            (b)   use given ber (def 1e-3)

option availability:
(c)  option available only in coverage simulation
(ch) option available only in coverage simulation with half bit fault model
(b)  option available only in bit error rate simulation
(f)  option available only in time simulation without CRC
(t)  option available only in timed simulation
```

Listing 3.1: Simulator Command Line Interface

Filters are specified through the `-a` option (see listing 3.1). The filters are applied in the same order than they are specified on the command line. The simulator provides detailed help about the possible filter methods by using the `--helpfilter` command line option as shown in listing 3.2.

```
> ./Xerxes --helpfilter

FILTER HELP
***********
available filters:
offset=displacement[:start={@+*}time][:stop={@+*}time]
add a time displacement (delay) to every event
the delay may be negative and is given in BCD units
scale=scale[:start={@+*}time][:stop={@+*}time]
scale the time axis around the edge given in start
this filter can be used to simulate oscillator drifts
jitter=jitterwidth[:start={@+*}time][:stop={@+*}time]
add a random offset to each edge (normal distribution
jitterwidth=3*sigma, values from -jw to +jw)
asym[:r=risetime][:f=falltime]
add an asymetric delay to each edge
first edge is falling, times in BCD units
```

```
high:{+*}time#duration
pull signal to high add time for duration [in BCD]
example: high:+20#0.4 will pull the signal to high
for 0.4 BCD at edge 20 BCD after frame start
low:{+*}time#duration
pull signal to low add time for duration [in BCD]
example: low:+20#0.4 will pull the signal to low
for 0.4 BCD at edge 20 BCD after frame start

arguments:
<arg>     mandatory arguments
[arg]     optional arguments
ab        multiple possibilities
@n        nht edge starting from 1, (**default all = @0,@0)
*5.5      absolute time in BCD units (frame starts at 5.0)
+8.9      relative time in BCD units (from frame start)
#0.5      relative time from first selected edge
:         argument seperator, may be omitted at the end
examples for filter expressions:
scale=1.0001           creates a slow clock drift (oscillator drift)
offset=0.5:start=@2    delays each but the first edge by 0.5 BCD
```

Listing 3.2: Filter Selection (Fault Injection)

Sampling parameters are set similar to the filter parameters, before the `-t` option. A summary of the options can be seen in listing 3.3.

```
> ./Xerxes --helptd

RECEIVER HELP
**************
receiver options:
rcv_win=0.5       receive window in BCD (rcv\_win2 for window half)
sync_win=0.5      scncronisation window in BCD (sync\_win2 for window
                  half)
                  >0 centered around 0, <0 -syncwin...0
offset=0          error in first synchronisation
gran=0.125        granularity (1/oversampling rate)
asym=0            use asymmetric sampling method
r                 reset to defaults
```

Listing 3.3: Sampling Parameters Selection

The example in Listing 3.4 shows a typical simulation run. The `-r` option sets the receiver to a receive interval of 0.5 `BCD` and a synchronisation interval of 0.5 `BCD`, preventing dynamic synchronisation. The edges in the 23rd Bit Cell are delayed by 0.1 `BCD` (10ns) and a worst case clock drift of 6000 ppm, set by the scale filter, is assumed. The next options add a jitter of 0.1 ns to each edge and simulate a transceiver with asymmetric delay. The frame length is set to 256 and the simulation is run for 1000 different frames with 1000 repeats for each frame. This is not necessary but recommended since the jitter filter may have an impact on the results.
Listing 3.4 shows the result of a simulation run with 1.000.000 simulated frames. The receiver was badly chosen so no errors could be detected by the sampling logic (`receiver`). Since the injected fault was rather grave, 999.851 errors were detected by the Xerxes decoder (`Xerxes`). No frames with less than six errors occurred (`CRC`) and 149 frames with six or more errors (`>=6`) were received. In this example no valid frame could be received. This can be explained with the receiver that lacks resynchronisation while forcing a clock drift of 6000 ppm.

```
> ./Xerxes -r rcv_win=0.5:sync_win=0.5 -a
offset=0.1:start=+23:stop=+24 -a scale=1.006 \
-a jitter=0.001 -a asym:f=0.2 -l 256 -p 1000 -n 1000 -t
len         Frames       Receiver         Xerxes            CRC     >=6 errors
256        1000000              0         999851              0            149
>
```

Listing 3.4: Simulation Example

3.3 Implementation Details

The following chapters contain a detailed description of the implementation of the simulator. The basics should be understandable without detailed knowledge of C++. For problems with the code snippets Bjarne Stroustrup's C++ Language reference [C++Ref] is recommended. Good knowledge of the STL is not required but advantageous since the code uses standard C++ components from the STL wherever it is possible and advantageous. So the complete simulator could be implemented in under 3.000 lines of code[1] and can be compiled in only 25 seconds.

3.3.1 Data Structures

The data structures used for the implementation are similar to data structures of the *Standard Template Library* (STL). A small performance gain is achieved by providing an non-STL implementation, since the STL provides dynamic array resizing which is not used but requires some additional checking at the cost of performance.

3.3.1.1 The BinStream Class

As already mentioned above, the class `BinStream` is used for the internal representation of sequences of bits. The class definition can be found in listing ??. The `BinStream` is not only used for the representation of data bits. As the constructor `BinStream(const CodeStream& cs)` shows it can also be used to represent half bits. This feature is used for the coverage simulation presented in chapter 4.2. Each bit is represented by a byte with the values "0" for logic 0, "1" for logic 1 or "2" for logic x. This needs more storage than a bit field but it improves access speed since no shift and mask operations have to be performed to gain access to a single bit. A `BinStream` object can also be created from a string containing any number of "0"s, "1"s or "x"s. The conversion from each character contained in the string to bytes and vice versa is done by the utility functions `to_char` and `from_char`. Another way to create a `BinStream` instance is by specifying a long variable that holds the bits to be placed in the `BinStream` and the size, the new bit stream should have. The maximum size of `BinStream`s that can be created this way is architecture dependent and limited by the bit size of long values. Since only twelve bits are needed and virtually all architectures provide long values with more than 16 bits this restriction is not expected to cause any errors. Finally random `BinStream`s can be created using the static member function `random` which allows the specification of the start sequence and the desired length of the BinStream. Additional methods exist to get the current size of the `BinStream`, to append data and to clear the `BinStream`.

In addition this class provides also the possibility to calculate a CRC from the stored data. The CRC can either be returned from the `CRC` member function or appended to the `BinStream`. It is also possible to check a given CRC against the `BinStream`.

[1]this counter includes all lines, also the empty ones and unfrequent comment lines

```
typedef unsigned char byte;
typedef unsigned short CRC;

class BinStream
{
 protected:
    const static int BinStreamSize = 300;
 public:
    inline             BinStream();
    inline             BinStream(const char* _src);
    inline             BinStream(const CodeStream& _cs);
                       BinStream(const BinStream& _bs);
                       BinStream(unsigned long _bits, int _size);
    static BinStream& random(const char* _start, int _size);

    static const byte B0 = 0;
    static const byte B1 = 1;
    static const byte BX = 2;
    static char        to_char  (const byte _b);
    static byte        from_char(const char _c);

    inline void        clear();
    inline int         size() const;
    void               append(const char* _src);
    void               append(const byte  _bit);
    void               append(const byte* _src, int _length);
    void               remove(int _length);

    inline BinStream& operator = (const char* _src);
    inline BinStream& operator = (const BinStream& _bs);
                       operator std::string() const;
    const byte         operator [] (int _addr) const;

    CRC                crc(int _skip_start=0, int _skip_end=0) const;
    void               append_crc(int _skip_start=0, int _skip_end=0);
    bool               check_crc(int _skip_start=0, int _skip_end=0) const;
 protected:
    void               assign(const BinStream& _bs);
    byte               m_data[BinStreamSize];
    int                m_size;
    // CRC
    static const CRC   CRC_Poly        = 0xc599;
    static const int   CRC_Width       = 16;
};
```
Listing 3.5: `BinStream` class

3.3.1.2 The Codestream Class

The `CodeStream` class is similar to the `BinStream` class. It holds the representation of already sampled edges, which may occur in the clock point (`C`), in the data point (`D`), in both sampling points (`B`) or not at all (`N`). Additionally a symbol for inactive, recessive state (`I`) and an error symbol (`E`) exists. The individual symbols are encoded numerically using the member functions `from_char` and `to_char` and stored in a byte. As in the `BinStream` the internal storage is implemented as a simple array of bytes. Besides the usual member functions to get the size, clear the entire `BinStream` and append data the member functions `begin` and `end` are implemented. The `begin` and `end` member functions return, depending on the calling semantics, an `iterator` or `const iterator`, a concept widely used in the

STL. Iterators are the connection between data structures and algorithms. When a class implements the necessary iterator functions all the available algorithms implemented in the STL can be used.

```
typedef unsigned char byte;

class CodeStream
{
 protected:
    const static int CodeStreamSize = 300;
 public:
    typedef byte*          iterator_type;
    typedef const byte*    const_iterator_type;

    static const byte I = 0;
    static const byte N = 1;
    static const byte C = 2;
    static const byte D = 3;
    static const byte B = 4;
    static const byte E = 5;

    inline              CodeStream();
    inline              CodeStream(const CodeStream& _cs)
                        CodeStream(const BinStream& _bs);

    static char         to_char  (const byte _b);
    static byte         from_char(const char _c);

    inline void         clear();
    inline int          size() const;
    inline void         append(const char* _src);
    inline void         append_data(const byte _src);

    inline CodeStream& operator = (const char* _src);
    inline CodeStream& operator = (const CodeStream& _cs);
    operator std::string() const;

    inline const char getc(const int _addr) const;
    inline const byte getb(const int _addr) const;
    inline void       setc(int _addr, const char _val);
    inline void       setb(int _addr, const byte _val);

    inline iterator_type       begin();
    inline const_iterator_type begin() const;
    inline iterator_type       end();
    inline const_iterator_type end() const;
 protected:
    byte  m_data[CodeStreamSize+1];
    byte  m_dummy;
    int   m_size;
};
```

Listing 3.6: CodeStream class

3.3.1.3 Storing Edge Data using a vector<double>

Edge data is stored in a simple STL vector<double>. A vector is the STL implementation of a dynamic resizeable array with random access on the individual elements. Another possibility to store edge data would be a list. The advantage of a list over a vector is the ability to insert (new edges) or delete (lost edges) elements anywhere at virtually no cost. When inserting or deleting elements of vectors, all elements behind the newly inserted or deleted one have to be moved (copied) which has significant impact on the performance. The disadvantage of the list is that the additional pointers necessary for list management require additional space and a non-linear processing of all list elements is slower than with a vector. Random access is also not possible within a list. Searches are slower because no binary search can be performed. The overall performance of a list and of a vector should be about the same in this application.

3.3.2 De- and Encoders

Some of the de/encoders that are only used to change between the different data representations[2] are very simple and are implemented as constructors of the individual data classes while others are more difficult and represent complex processes like bit sampling or applying a complicated encoding scheme to a series of bits or symbols. These de/encoders are implemented in classes. This approach gives the advantage of a clear interface design. Additional coding schemes or sampling logics can be implemented by sub-classing. Function pointers are a different approach used by standard `C`. Function pointers have the drawback that no state information can be stored. This would require static variables in the individual functions which could not be shared between different functions. This problem is solved by an additional argument in the function call: a pointer to a memory location where the necessary information can be stored. In fact the `C++` compiler does exactly the same with the class information transparently to the user with no performance loss[3].

```
class Encoder {
 public:
    virtual void encode(const BinStream& BS, CodeStream& CS) = 0;
};

class Decoder {
 public:
    virtual void decode(const CodeStream& CS, BinStream& BS) = 0;
    static const char* errstring(int _reason);

    static const int CodeError    = 1;
    static const int SamplerError = 2;
};
```

Listing 3.7: `Encoder` and `Decoder` classes

Listing 3.7 and listing 3.8 are showing the interface of the decoder and encoder classes, which is, like already mentioned, very simple and clean. For special purposes this interface can be extended by adding member functions to the specific sub-classes, which has be done by the implementation of the XerxesEncoder and XerxesDecoder.

3.3.2.1 The XerxesEncoder Class

A sub-class of the `Decoder` is the `XerxesDecoder` as shown in listing 3.9. For the coverage test made in chapter 4 the additional command `setState` is necessary. The byte constants

[2]i.e. the conversion from CodeStream to BinStream when the BinStream is used as representation of the logic level in half bit cells (chapter 4.2)

[3]when using virtual functions there is a little performance loss since the address of the function has to be fetched from the class information table. This effort is implementation dependent but in most cases it is comparable to the effort of fetching a function address from a function pointer.

```
class TimeEncoder {
public:
    void encode(const CodeStream& _CS, std::vector<double>& _timevec);
};

class TimeDecoder {
public:
    virtual void decode (const std::vector<double>& _timevec,
                         CodeStream& _CS) = 0;
    virtual void set_values(const std::string& _arguments) = 0;
};
```

Listing 3.8: `TimeEncoder` and `TimeDecoder` classes

defining the states are public, so symbolic names can be used to set the states from outside the class. The encoding is table based. When creating a `Decoder` object the `init` function is called and populates the `m_transitions` array. The `init` function uses the `addTransition` method to add new transitions to the `m_transitions` array. The array index is eight bits wide and coded like shown in table 3.1 where the encoding of the array values can be found too. The encoding process is simple: the current state, the current bit and the next bit are used to calculate an array index. On the indexed position a value consisting of the next state and the generated code can be found. Another solution would be hard-coding the state machine using "case" and "if" statements. But the hard-coded solution has two drawbacks: first this solution is hard to maintain, changes are not easy to be made without possible side effects and the second but even more important drawback, is calculation speed. A series of "if" or "case" statements is slower than a look up table. In addition, the current solution generates code which is easy to read since it is a direct representation of the encoder state machine (as is seen in line 61 and below).

7	6	5	4	3	2	1	0
current bit		next bit		current State			

7	6	5	4	3	2	1	0
generated code symbol				next State			

Table 3.1: Bit Positions in the Index and Values of the `m_transitions` array

```
1  class Xerxes_Encoder : public Encoder {
2   public:
3                   Xerxes_Encoder(bool _frame_end = false);
4      static  void init();
5      void         encode(const BinStream& BS, CodeStream& CS);
6      void         setState(const byte _newstate);
7      byte         getState() const;
8      static const byte xSta  =   1;
9      static const byte xA    =   2;
10     static const byte xX    =   3;
11     static const byte xD    =   4;
12     static const byte xC    =   5;
13     static const byte xY    =   6;
14     static const byte xZ    =   7;
15     static const byte xB    =   8;
16     static const byte xFCS  =   9;
17     static const byte xF_C  =  10;
18     static const byte xF_N  =  11;
19     static const byte xSto  =  12;
```

```
 protected:
    static void  addTransition(const byte  _c_state, const byte  _n_state,
                               const char* _c_bit,   const char* _n_bit,
                               const byte  _symbol);
    inline byte  do_transition(byte& _State, byte _curr, byte _next);
    static byte  m_transitions[256];
    bool         m_frame_end;
    int          m_line;
    byte         m_start_state;
};

void Xerxes_Encoder::addTransition(const byte  _c_state, const byte  _n_state,
                                   const char* _c_bit,   const char* _n_bit,
                                   const byte  _symbol)
{
    while(*_c_bit != 0) {
        byte c = BinStream::from_char(*_c_bit++);
        const char* next = _n_bit;
        while(*next != 0) {
            byte n = BinStream::from_char(*next++);
            m_transitions[(_c_state & 0x0f) | ((n & 3)<<4) | ((c & 3)<<6)] =
              (_n_state & 0x0f) | ((CodeStream::from_char(_symbol) & 7) << 4);
        }
    }
}

inline byte Xerxes_Encoder::do_transition(byte& _St, byte _curr, byte _next)
{
    byte rv = m_transitions[(_St&0x0f) | ((_next&3)<<4) | ((_curr&3)<<6)];
    if(rv==0) throw int(1);
    _St = rv & 0x0f;
    byte code = (rv >> 4) & 7;
    if(code==CodeStream::C || code==CodeStream::D) m_line = 1-m_line;
    return code;
}

void Xerxes_Encoder::init()
{
    m_start_state = xSta;

    for(int i=0; i<=255; m_transitions[i++]=0);
    addTransition(xSta, xSta,   "X",  "X",    'I');
    addTransition(xSta, xA,     "X",  "1",    'I');
    addTransition(xSta, xD,     "X",  "0",    'I');

    addTransition(xA,   xA,     "1",  "01X",  'D');
    addTransition(xA,   xX,     "0",  "01X",  'N');
    addTransition(xA,   xF_C,   "X",  "X",    'N');

    addTransition(xD,   xX,     "0",  "01X",  'C');
    addTransition(xD,   xA,     "1",  "01X",  'D');
    addTransition(xD,   xF_C,   "X",  "X",    'C');

    // ... the rest of the init method is not shown in the listing
}
```

Listing 3.9: Xerxes_Encoder class

3.3.2.2 The XerxesDecoder Class

The `XerxesDecoder` is very similar to the `XerxesEncoder` in its implementation. Further differences can be seen in listing D.1. The only difference are the arguments of the member functions and the coding of the array index and value.

3.3.2.3 The TimeEncoder Class

An interesting example is the implementation of the `TimeEncoder::encode` function shown in listing 3.10. This example shows how comfortable the programming is in `C++`, especially following a clean class design. The usage of the `begin` and `end` iterator functions is also pointed out in this example. The clock value represents the simulation time and is preset to 5 `BCD`, where the transmissions are supposed to start. We need not take care of “`B`” signals since they can only occur due to errors[4].

```
void TimeEncoder::encode(const CodeStream& _CS, std::vector<double>& _timevec)
{
    double t_clock = 5.0; // start at 5 BCD
    for(int CodeStream::iterator_type i=_CS.begin(); i!=_CS.end(); ++i) {
        if (_CS.getb(i) == CodeStream::C) {
            _timevec.push_back(t_clock);
        } else if (_CS.getb(i) == CodeStream::D) {
            _timevec.push_back(t_clock + 0.5);
        }
        t_clock += 1.0;
    }
}
```

Listing 3.10: `encode` member function of the `TimeEncoder` class

3.3.2.4 The TimeDecoder Class

The more sophisticated class `TimeDecoder` implements the simulation of the receiver logic. It was possible to structure the class `TimeDecoder_Default` flexible enough so that all receiver implementations examined in this work could be simulated. The `TimeDecoder_Default` is the only sub-class of the `TimeDecoder` that was used, although the simulator framework would allow the use of multiple `TimeDecoder` sub-classes.

```
class TimeDecoder_Default : public TimeDecoder {
    // static decoder with adjustable receive Window
    // synchronisation: first edge => D (center+offset)
    //
    //
    //      |<windowwidth>|<g>|<windowwidth>|<g>|<windowwidth>|
    //      |             |   |             |   |             |
    //      |  |<rswin>|  |   |  |<rswin>|  |   |  |<rswin>|  |
    //   ----------|-----------------|-----------------|--------
    //             C                 D                 C
    //             |                                   |
    //             |<-------------- BCD -------------->|
    //
public:
    TimeDecoder_Default(const std::string& _arguments="");
    void set_values(const std::string& _arguments);
    void decode(const std::vector<double>& _timevec, CodeStream& _CS);
```

[4]under the assumption, the `XerxesEncoder` works correctly and does not produce errors

```
private:
    static const char* m_defaults;
    double m_rcv_win_half;
    double m_sync_win_half;
    double m_offs;
    double m_granularity;
    int    m_asym;
};
```

Listing 3.11: Definition of the `TimeDecoder_Default` class

As it can be seen in the comments, the first edge occurs always in the data point. This can be assumed since in the timed simulation always complete frames are sent, starting with the *Frame Start Sequence* (FSS) which consists of a series of “1” bits. Using the Xerxes encoding state machine from figure 2.2 it can be seen that “1”s after the idle state always are encoded with an edge in the data point. The receiver parameters are set either in the constructor or with the `set_values` member function using a string. The string is formatted as a series of `name[=value]` pairs separated by colons. This has the same format as on the command line and in fact it is a copy of the string specified using the `-r` argument on the command line. The string is parsed using an utility function that creates a mapping from key strings to values strings[5]. The values are returned by the operator `[]` using a syntax like “`my_double = strtod(config["rcv_win"].c_str(),NULL);`”. Listing 3.12 shows the implementation of the sampling logic in pseudo code, which is shorter and hopefully better understandable than the C++ code that can be found in the appendix D.2. A description of the intervals used for sampling is given in figure 5.1.

```
SET codestream = ""                               # clear the return value
SET transmission_line = 1                         # the bus is logic 1 when idle
SET internal_clock = time_of_first_edge-0.5-offset+0.5*sampling_interval
WHILE there are edges remaining BEGIN
  FOR sampling_point IN "clock", "data" DO
    SET VARIABLE_NAMED(sampling_point) = 1      # set clock/data to 0
    IF next_edge within internal_clock +/- 0.25 THEN
      SET this_edge = get_next_edge()
      IF last_edge was in the last sampling interval THEN
        THROW ERROR "two edges within " + sampling_point + " point"
      END IF
      IF this_edge NOT within internal_clock +/-receive_interval THEN
        THROW ERROR "outside receive interval"
      END IF
      SET diff = this_edge-internal_time+sync_window_offset
      IF diff NOT within +/-sync_interval THEN
        SET correction_term = floor(diff /sampling_interval)*sampling_interval
        SET internal_time = internal_time + correction_term
      END IF
      IF correction of asynchronous delays enabled THEN
        IF line==1 THEN
          SET internal_clock = internal_clock - async_delay_correction_factor
        ELSE
          SET internal_clock = internal_clock + async_delay_correction_factor
        END IF
      END IF
      SET line = 1 - line                         # toggle the line with each edge
    END IF
    SET internal_time = internal_time+0.5       # advance to next sampl. point
    SET VARIABLE_NAMED(sampling_point) = 1      # set either clock or data to 1
  END FOR
```

[5]the mapping is implemented using the STL container `map<string, string>`

```
  IF clock<>0 and data<>0 THEN  THROW ERROR
  ELSIF clock<>0 THEN           SET codestream = codestream + "C"
  ELSIF clock<>0 THEN           SET codestream = codestream + "D"
  ELSE                          SET codestream = codestream + "N"
  END IF
END WHILE
```

Listing 3.12: Sampler logic in Pseudo Code

3.3.3 The Filter Framework

Most of the implementation effort has been put into the filter framework. The filter framework consists of two classes, the `Filter` class with its sub-classes and the `FilterFactory` class. All filters provide an `info()` member function which returns a `FilterInfo` struct, containing the filter name, the help text, the default values and a function pointer. The function referenced by this pointer creates the specified filter and returns a pointer to a `Filter` object. In the constructor of the `FilterFactory` class all filters are registered and their `FilterInfo` structure is stored in an `STL` vector. To do the initialisation the `info()` method has to be declared static. When a filter operation is called from the `main` function this is done, using the `FilterFactory` class, which provides, besides the constructor, four public functions to influence the filter network. When using the `--helpfilter` command the method `Filter::help()` is executed. This function iterates over a vector containing the `FilterInfo` of all filters that are registered and print the help message of each filter plus an short help about options common to all filters. To append new filters at the end of the filter chain - which is the same as injecting faults - the `-a` option is used. The argument used with this option is directly passed to the `FilterFactory::append` method which is displayed in listing 3.13.

```
void FilterFactory::append(const std::string& _filter_desc)
{
    std::vector<std::string> temp;
    stringtok(temp, _filter_desc, ":=", "", 2);
    if(temp.size()>=1)
        for(std::vector<FilterInfo>::const_iterator i=m_finfo.begin();
              i!=m_finfo.end(); ++i)
            if(i->m_name == temp[0])
                m_filters.push_back(i->m_create(_filter_desc));
}
```

Listing 3.13: The `FilterFactory::append` method

The `append` method takes the argument and extracts the first token that is separated by a column. At next, it iterates over all `FilterInfo` structures stored in the vector `m_finfo` and compares the filter name to the first token of the argument. If a match is found, a new `Filter` is created using the `m_create` function which is provided by the `FilterInfo` structure. The new `Filter` is appended to the `m_filters` vector, which hold all pointers to the created filters. This approach allows to generate chains consisting of any number of serialised `Filters`. The `-d` command, which clears the entire filter chain is easy implemented. A call to `FilterFactory::clear` iterates over all pointers stored in `m_filters` and calls the `delete` function to destroy the `Filter` instances before `m_filters.clear()` is called to empty the vector storing the pointers to the `Filters`. Listing 3.14 shows the `FilterFactory::filter` method that is used to apply all filters of a chain to a series of edges stored in a `vector<double>`. The variable `_timevec` holds the timing information as described in chapter 3.3.1.3 and will be modified by each `Filter`. The full listing of the Filter framework can be found in the appendix D.3.

```
void FilterFactory::filter(std::vector<double>& _timevec)
{
    for(std::vector<Filter*>::iterator i=m_filters.begin();
          i!=m_filters.end(); ++i)
        (*i)->filter(_timevec);
}
```

Listing 3.14: The `FilterFactory::filter` method

3.3.4 The assembled Program

This chapter describes the code that combines the previous described classes and how they interact.

3.3.4.1 The `main` Function

The challenge in `C++` class design is the creation of classes in a way which allows connecting them with a minimum amount of "glue" between them. This is done with the `main` function and a few utility functions. The main function uses the `GNU getopt` library described in the representative manual page and is rather small. Listing 3.15 shows the `main` function.

```
int main (int argc, char* argv[])
{
    g_encoder  = new Xerxes_Encoder;
    g_decoder  = new Xerxes_Decoder;
    g_time_enc = new TimeEncoder;
    g_time_dec = new TimeDecoder_Default;

    int          errors=-1, count=10000, framecount=10000,
                 framelen=32, use_crc=0, modify=modify_half;
    double       ber = 0.001;
    CodeStream   CS;
    BinStream    BS;
    std::string  s, t, start_state("A");

    while (1) {
        int c;
        int option_index = 0;
        option long_options[] = {
            {"help",           0, 0, 'h' },
            {"coverage",       0, 0, 'c' },
            {"frame",          0, 0, 'f' },
            {"ber_frame",      0, 0, 'b' },
            {"errors",         1, 0, 'e' },
            {"framecount",     1, 0, 'p' },
            {"count",          1, 0, 'n' },
            {"seed",           1, 0, 's' },
            {"framelen",       1, 0, 'l' },
            {"timed",          0, 0, 't' },
            {"start",          1, 0, 0   },
            {"half",           0, 0, 0   },
            {"full",           0, 0, 0   },
            {"ber",            1, 0, 0   },
            {"helpfilter",     0, 0, 0   },
            {"add-filter",     1, 0, 'a' },
            {"del-filter",     0, 0, 'd' },
            {"timedec",        1, 0, 'r' },
            {"test-encoder",   1, 0, 0   },
            {"test-decoder",    1, 0, 0   },
            {"test-timeenc",    1, 0, 0   },
            {"test-timedec",    1, 0, 0   },
            {"test-filter",     1, 0, 0   },
            {"helptd",          0, 0, 0   },
            {0, 0, 0, 0}
        };

        c = getopt_long(argc, argv, "hcfe:p:n:s:l:ta:dr:b",
                        long_options, &option_index);
        if (c == -1) break;

        switch (c) {
          case 0:
            if (long_options[option_index].name == "half")
                modify = modify_half;
            else if (long_options[option_index].name == "full")
                modify = modify_full;
            else if (long_options[option_index].name == "ber")
                ber = strtod(optarg, NULL);
            else if (long_options[option_index].name == "start")
                start_state = optarg;
            else if (long_options[option_index].name == "helpfilter") {
                std::cout << Filters.help();
                exit(0);
            } else if (long_options[option_index].name == "helptd") {
                std::cout << td_help;
                exit(0);
            } else if (long_options[option_index].name == "test-encoder") {
                BS.clear();
                BS.append(optarg);
                CS.clear();
                g_encoder->encode(BS, CS);
                s=BS;
                t=CS;
                std::cout << "BS: " << s << std::endl
                          << "CS: " << t << std::endl << std::endl;
            } else if (long_options[option_index].name == "test-decoder") {
                CS.clear();
                CS.append(optarg);
                BS.clear();
                try { g_decoder->decode(CS, BS); }
                catch (int& e) { std::cout << "ERROR:" << std::endl; }
                s=BS;
                t=CS;
                std::cout << "CS: " << t << std::endl
                          << "BS: " << s << std::endl << std::endl;
            } else if (long_options[option_index].name == "test-timeenc") {
                std::vector<double> dv;
                CS.clear();
                CS.append(optarg);
                g_time_enc->encode(CS, dv);
                t=CS;
                std::cout << "CS:    " << t << std::endl
                          << "tvect: " << dv << std::endl << std::endl;
            } else if (long_options[option_index].name == "test-timedec") {
                std::vector<double> dv;
                read_double_vec(dv, optarg);
                CS.clear();
                try { g_time_dec->decode(dv, CS); }
                catch (int& e) { std::cout << "ERROR:" << std::endl; }
                t=CS;
                std::cout << "tvect: " << dv << std::endl
                          << "CS:    " << t << std::endl << std::endl;
            } else if (long_options[option_index].name == "test-filter")  {
                std::vector<double> dv_in, dv_out;
                read_double_vec(dv_in, optarg);
                dv_out = dv_in;
                Filters.filter(dv_out);
                std::cout << "in:  " << dv_in  << std::endl
                          << "out: " << dv_out << std::endl << std::endl;
            } else assert(0);
            break;
          case 'h':
            usage_exit();
          case 'e':
            errors = atoi(optarg);
            break;
          case 'l':
            framelen = 8 * (atoi(optarg)/8);
            break;
          case 'p':
            framecount = atoi(optarg);
            break;
          case 'n':
            count = atoi(optarg);
            break;
          case 's':
            srandom(atoi(optarg));
            break;
          case 'd':
            Filters.clear();
            break;
          case 'a':
            Filters.append(optarg);
            break;
          case 'r':
            g_time_dec->set_values(optarg);
            break;
          case 'b':
            ber_frame(framelen, count, framecount, ber);
            break;
          case 'c':
            coverage(start_state, modify);
            break;
          case 'f':
            crc_frame(framelen, count, framecount, errors);
            break;
          case 't':
            time_frame(framelen, count, framecount);
            break;
          default:
            usage_exit(-1);
        }
    }
    return 0;
}
```

Listing 3.15: The `main` function

All operations are performed by the `main` function. Exceptions are the utility functions `ber_frame`, `coverage`, `crc_frame` and `time_frame` and which are described later. Especially the `--test-x` functions could be implemented in a very compact way. Implementing supporting functions like "`ostream& operator << (ostream& _out, <class T>)`" for the `BinStream`, `CodeStream`, and `vector<double>` classes helps to keep the code short and clean.

3.3.4.2 The `coverage` Function

This chapter describes how the `coverage` function works. At the beginning the vectors code and transmission are filled with all possible twelve bit combinations (chapter 4.2 explains what "possible" combinations are and why they are twelve bits long) respective their Xerxes encoded counterpart, with the encoders internal state set to `start_state` at the begin of encoding. Bit patterns that can not be encoded from the `start_state` can be skipped. In the following loop all bit patterns and the transmitted signals are compared according to the selected fault model. If the number of different data bits is greater than the number of different half or full bits in the transmitted signal an error message is printed. A case where n transmission errors can cause $n + m$ errors in the received data has been found.

```
void coverage(std::string start_state, int modify)
{
    byte _start_state = string2state(start_state);
    if(_start_state>127) return;
    std::vector<BinStream> code, transmission;

    for(unsigned long str=0; str<4096; str++) {
        boolean skip = false;
        g_encoder->init();
        g_encoder->set_state(_start_state);
        BinStream  BS(str, 12);
        CodeStream CS;
        try { g_encoder->encode(BS,CS); }
        catch (int& e) { skip=true; }
        if (! skip) {
            code.push_back(BS);
            BinStream BS_trans(CS);
            BS_trans.append("00");
            transmission.push_back(BS_trans);
        }
    }

    for(int i=0; i<code.size()-2; i++) {
        for(int j=i+1; j<code.size()-1; j++) {
            int transmission_errors = 0;
            int data_errors = 0;
            for(int k=0,l=0; k<code.size(); k++,l+=2) {
                if(code[i][k] != code[j][k])
                    data_errors++;
                if(modify==modify_full) {
                    if(transmission[i][l] != transmission[j][l] ||
                       transmission[i][l+1] != transmission[j][l+1])
                        transmission_errors++;
                } else {
                    if(transmission[i][l] != transmission[j][l])
                        transmission_errors++;
                    if(transmission[i][l+1] != transmission[j][l+1])
                        transmission_errors++;
                }
                if(data_errors>transmission_errors &&
                   transmission[i][l] == transmission[j][l] &&
                   transmission[i][l+1] == transmission[j][l+1]) {
                    CodeStream CS1(transmission[i]);
                    CodeStream CS2(transmission[j]);
```

```
                        std::cout << "ERROR: " << transmission_errors
                                  << " < " << data_errors << std::endl
                                  << "b1:" << code[i] << "   =>   "
                                  << "c1:" << CS1 << std::endl
                                  << "b2:" << code[j] << "   =>   "
                                  << "c2:" << CS2 << std::endl;
                        return;
                    }
                }
            }
        }
}
```

Listing 3.16: The coverage function

3.3.4.3 The ber_frame Function

The ber_frame function is used to simulate frames with a given bit error rate. The half / full bit error rate corresponds to the probability that a single half / full bit is flipped. This simulation is used in chapter 4.3.3.2. The results of this simulation are very interesting and lead to a connection between the two fault models introduced in chapter 4. The listing below shows the ber_frame2 function that is repetitive called by the ber_frame function until the necessary number of frames have been simulated. The function is called with a BinStream representing the data bits. The data bits are encoded using the g_encoder, which is a Xerxes encoder by default. The resulting CodeStream is converted to a half bit representation and stored in the variable BS_halfbit_string_copy. In each pass of the loop, repeated until _framecount frames have been simulated, a copy of the half bit representation of the transmitted data is made. In this copy each full bit or half bit is toggled accordingly to the error model and the given bit error rate. The result is converted back to a CodeStream and fed into a Xerxes decoder. If the decoder detects the error it is counted using the _xer_det variable, else the number of erroneous bits are counted and the according counter value is increased.

```
unsigned long ber_frame2(BinStream& _frame, const int _framecount,
                         double _ber, unsigned long& _xer_det,
                         unsigned long& _crc_det, unsigned long& _not_det,
                         unsigned long errors[17])
{
    BinStream& BS_corr = _frame;
    CodeStream CS_corr;
    BinStream  BS_err;

    long randval = static_cast<long>(static_cast<double>(RAND_MAX-1) * _ber);

    BS_corr.append_crc(8);
    g_encoder->encode(BS_corr, CS_corr);
    BinStream  BS_halfbit_string_copy(CS_corr);

    for(int i=0; i<_framecount; ++i) {
        BinStream  BS_halfbit_string(BS_halfbit_string_copy);
        if(modify == modify_full)  // full bit fault model
            for(int j=0; j<BS_halfbit_string.size()/2-1; ++j)
                if(random() < randval)
                    toggle_fullbit(BS_halfbit_string, j);
        else                       // half bit fault model
            for(int j=0; j<BS_halfbit_string.size()-1; ++j)
                if(random() < randval)
```

```
                        toggle_halfbit(BS_halfbit_string, j);
                CodeStream CS_err(BS_halfbit_string);
                BS_err.clear();
                bool caught = false;
                try {
                    g_decoder->decode(CS_err,BS_err);
                } catch (int& e) {
                    // error detected by Xerxes decoder
                    caught = true;
                    ++_xer_det;
                }
                if(!caught) {
                    int diff = diffrence(BS_corr, BS_err);
                    if(diff > 15) diff=16;
                    errors[diff]++;
                    if(diff > 6) {
                        if(!BS_err.check_crc(8)) ++_crc_det;
                        else ++_not_det;
                    } else if (diff > 0) _crc_det++;
                }
        }
        return _framecount;
}
```

Listing 3.17: The ber_frame2 function

3.3.4.4 The crc_frame Function

Since the crc_frame function is nearly a verbatim copy of the ber_frame function it is not shown here. The only difference is the injection of faults shown in listing 3.18. While the ber_frame function uses a given half or full bit error rate the crc_frame function is used to inject a specified number of faults. Table 4.2 and figure 4.8 have been calculated using this function. The functions toggle_full_bit and toggle_half_bit are defined outside the crc_frame function.

```
for(int i=0; i<num_errors; i++) {
    if(modify == modify_full) {
        int pos = random() % BS_halfbit_string.size();
        toggle_full_bit(BS_halfbit_string, pos>>1);
    } else {
        int pos = random() % BS_halfbit_string.size();
        toggle_half_bit(BS_halfbit_string, pos);
    }
}
```

Listing 3.18: Error Injection in the crc_frame function

3.3.4.5 The time_frame Function

Since the time_frame function is very similar to the previously presented functions it is only shown in the appendix D.4.

3.4 Evaluation of the Simulation Results

As it has been mentioned before, the simulation results are stored and evaluated using a SQL database. This chapter contains a detailed description how the values from the simulation are stored and how they can be evaluated using SQL queries.

The tables used to store the necessary information are shown in listing 3.19 and listing 3.20. The individual attributes are defined as in table 3.2. Since most attributes are common to both tables, the description for both tables has been merged into a single table. A detailed description of the used `SQL` commands can be found in the PostgreSQL User's Guide [PSQL].

Attribute	Description
len	frame length (16, 64 or 128 bit)
receiver	receiver type (`s01`, `s02`, `d01` ... `d09`)
rcvoffs	offset of the first edge within the receivers current bit cell, i.e. -0.1 causes the first edge is receuved 0.1 `BCD` before the first data point
scale	clock drift (1.0 = no clock drift)
jitter	jitter setting
t_asym	asynchronous frame (value for falling edge)
e_asym	asynchronous edge (only a single edge is asynchronous)
offs_val	offset of part of frames (see chapter 5.5.2.1)
high	signal is forced to high for 0.8 `BCD`
low	signal is forced to low for 0.8 `BCD`
delay	the 16^{th} and 17^{th} edge are delayed (chapter 5.5.2.1)
num_frames	number of frames simulated with this settings
num_dec	number of errors detected in the sampling logic
num_xrx	number of errors detected in the xerxes decoder
num_crc	number of errors securely detectable by crc
num_gr6	number of frames with more than 6 errors
num_corr	number of correct received frames
b0n	number of frames with n bit errors

Table 3.2: Attribute definitions for the `SQL` Tables

```
CREATE TABLE res (
        len        int,                    receiver   char(3),
        rcvoffs    float8,                 scale      float8,
        jitter     float8,                 t_asym     float8,
        offs_val   float8,                 high       boolean,
        low        boolean,                delay      boolean,
        num_frames bigint,                 num_dec    bigint,
        num_xrx    bigint,                 num_crc    bigint,
        num_gr6    bigint,                 num_corr   bigint)
```

Listing 3.19: `SQL` Table Definition for Resynchronisation Simulations I

The log files generated during the simulation (see listing 3.21) are evaluated by a Perl script that populates the database with the values achieved from the simulation. This is done via the `DBI` interface which is described in the DBI manual Page [DBI]. All files that shall be filled into one table have to be imported at once, since the table is dropped and new created each time to ensure that the table is empty and no old values are polluting the results. The log files have a total size of 90 MB for the first part of the simulations and 60 MB for the second part and the parsing takes several minutes.

The queries needed to evaluate the simulation runs are shown in listing 3.22 and listing 3.24. The evaluation of the simulations of the first part using listing 3.22 lasts about one minute, evaluations of the second part need about eight seconds. Using the `select` clause

```
CREATE TABLE res_asym (
        len         int,                  receiver   char(3),
        rcvoffs     float8,               scale      float8,
        jitter      float8,               t_asym     float8,
        e_asym      float8,               num_frames bigint,
        num_dec     bigint,               num_xrx    bigint,
        num_crc     bigint,               num_gr6    bigint,
        num_corr    bigint,               b00        bigint,
        b01         bigint,               b02        bigint,
        b03         bigint,               b04        bigint,
        b05         bigint)
```

Listing 3.20: SQL Table Definition for Resynchronisation Simulations II

```
-r rcv_win=0.5:sync_win=0.5:offset=0.0 -a scale=0.98 -a jitter=0.01 -a asym:¬
r=0:f=0 -a offset=0.0 --framelen 64 -t
 len         Frames      Receiver        Xerxes            CRC     >6 errors
  64         100000             0         99677              0           323
-r rcv_win=0.25:sync_win=0.5:offset=0.0 -a scale=0.98 -a jitter=0.01 -a asym¬
:r=0:f=0 -a offset=0.0 --framelen 64 -t
 len         Frames      Receiver        Xerxes            CRC     >6 errors
  64         100000        100000             0              0             0
-r rcv_win=0.5:sync_win=0.25:gran=0.25:offset=0.0 -a scale=0.98 -a jitter¬
=0.01 -a asym:r=0:f=0 -a offset=0.0 --fram
 len         Frames      Receiver        Xerxes            CRC     >6 errors
  64         100000             0             0              0             0
-r rcv_win=0.375:sync_win=0.125:gran=0.125:offset=0.0 -a scale=0.98 -a ¬
jitter=0.01 -a asym:r=0:f=0 -a offset=0.0 --
 len         Frames      Receiver        Xerxes            CRC     >6 errors
  64         100000             0             0              0             0
```

Listing 3.21: four example Log Entries for Part II of the Resync. Simulation

```
select
    receiver                              as "receiver",
    sum(num_frames)/sum(num_frames)       as "Frames",
    sum(num_corr)/sum(num_frames)         as "corr",
    sum(num_dec)/sum(num_frames)          as "sampler",
    sum(num_xrx)/sum(num_frames)          as "decoder",
    sum(num_crc)/sum(num_frames)          as "CRC",
    sum(num_gr6)/sum(num_frames)          as ">6 err.",
    sum(num_frames)                       as "#abs"
from res
where offs_val        =   0
and   t_asym          <=  0.1
and   scale           =   1
and   high            =   false
and   delay           =   false
group by receiver;
```

Listing 3.22: SQLQuery for Resynchronisation Simulations I

the attributes, functions or aggregate functions[6] are selected. The first item that is to be displayed using the query from listing 3.23 is the receiver type and the sum of the individual

[6]SQL aggregate functions are functions like Sum, Average, Count, ...etc. which can be used to calculate totals on sets of rows

frames divided by the sum of all simulated frames (giving the percentage). The last item is the total number of simulated frames for each row. The `from` clause specifies that the values shall be taken from the `res` table. The `where` clause specifies the conditions that have to be met by the selected columns: the offset of the first received edge must be zero, the asymmetric delay has to be less or equal than 0.1, the quartz drift is zero (scale=1) the signal is not forced and there are no delayed edges. The `group by` clause takes care that not all matching rows are displayed, which would be thousands of lines, but only a single instance for each kind of sampling logic (receiver) which combines all other values by means of the aggregate functions. The second Query found in listing 3.24 works similar.

Listing 3.23 shows the results of the first query. A real benefit when using `SQL` is the ability to specify an arbitrary precision. For this application it was not necessary but if simulations are performed with a more moderate fault model the values will be smaller and could be dropped when doing floating point arithmetics.

```
 receiver |    Frames     |     corr      |    sampler    |    decoder    |      CRC      |    >6 err.    |    #abs
----------+---------------+---------------+---------------+---------------+---------------+---------------+-----------
 d01      | 1.0000000000 | 0.0244435844 | 0.1116044198 | 0.8534674774 | 0.0066440041 | 0.0038405144 | 243000000
 d02      | 1.0000000000 | 0.0415189218 | 0.1060493169 | 0.8367012840 | 0.0070425473 | 0.0086879300 | 243000000
 d03      | 1.0000000000 | 0.0389081029 | 0.1506637325 | 0.7942535597 | 0.0100161358 | 0.0061584691 | 243000000
 d04      | 1.0000000000 | 0.0175096584 | 0.8274118107 | 0.1503758560 | 0.0018186667 | 0.0028840082 | 243000000
 d05      | 1.0000000000 | 0.0365073210 | 0.1229381111 | 0.8223769630 | 0.0086237037 | 0.0095539012 | 243000000
 d06      | 1.0000000000 | 0.0080581975 | 0.9323226996 | 0.0579250123 | 0.0006561811 | 0.0010379095 | 243000000
 d07      | 1.0000000000 | 0.0380099877 | 0.1460183704 | 0.7981225514 | 0.0087349136 | 0.0091141770 | 243000000
 d08      | 1.0000000000 | 0.0596225432 | 0.0814803868 | 0.8360584074 | 0.0101110247 | 0.0127276379 | 243000000
 d09      | 1.0000000000 | 0.0256648230 | 0.7736938354 | 0.1937827654 | 0.0025299424 | 0.0043286337 | 243000000
 s01      | 1.0000000000 | 0.0234158601 | 0.1743761893 | 0.7919622099 | 0.0054637860 | 0.0047819547 | 243000000
 s02      | 1.0000000000 | 0.0040222757 | 0.9670932140 | 0.0279620905 | 0.0003680905 | 0.0005543292 | 243000000
(11 rows)
```

Listing 3.23: Example Query Results for Simulations I

```
select
    receiver                                    as "receiver",
    t_asym                                      as "asym",
    round(sum(num_corr)/sum(num_frames),10)     as "correct",
    round(sum(num_dec)/sum(num_frames),10)      as "sampler",
    round(sum(num_xrx)/sum(num_frames),10)      as "xerxes",
    round(sum(b01)/sum(num_frames),10)          as "b01",
    round(sum(b02)/sum(num_frames),10)          as "b02",
    round(sum(b03)/sum(num_frames),10)          as "b03",
    round(sum(b04)/sum(num_frames),10)          as "b04",
    round(sum(b05)/sum(num_frames),10)          as "b05",
    round(sum(num_gr6)/sum(num_frames),10)      as ">6 err.",
    sum(num_frames)                             as "#abs"
from res_asym
where e_asym = 0
group by receiver, t_asym
order by receiver, t_asym;
```

Listing 3.24: `SQL` Query for Resynchronisation Simulations II

Chapter 4

Code Coverage

This chapter describes the influence of errors on Xerxes-encoded transmissions without the influence of sampling. The analysis was performed using two different fault models with the previously described simulator. The results are presented in chapter 4.3.1 and chapter 4.3.2.

Error propagation is the effect that n transmission errors can cause $n + m$ errors in the received bit stream. Bit stuffing and other encoding schemes are vulnerable to error propagation. This is a serious problem when using CRC checksums: Using an 16 bit CRC we have a Hamming distance of 6 which is equal to $n + m$. If $m > 0$ then n will drop. The Number of errors that can be securely detected is equal to $max(n)$ for which in all cases $n + m(n) < 6$. For CAN this can be determined by:

$n = 1 \quad \rightarrow \quad m_{max} = 0 \quad \rightarrow \quad n + m < 6$

$n = 2 \quad \rightarrow \quad m_{max} = framelength - 12 \quad \rightarrow \quad n + m \nless 6$

This shows that the maximum number of securely recognisable transmission errors in CAN equals one.

The examination provides a 100% coverage of all possible combinations of transmitted code and all permutations of up to six errors. The results reveal whether error propagation can happen in two different fault models. The simulator builds a sequence of codes from given code streams and simulates 1 to 6 failures on each possible permutation of positions. The resulting code stream is decoded and compared to the original input stream. Frames with more bit errors than injected transmission errors are reported.

There are two possible solutions to find out if a special code propagates transmission errors: Find a transmission error that is propagated or, prove that there is no error that will be propagated. This can be done either by logic deduction or if that is not possible by a simulation that tests out a 100% coverage of the code. For the half bit fault model (chapter 4.1.1) a combination of logic deduction and simulation is used. For the full bit fault model (chapter 4.1.2), propagated errors can easily be found. Note that the full bit fault model was included only for completeness and might not fit well for edge triggered protocols.

4.1 Fault Model

Two different fault models are used to perform this simulation: The half bit fault model, which is similar to the fault model used in [Tran 99], and the full bit fault model, which is more suited to edge triggered protocols like Xerxes. Because of the physical layer, which determines the selection of the fault model, is still undefined, simulations with both fault models are performed.

The fault model is very simple and consists only of a flipped half bit or a flipped bit. From an edge triggered point of view, this half bit or full bit flipping causes three possible modifications on the received code:

- edges that are moved to the prior/next possible sampling point (clock point → data point or data point → clock point)
- new (pairs) of edges (which requires at least two half bit faults to produce a valid signal as defined in chapter 2.1.5 for the half bit fault model)
- lost edges (this requires even three half bit errors)

To inject faults, an arbitrary position of the bit stream is chosen and the corresponding bit (half bit fault model) or two bits (full bit fault model) are flipped. Since edges can appear in the middle of bit cells, each bit cell has to be represented by two bits in the bit stream and therefore a half bit error affects a single bit position while a full bit error changes two adjacent bit positions.

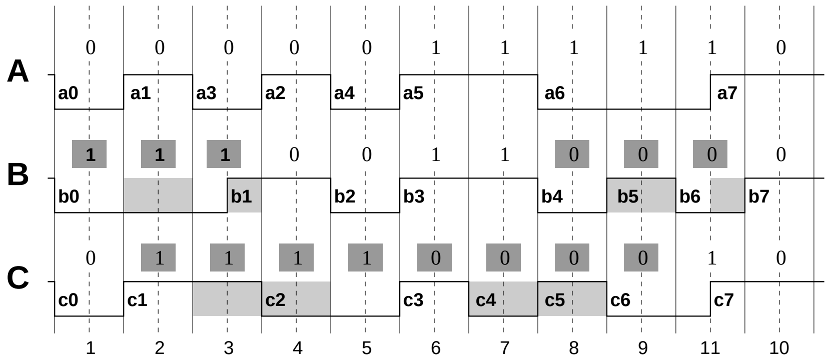

Figure 4.1: Fault Injection Example

Figure 4.1 gives an example of all possible ways of error injection. The first signal (A) is the original signal. Signals B and C are the original signal with injected half bit respectively full bit errors. Looking at signal B the second and third bit cell show two lost edges (a1 and a2) and the movement of a edge (a2→b1) to the prior sampling point (both must occur together to produce a valid signal). Next comes the introduction of a new pair of edges (b5, b6) and the required movement of a7 to the next sampling point (a7→b7). The third Signal (C) shows the loss of two edges (a2 and a3) and the movement of the following edge to the prior sampling point (a4→c2). The next injected faults moves a6 to the prior sampling point (a6→c4) and introduce two new edges (c5 and c6). The flipped half or full bits are emphasised.

4.1.1 The Half Bit Fault Model

The half bit fault model is the preferred model for this simulation. The smallest logical unit that can be distinguished by the decoder is a half bit, and therefore this unit should be used. When creating new pairs of edges by flipping a half bit they are located within one bit cell, causing a decoding error. To inject two valid edges, at least three failures are necessary as shown in figure 4.2. From the original signal (A), insert two edges (B), move one of them to the right (C) and move the following edge to the right (D) to avoid Xerxes code violations.

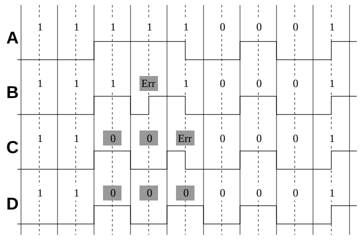

Figure 4.2: Half Bit Fault Model

4.1.2 The Full Bit Fault Model

This fault model , which was only introduced for easier comparisons to fault models used in level triggered protocols and in [Tran 99], is only included for completeness since it is not well suited for edge triggered protocols. From the original signal (A) only one error injection is necessary to move to the second signal (B), but two edges have been changed to avoid a code violation: The edge at the end of the fourth bit cell has disappeared and a new edge appeared at the beginning of the fourth bit cell. It could also be said that the edge shifted two positions to the left.

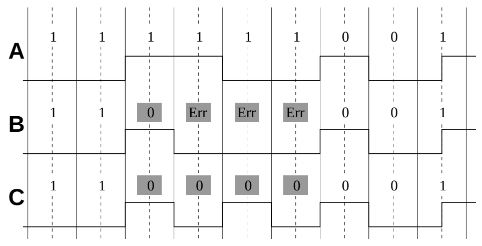

Figure 4.3: Full Bit Fault Model

As figure 4.3 shows, it is easy to find a way to change two bit cells of the received signal and get four bit errors in the decoded stream. It is not needed to run simulations for this fault model but it can be done to verify the simulator to find other combinations of possible bit errors.

4.2 Simulation Runs

The simulation was run with both fault models. Figure 4.4 shows the block diagram of the simulator. A bit pattern generator generates sequences to fully cover all possible paths of the Xerxes decoder.

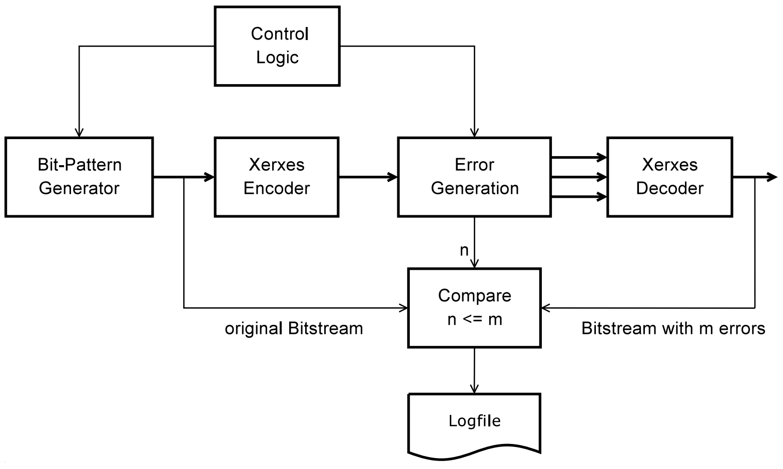

Figure 4.4: Code Coverage Mode

The goal of this simulation is to find sequences with up to six injected faults that lead to error propagation. First of all it has to be figured out how long sequences with six interdependent faults can be. This can easily be derived from the basic decoding rule introduced in chapter 2.1.6. The longest sequence defined in the basic rules is a sequence consisting of two bit cells with no edge between them (second rule). Therefore we can assume that a correctly received sequence of at least two bit cells will always be correctly decoded and that the largest possible distance between two interdependent faults may be only one bit cell.

Under this assumption the maximum code length to hold six interdependent faults is twelve. The Xerxes decoder state machine must, by definition, at least satisfy the requirements of the basic decoding rules, and therefore we can apply these findings from the basic decoding rules on the decoder state machine.

All possible transitions of the Xerxes decoder state machine are shown in figure 2.3. If an error is injected, the decoder state machine must still be traversed using valid transitions. A series of injected errors transforms a valid signal into another valid signal. There must be two valid paths through the Xerxes decoder state machine: the first path is the path that would normally be taken when no faults are injected while the second path, derived from the first one by toggling an arbitrary number of (half-) bits, must also be valid. The injected errors can be seen as a invertible function that maps two valid paths to each other. This is illustrated in figure 4.5. By toggling the first half bit of the fourth bit cell the path through the Xerxes decoder changes from "`XYZYZBX`" to "`XYZWCXD`". To find out if error propagation can occur, all branches have to be examined, whether the number of errors in the decoded stream is greater than the number of injected faults.

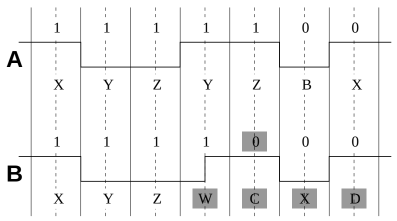

Figure 4.5: Example of different Paths through the Decoder State Machine

The procedure that achieves this is simple. First of all each possible sequences consisting of twelve bits (this makes a number of $2^{12} = 4096$) has to be found. This has to be done for each of the decoders seven states. The result are 4096 sets of twelve data bits and 24 bits representing the signal level of each half bit transmitted over the network for each of these seven states. Now the individual transmitted data can be compared to each other. Comparing each set to all others needs $n(n+1)/2$ operations, these are 8.390.656 comparisons for each state and 58.734.592 for all seven states. The comparison is performed by counting the half-bits (or full-bits) by which two transmissions differ and comparing the result to the number of differences in the transmitted data. If the number of toggled half- or full-bits is less than the number of different data bits, a case where n transmission errors causes $n+1$ errors in the transmitted stream has been found. In some cases it can happen, that the following bit cell influences the current bit cell (chapter 2.1.4). In such cases the number of injected faults may not be greater than the number of defective bits at the first correctly transmitted bit cell. The procedure for the example in figure 4.5 is shown in table 4.1. The rows "difference trans." and "difference data" show the running sum of differences of the signal and the resulting data.

4.3 Simulation Results

Simulations were performed for both fault models to find out whether the Xerxes encoding scheme is vulnerable to error propagation.

4.3.1 The Half Bit Fault Model

After a few hours the simulator finishes. No error propagation could be detected for up to six injected half bit failures.

BCD	1	2	3	4	5	6	7
half bits A	11	00	00	11	11	00	11
half bits B	11	00	00	01	11	00	11
difference trans.	0	0	0	1	1	1	1
data bits A	1	1	1	1	1	0	0
data bits B	1	1	1	1	0	0	0
difference data	0	0	0	0	1	1	1
d trans $<$ d data	f	f	f	f	f	f	f

Table 4.1: Comparing Transmission Faults and erroneous Data Bits from Figure 4.5

4.3.2 The Full Bit Fault Model

Two injected full bit errors may cause up to four decoded bits to flip. The simulation stops after a short run of about one minute with the error message shown in listing 4.1. Listing 4.1 shows an example of a possible two bit failure that results in a four bit fault in the decoded stream. The transmitted bit cells have been encoded with the symbols `N`, `C` and `D` representing no edge, an edge at the clock point and an edge at the data point.

```
ERROR: 2<3
b1:111110101000  =>  c1:DDDDDNDNDNCC
b2:111110100110  =>  c2:DDDDDNDNCCNC
```

Listing 4.1: Coverage Test Results for Full Bit Fault Model

The differences of this behaviour in contrast to the CAN multi bit error vulnerability is that this fault does not produce decoding errors of arbitrary length. The transmitted signal must be flipped in two adjacent bit cells while the number of bit cells between the two erroneous bit cells in the CAN protocol is only limited by the frame length. This condition could easily be caused by a nearby wire with a short current pulse of 200ns going through it and inducing a positive voltage for 100 ns and a negative voltage for another 100ns. A timed simulation has also been made with similar assumptions in chapter 5.7.5.

4.3.3 Remarks

This chapter contains some observations that were made during the simulations plus one additional simulation that was used to compare the results of both fault models.

4.3.3.1 Fault Model

While the different fault models lead to different results in chapter 4.3.1 and chapter 4.3.1, the next section will show a way to compare both fault models. The comparison of both fault models leads to the conclusion that both fault models result in the same error rates for the assumption that the probability for half bit errors is twice as high as the probability for full bit errors.

4.3.3.2 Frame Error Comparison

The previous chapters examined if error propagation can occur for equal or less than six injected errors. The following chapter will additionally include cases where more than six faults are injected in the examination. The difference in the fault injection is, that faults are not injected by a deterministic method but rather by a given probability. The simulation runs may give different results, depending on the used random seeds. Using different random seeds results differing by less than $\pm 10\%$[1] could be seen. All results used in this chapter have been done with a random seed of zero.

An other difference between this approach and the exhaustive procedure is that the exhaustive method uses only valid codes and compares them in the data and signal domain, so it cannot be measured how many errors would be detected by the decoder. The stochastic procedure uses also non-valid signals, therefore the error detection abilities of the Xerxes decoder can be examined only with this method.

It can be shown that the half bit fault model and the full bit fault model give similar simulation results when the errors are generated by stochastic methods: For the simulations run in this chapter a specified half bit error rate (hber) or bit error rate (ber) is assumed. This is the probability of one half bit respective one bit to flip. Simulations with bit error rates ranging from 10^{-8} to $5 * 10^{-1}$ were performed. Each simulation was run for 10^7 frames with a length of 128 bits.

The result of the half bit error simulation is shown in figure 4.6. The "`decoder`" entry denotes errors detected in the Xerxes decoder. "`<6 errors`" shows errors affecting less than six bits and "`>=6 errors`" designates frames with six or more errors. Since all simulations were run with 10^7 frames the according grid line marks the total number of frames.

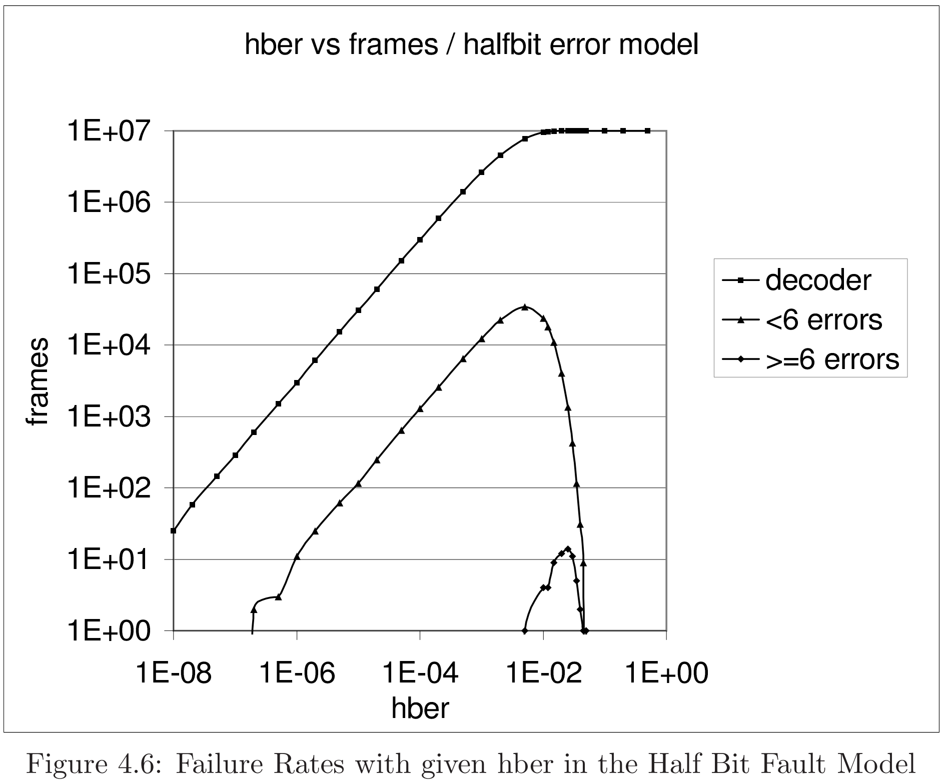

Figure 4.6: Failure Rates with given hber in the Half Bit Fault Model

The result for the same simulations using bit errors instead of half bit errors are shown in figure 4.7. Comparing figure 4.6 and figure 4.7 (the exact values can be found in appendix B) it can seen that up to a ber of $2 * 10^{-2}$ the error rates for full bit errors are about half the size as the error rates for half bit errors. This is caused by the fact that there are only 128 bit positions but 256 half bit positions in which errors can occur. Whether error propagation occurs or not, the results of these simulations is the same for both fault models: The errors found in the decoder rise linear with increasing error probabilities until each frame is fully

[1]usually the differences were below $\pm 4\%$ but with a random seed of 8 and a half bit error rate of $4.5 * 10^{-2}$ ten frames with less than six errors were observed instead of nine

corrupted at a hber of $2 * 10^{-2}$ or a ber of $5 * 10^{-2}$. Errors of up to five bits rise linear until a hber of $5 * 10^{-3}$ or a ber of 10^{-2} before dropping to zero within the duplication of the error rate. Errors resulting in six or more defective bits start at a hber or ber of $5 * 10^{-3}$, reaching the maximum at a hber/ber of about $3 * 10^{-2}$ with a probability of $1.4 * 10^{-6}$ for half bit errors and $3.5 * 10^{-6}$ for full bit errors.

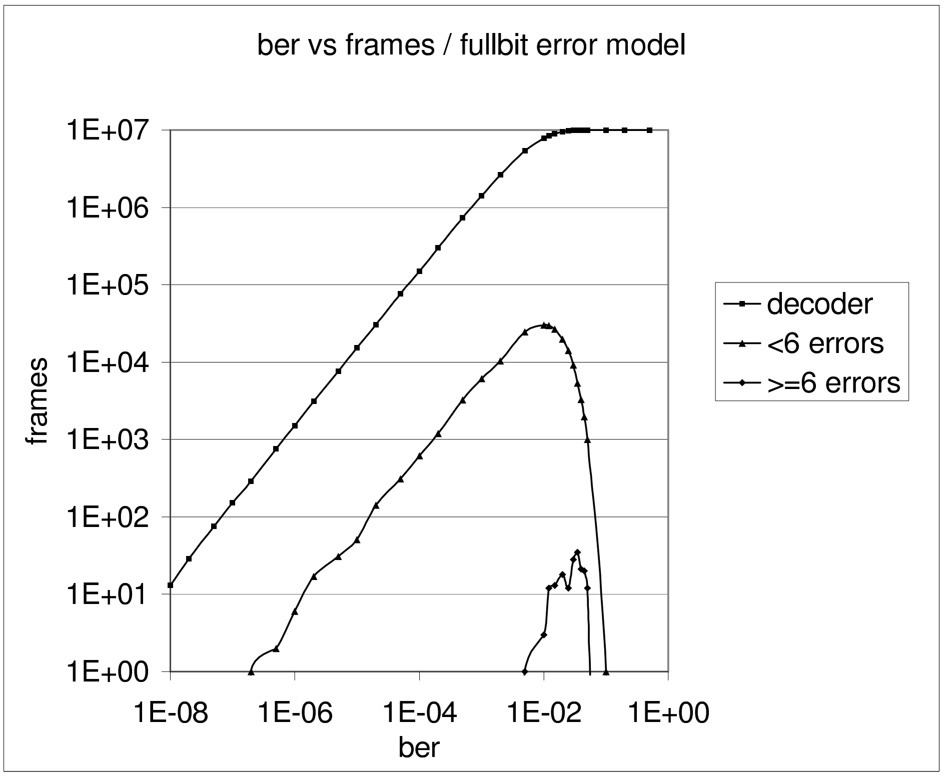

Figure 4.7: Failure Rates with given ber in the Full Bit Fault Model

4.4 Evaluation

Results show that the vulnerability to error propagation of the Xerxes encoding scheme highly depends on the fault model. In the half bit fault model, no error propagation occurs, whereas in the full bit fault model error propagation can occur.

When running simulations with stochastic error generation and a hber that is about twice the ber both fault models give the same results. The maximum probability of undetected errors is $3.5 * 10^{-6}$ for the full bit fault model and $1.4 * 10^{-6}$ for the half bit fault model at error rates of about $2 * 10^{-2}$. These values have to be multiplied by a factor of $3 * 10^{-5}$, the residual value of the CRC, to get the overall probability for undetected errors. The resulting error probabilities are $1.05 * 10^{-10}$ for the full bit fault model and $4.2 * 10^{-11}$ for the half bit fault model.

As by now only the Xerxes encoding has been considered. The FlexRay™ frame format brings a strong weakening to the Xerxes encoding. The FlexRay™ frame format defines an additional optional bit at the frame end called the *Code Violation Avoidance Bit* (`CVAB`). This is necessary because the look ahead is only one bit but the symbols "B", "C" and "D" (see chapter 2.1.2) require an ending zero, else the Xerxes code would be violated. The `CVAB` itself does not weaken the Xerxes encoding but after the `CVAB` an optional edge may be created to bring the transmission line to the recessive state. The problem with this optional edge is that there is no way for the decoder to detect if this edge is part of the encoded data or

Bit Errors	Frames	Xerxes	CRC
1	20.253.148	20.253.148	0
2	9.645.517	9.410.219	235.298
3	5.064.531	5.064.531	0
4	2.510.543	2.495.112	15.431
5	1.265.722	1.265.722	0
6	629.969	628.534	1.435
7	315.613	315.613	0
8	314.957	314.602	355
Sum	40.000.000	39.747.481	252.519

Table 4.2: Half Bit Errors recognised in the Xerxes Decoder using an End Of Frame Sequence

not. In case of an synchronisation error (like shown in chapter 5) this optional edge could be considered as part of the frame.

The optional edge directly affects one of the strengths of the Xerxes encoding scheme that ensures the detection of an odd number of half bit errors. This can be explained by the way the Xerxes encoding works: The symbols “B”, “C” and “D” (as defined in chapter 2.1.2) always have a opening and a closing zero bit. By injecting a single half bit error a ”1” would be transformed to a ”0” and vice versa. The Xerxes code would be violated at the end of the frame causing a code violation. The optional transition at the frame end introduced by the `CVAB` can be “borrowed” and used as closing zero bit and the error becomes undetectable by the Xerxes decoder. Table 4.2 shows where bit errors are detected when fixed frame end sequences without optional transitions are used. “Frames” give the number of simulated frames, “Xerxes” stands for defective frames are recognised by the Xerxes decoder and “CRC” denotes that errors that pass the Xerxes decoder and must be detected using the CRC checksum (with will of course only work with a probability of about $1.3 * 10^{-5}$).

An additional simulation run has been performed to examine the impact of the optional edge after the `CVAB`. Figure 4.8 shows the number of 128 bit sized frames with n errors that passed the Xerxes decoder undetected. 10.000.000 frames were simulated for each n from 1 to 8. While only pairs of errors pass the Xerxes decoder with an appropriate end of frame sequence like table 4.2 shows, frames with odd numbers of errors may also pass the `CVAB`-crippled Xerxes decoder with only a slightly reduces probability by “borrowing” the optional edge after the `CVAB`. This applies also to full bit errors. The probability for 6 half / full bit errors to pass the decoder undetected is equal to $1,79 * 10^{-5}$ / $1,06 * 10^{-5}$. When using a end of frame sequence an other CRC specialised on odd numbers of errors could be used, increasing the overall error detection rate.

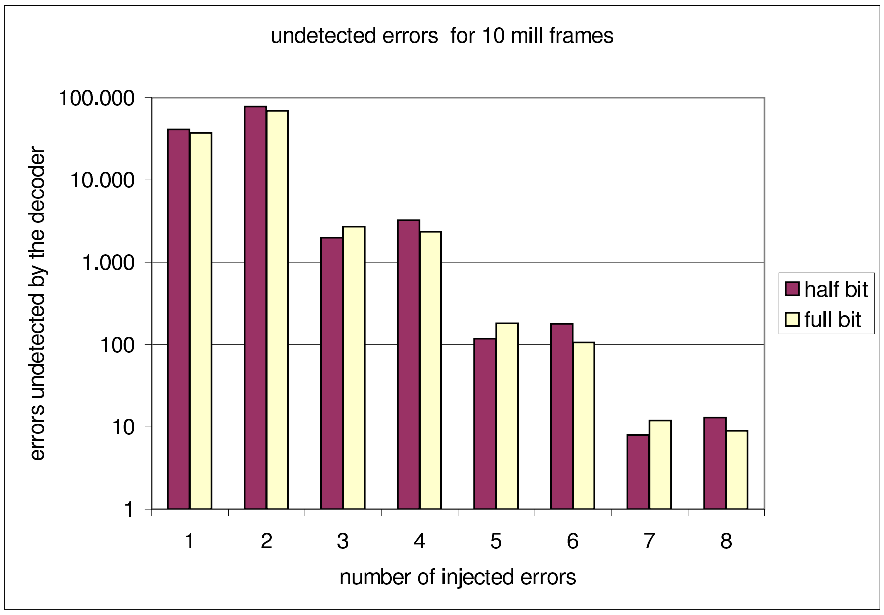

Figure 4.8: Error Detection Abilities of the Xerxes Decoder

Chapter 5

Resynchronisation

While chapter 4 covers only errors that map to half or single bit errors after sampling, this chapter examines the impact of transmission errors, properties of the physical layer (such as asymmetrical delays for falling and rising edges) and the chosen sampling method on the transmitted data stream.

The first part of this chapter describes how resynchronisation works and gives an explanation how the figures in this chapter can be read. The next part gives two basic examples, how the synchronisation can be lost. The third part describes some internals of the simulator. The fourth part describes the fault model used for this simulation. The fifth part introduces the different resynchronisation methods and simulation parameters. The next sub-chapter shows the results of the simulation runs for the different sampling methods. The last part compares the different resynchronisation methods and gives a short conclusion.

5.1 Basics

One bit cell is divided into tree different kinds of intervals as shown in table 5.1. Edges that are located in the sync interval are recognised as valid and will not cause any resynchronisation. Edges in the receive interval are recognised as valid but will cause resynchronisation and edges in the error interval will be recognised as erroneous.

Figure 5.1 is a legend for the figures used below in this chapter. In the following drawings the time axis and the signal level axis will be skipped since they do not improve readability. The diagrams show different states of the sampling logic. The line of different shaded rectangles shows how edges are interpreted by the sampler and will be referred to as synchronisation grid from now on. When resynchronisations happen the synchronisation grid will be moved to the left or to the right in multiples of the granularity parameter given to the simulator. In most cases the granularity equals the oversampling rate, since this is the smallest unit that can be distinguished by the receiver. The synchronisation interval, the receive interval and the error interval are multiples of the granularity.

For each resynchronisation that happens, a new resynchronisation grid is drawn to visualize the behaviour of the sampler during resynchronisation.

Interval	Edge valid	Resynchronisation
sync interval	Yes	No
receive interval	Yes	Yes
error interval	No	n.a.

Table 5.1: Different Oversampling Intervals

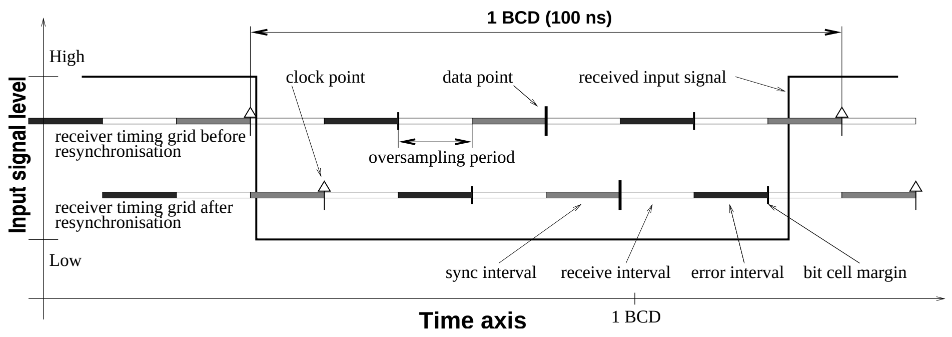

Figure 5.1: Sampling Diagram

5.2 Case Study of Synchronisation Loss

From chapter 4 we know that Xerxes is stable in respect to single synchronisation errors (one edge is moved for $\frac{1}{2}$ `BCD`). Delaying the whole signal for $\frac{1}{2}$ `BCD` or moving the synchronisation grid for the same amount will cause errors that can affect more than five bits and cannot be securely detected. This chapter contains a short overview to show, how synchronisation can be lost due to false resynchronisation.

This can easily be done without any simulations. The resynchronisation algorithm for this example uses a receive interval of $\frac{1}{2}$ `BCD` and a sync interval of $\frac{1}{4}$ `BCD` with a granularity of 0.125 (= 8 times oversampling).

Figure 5.2 shows a false resynchronisation due to an injected pulse. The original signal (black line and dotted line) contains edges at the clock point, a bit cell with no edge and an edge at the data point. This is the ending of a sequence of type C (odd number of "1"s enclosed in a pair of "0"s). The grey line shows an injected pulse. At point 1 the received edge of the pulse is out of sync interval and resynchronisation has to be done. The synchronisation grid is shifted to the left for one oversampling unit. The falling edge of the injected pulse is also not in the receive interval and the synchronisation grid is shifted to the left again. Then the next edge of the original signal is received, perfectly in time. Unfortunately the synchronisation grid has been moved to fit the erroneous pulse and again, resynchronisation is done. The synchronisation grid has been moved for $\frac{3}{8}$ `BCD` to the left. Clock and data point have been exchanged.

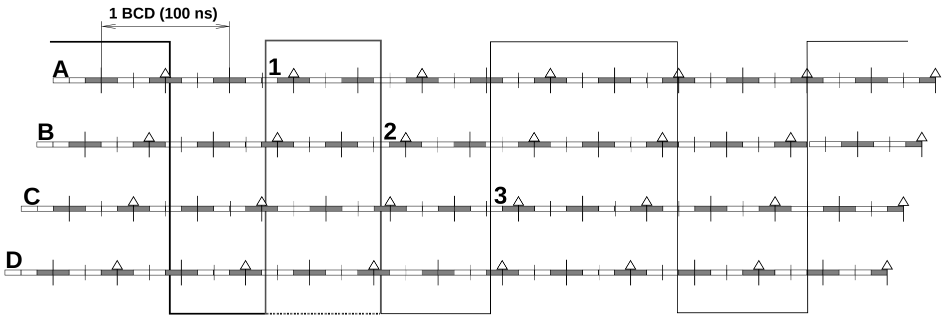

Figure 5.2: Resynchronisation Failure through injected Pulse

Figure 5.2 shows how resynchronisation failures can happen with two delayed edges. The second edge is delayed for 12.5 ns and causes resynchronisation at point 1. The third edge is delayed for approximately 25 ns and causes resynchronisation to happen again. With the fourth edge, that comes in time, the third resynchronisation happens and the synchronisation grid has been displaced by $\frac{1}{2}$ `BCD`.

For other types of sampling logic, the failures that lead to false resynchronisation are basically the same. However, both error types could be prevented by allowing only a specified

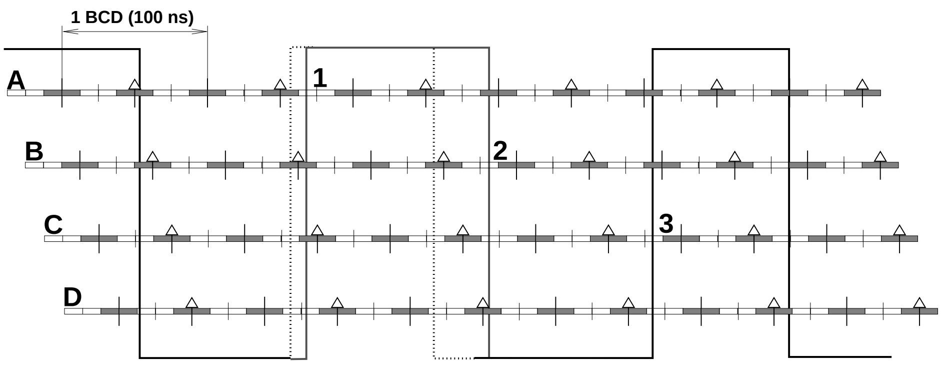

Figure 5.3: Resynchronisation Failure due to two delayed Edges

number of resynchronisations to take place during a single frame. Using static synchronisation is not possible: The last bit in a frame with 256 bits length could be displaced by $256 * 2 * 3000 * 10^{-6}$ or 1.53 BCD units[1]. The maximum error of the last edge that could be tolerated is 0.25 BCD (depending on the used sampling parameters).

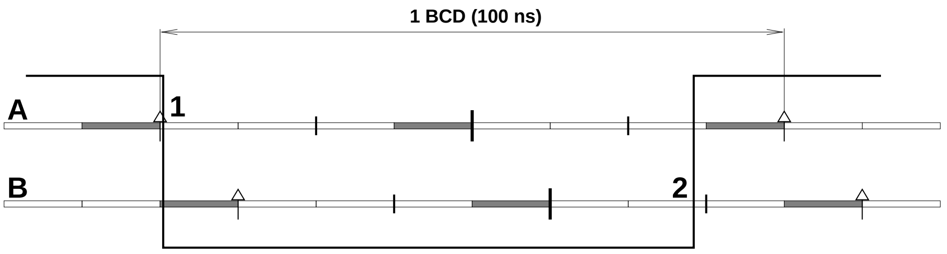

Figure 5.4: Synchronisation loss due to asymmetric Delay

Figure 5.4 shows an asymmetric signal. All falling edges are delayed for 0.1 BCD. The edge has, however, moved out of the sync-interval in point 1. The sampling grid was moved for one oversampling unit to bring the edge back to the synchronisation window. The duration of one oversampling unit is, assuming eight times oversampling, 12.5 ns. The time offset caused by the asymmetric transceiver delay is 10ns. If the rising edge comes early for more than 2.5ns the synchronisation can be lost. This is a case where the jitter described in the next chapter has influence on the simulation results.

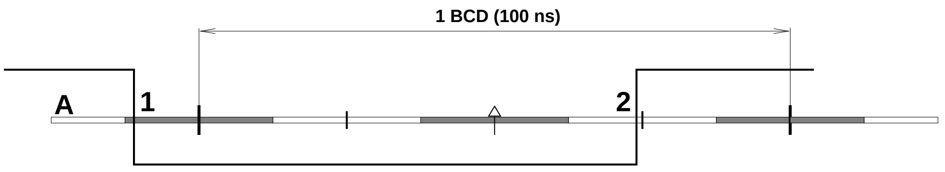

Figure 5.5: Synchronisation lost on second Edge

Figure 5.5 shows an error that can occur on the first two edges of the signal. Edge 1 is the first edge in the signal that has an asymmetric delay of 0.1 BCD (10ns). The first edge is received 10ns before the clock point. Both time offsets cause edge 2 to be received 20 ns before the next clock point. If there is some other distortion that causes an additional delay of edge 1 or makes edge 2 come early then edge 2 could move to the previous data point. The sum of both distortions has to be more than 5ns. This can also be caused through jitter.

[1]this is a requirement set by the FlexRay™ protocol working group: the quartz of the FlexRay™ controller may be off by 1.500 ppm in normal operation, leading to a worst case quartz drift of 3.000 ppm for two controllers

5.3 Simulation Details

Chapter 5.2 showed that the synchronisation can be lost by error. Neither the sampling logic, nor the Xerxes decoder nor the CRC can reliably detect this. This matter of fact does not have to be proven by simulation, rather the main focus of the following simulations is to find out how good different sampling logics work with Xerxes encoding.

This examination is done by simulating a huge number of frames under different conditions. When performing timed simulations, no CRC is used, increasing performance. CRC checks are also not very meaningful since the residual error value of the CRC is known. A better solution is to categorise frames with up to five bit errors in the decoded streams and frames with six or more bit errors as two classes of erroneous frames. The first class is always detected by the CRC the second is only detected with the remaining error probability of the used CRC.

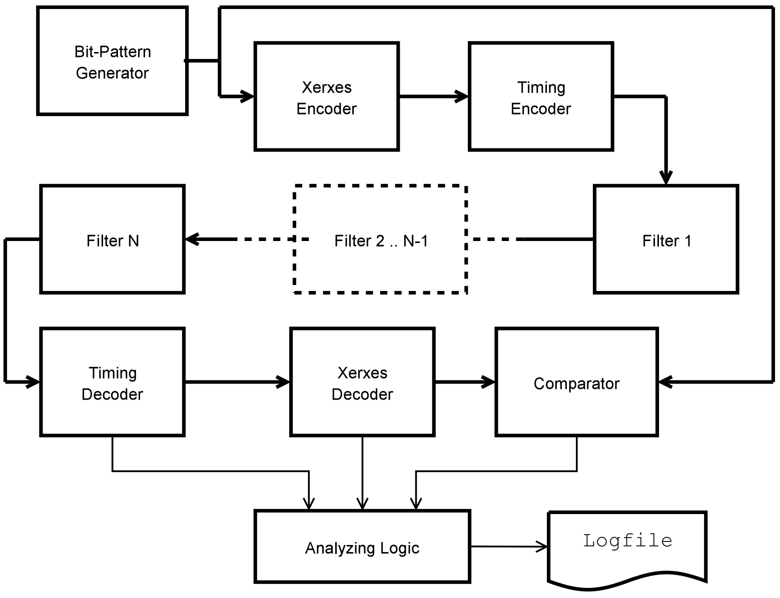

Figure 5.6: Timed Simulation Mode

Figure 5.6 shows the block diagram of the simulator when running in timed mode. Filters and sampling parameters can be specified according to chapter 3.2.

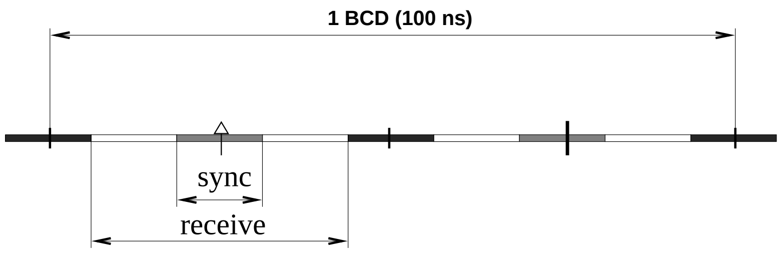

Figure 5.7: Receiver Parameters

Figure 5.7 shows the receiver parameters that can be specified on the command line. The receive interval is always located symmetrically around the sampling point. The synchronisation interval may also be located symmetrically around the sampling point (positive value) or left of the sampling point (negative value). This may be necessary for eight times oversampling with no error interval, when the synchronisation cannot be in the centre of the receive interval. The granularity parameter specifies the step size for resynchronisation. When performing resynchronisations the synchronisation grid can only be offset by multiples of this value. The parameters for the receiver shown in Figure 5.7 are snyc interval 0.125, receive interval 0.375 and granularity 0.125.

5.4 Fault Model

The fault model introduced in this chapter does not only include faults but also some features like asymmetrical delay or clock drift between two oscillators. The faults that can be injected with the simulator are:

Shift: add or subtract a custom time displacement to one or more edges. It is unlikely that whole parts of frames are delayed/early since this kind of failure can hardly be explained with the transmission media. It is however possible, that individual edges or small groups of subsequent edges are shifted and the synchronisation is lost. This filter can be used to examine this behaviour.

Scale: Scale: This filter simulates a slow clock drift between two controllers. While this is not an error condition it has also impact on the bit sampling and has to be included in the simulations.

Jitter: In applications under harsh environmental conditions such as in bonnets there is a lot of noise and signals may be delayed or be too early by minimal amounts of time. This filter can be used to make the simulation more realistic, since it adds randomly small time offsets to each edge. These offsets are normal distributed and the maximum time offset may be given. It is advised to use this filter in every simulation.

Asym: Asymmetric delays between falling and rising edges are a transceiver property and not a fault but have heavy impact on bit sampling. A simulation that does not consider asymmetric delays would not be realistic.

High: Pull the transmission line up to high for a given amount of time.

Low: Pull the transmission line down to low for a given amount of time.

Other faults may be generated by combinations of these failures. Periodic failures generated by a 1MHz signals simply consist of failures at `BCD` 0,10,.,130,140 and can be modelled by applying a series of filters in sequence. Asymmetric delays of single edges can be modelled using the Shift filter and selecting specified edges. Odd edges are always falling, even edges rising.

5.5 Simulation Parameters

This chapter describes the used receiver types and the injected faults. All simulations were run with frame lengths of 16, 64 and 128 bits. Each combination of errors from a special chosen set of errors was simulated with each receiver type, even in cases where it was expected that all errors would be caught by the receiver logic. This was done to have the same simulations for each receiver allowing easier comparison and to verify if the assumptions made about the receiver logic were correct.

5.5.1 Receiver Types

This chapter provides a brief description about the sampling methods used for the simulations. Table 5.2 contains a list of the different sampling logics used for the simulations, figure 5.8 shows a graphical representation of table 5.2. The sampling logics S1 and S2 have the sync interval set to 0.5 and no resynchronisation will happen (static synchronisation). In S2 the receive interval is only 0.25 `BCD` units wide. Edges received not within this interval

receiver	sync interval	receive interval	granularity (=1/overs)	oversampling rate	asymmetric
S1	0.500	0.500	0.125	8	No
S2	0.500	0.250	0.125	8	No
D1	0.250	0.500	0.250	4	No
D2	0.250	0.500	0.125	8	No
D3	−0.125	0.500	0.125	8	No
D4	0.125	0.375	0.125	8	No
D5	0.100	0.500	0.100	10	No
D6	0.100	0.300	0.100	10	No
D7	0.300	0.500	0.100	10	No
D8	0.250	0.500	0.125	8	Yes
D9	0.125	0.375	0.125	8	Yes

Table 5.2: Different bit Samplers used in Simulations

will cause errors detected by the receiver. D1 uses four times oversampling to examine the impact of low oversampling rates. The difference between D1 and D2 is the granularity value. The synchronisation grid in D1 can only be shifted by multiples of 0.25 `BCD`. When using D2 it can be shifted by multiples of 0.125 `BCD`. D2 and D8 use an oversampling factor of eight and a sync interval of 0.25. The difference between D2 and D8 is that D8 has compensation for asynchronous transceivers. After each falling edge the synchronisation grid is shifted to the left for a specified number of granularity units and after each rising edge the synchronisation grid is shifted back to the right. The advantage of this resynchronisation method is that the received edges are always in the same oversampling interval. Normally with eight times oversampling and an asynchronous delay of more than 12.5 ns the edges are always received in different oversampling intervals. By compensating these asynchronous delays resynchronisation takes place only when it is really necessary (i.e. clock drift). D3 has a smaller synchronisation window than D2. D4 and D9 use eight times oversampling with an error interval. The assumption for D4 is that it will fail for asynchronous delays more than 12.5 ns (one oversampling interval). D5, D6 and D7 use ten times oversampling to examine if higher oversampling rates bring any advantages.

All simulations were run with an offset of 0.0 and ±0.1 (±0.09) `BCD` units on the first edge. The smaller values were used for receivers with 10 times oversampling. This value can be given with the receiver parameters and shall simulate the fact that the edge will lie somewhere within the oversampling interval and not exactly on a sampling point.

5.5.2 Injected Faults

The simulation is split into two parts. The first part examines the overall error behaviour of the different sampling methods by applying a combination of different faults on the physical layer. The second part applies also a combination of different faults but the focus is set on asymmetric transceiver delay.

5.5.2.1 Overall Error Behaviour

The faults injected in the first part of the simulation are composed by each possible combination of the following faults (or features of the physical layer):

- clock drifts: ±10.000 ppm, <u>±1.000 ppm</u>, <u>0 ppm</u>
- jitters: <u>0.01</u>, <u>0.05</u> `BCD`

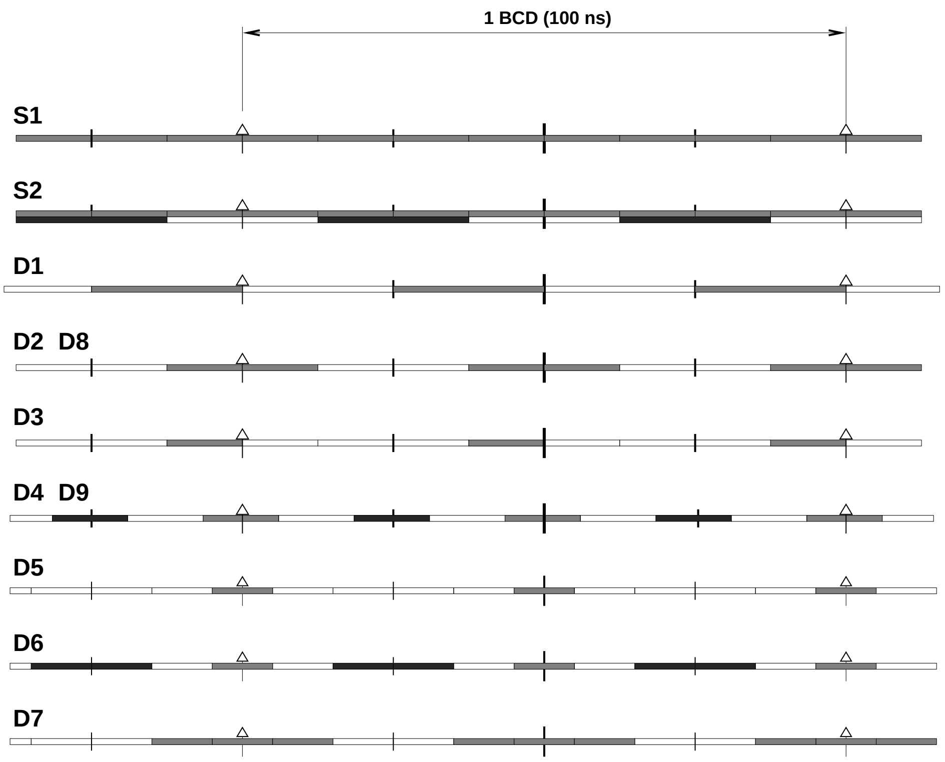

Figure 5.8: Different bit Samplers used in Simulations

- asymmetric delays: <u>0</u>, <u>0.05</u>, <u>0.1</u>, 0.2 and 0.3 BCD
- offsets of parts of frames: <u>0</u>, 0.3, 0.5, 1.0 and 1.5 BCD offset starting at BCD 12
- high: signal forced to high for a duration of 0.8 BCD at BCD 18
- high/low: signal forced to high for 0.8 BCD at BCD 18.2 and to low for 0.8 BCD at BCD 19
- delayed edges: delay for 0.12 BCD at 16th and for 0.24 BCD at 17th edge.

All simulations were run with frame lengths of 64, 128 and 256 bits and 10.000 frames. In total 267.300 simulation runs were performed with 2.673.000.000 simulated frames.

Combinations of the underlined values are used in every simulation and can be interpreted as “background noise”. All of this faults should be tolerated during normal operation. In some special cases (i.e. `scale=1.00`) some of this faults are obviously not present.

5.5.2.2 Asymmetric Delays

The fault injected in the second part consisted of all combinations of the following faults:

- clock drifts: ±10.000 ppm, 0 ppm
- jitter: 0.05
- asymmetric delays of single edges: 0.0, 0.1, 0.2, 0.3, 0.4 and 0.5 BCD
- asymmetric delays of full frames: 0.0, 0.1, 0.2, 0.3, 0.4 and 0.5 BCD

The simulations run for the second part were performed with frame lengths of 64, 128 and 256 bits and 500.000 frames per simulation. A total of 3.266 simulations runs with 1.633.000.000 simulated frames were performed.

5.6 Simulation Runs

The two following chapters contain the results of the performed simulation runs. In the next chapter the errors are classified into sampling errors, decoder errors, errors with up to five faulty bits and errors with six or more erroneous bits. In the following chapter the same error classification is used as in the chapter before, in addition the number of errors for up to six bit errors are counted.

5.7 Fault Simulations I

This sub-chapter covers the first part of the simulations described above. Combinations of the faults introduced in chapter 5.5.2 were used in each simulation, the other faults were only simulated when examining a specific fault type. Some combination of faults that are tolerated when appearing solitary can also cause errors. This situation is described in chapter 5.2, figure 5.4 and figure 5.5. In real applications the requirements for the physical layer must be set strictly enough to prohibit these error cases.

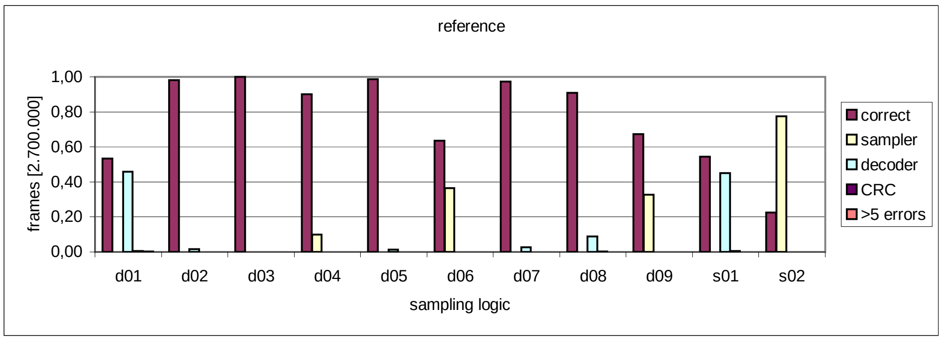

Figure 5.9: The Reverence Diagram

Figure 5.9 shows the reference simulation results. These results are derived from simulations of all possible permutations of the faults emphasised in chapter 5.5.2.1. The y-axis shows the normalised number of frames. The total number of frames can be found on the labelling of the y- axis. The x-axis shows the different sampling logic types. Each sampling logic contains five categories: correct frames (“`correct`”), frames with errors detected in the sampling logic (“`sampler`”), frames with errors detected in the Xerxes decoder (“`decoder`”), frames with errors detected by the CRC value (“`CRC`”) and frames with more than five errors (“`>5 errors`”). These values are complementary can be added up to 1, which represents the total number of frames. Since small values cannot be shown in the diagrams, all values used for the diagrams of this chapter can be found in appendix C.

5.7.1 Clock Drift (Time Scale)

Figure 5.10 to figure 5.12 show the impact of clock drifts on the transmitted frames. This simulation is also called scale because small clock drifts can be modelled by scaling the time axis of the transmitted data. A scale of 1.01 can be used to model a clock drift of 1% with the receiver running faster than in the transmitter.

Figure 5.10 shows a simulation run with no clock drift. In figure 5.11 the receiver runs 10.000 ppm faster than the transmitter and in figure 5.12 the receiver is 10.000 ppm slower than the transmitter. Note that figure 5.10 differs from the simulation reference because the scale is set to `1.000` so the impact of the clock drifts of ±1.000 ppm are not included.

It is obvious that figure 5.11 and figure 5.12 differ in some points: d01 can handle only senders with slower or equal clocks. This is due to the fact that the sync interval is aligned

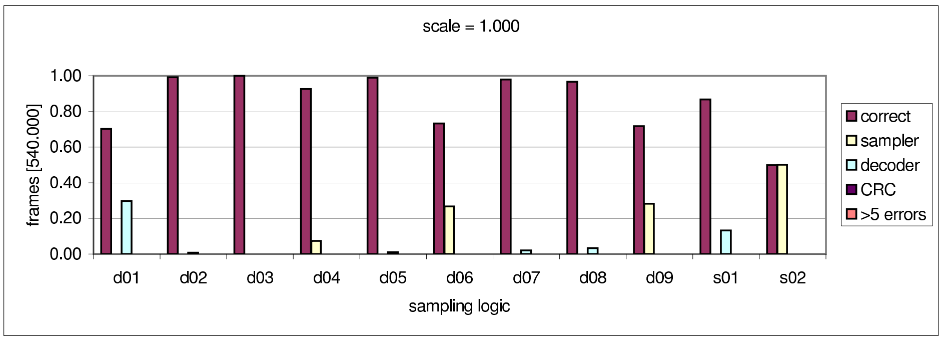

Figure 5.10: No Clock Drift between Sender and Receiver

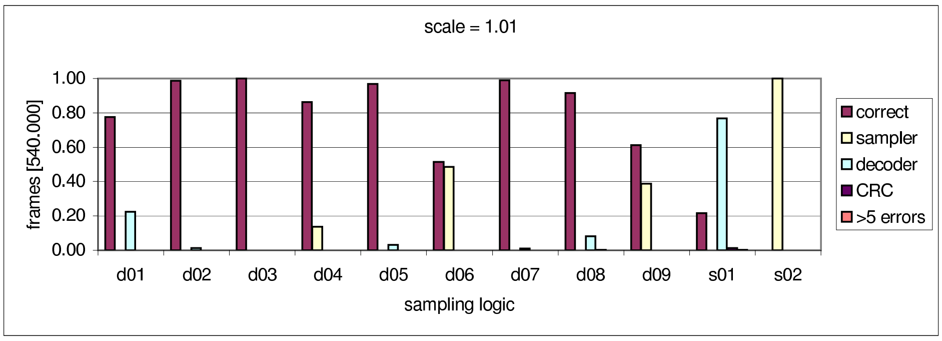

Figure 5.11: Receiver Clock faster than Sender Clock

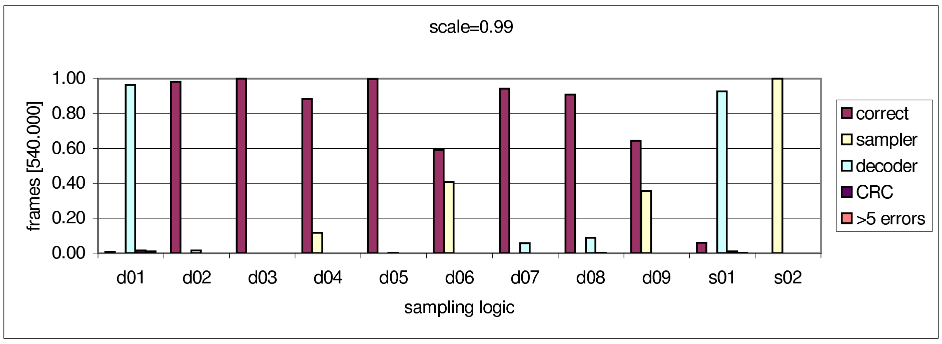

Figure 5.12: Receiver Clock slower than Sender Clock

with the bit cell border and edges that are out of this interval on the left side will cause the sampling logic to fail. s01 and s02 perform, as expected, very weak in these simulations. Since s01 has no error window, slow clock drifts cannot be detected by the sampling logic and can cause frames with more than five bit errors as figure 5.11 and figure 5.12 show. It can also be seen that the results with scale 0.99 are mostly better than the results with scale 1.01. It is also interesting that D3 performs better in this simulation than D2. Exact values from the simulation runs can be found in appendix C.

5.7.2 Jitter

Simulations have shown that all sampling logics are relatively insensitive to this fault type. While the jitter of 0.01 `BCD` was only introduced to add numerical noise to the simulation timing, the jitter of 0.05 `BCD` was expected to change simulation results significantly. In some cases, like in the example shown in chapter 5.2, the jitter may change the results for several frames but in general the impact on the simulation results is small. Receiver logics with an oversampling rate of 10 or an error window (d05-d07) are more sensible to jitter sizes of 0.05 BCD.

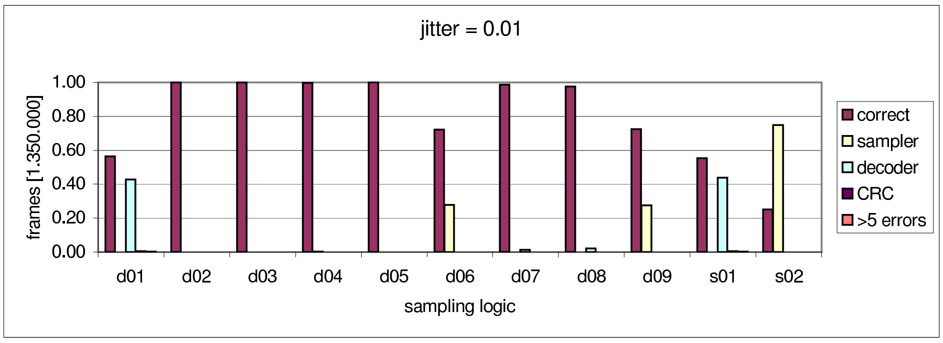

Figure 5.13: Frames with a 0.01 BCD Jitter

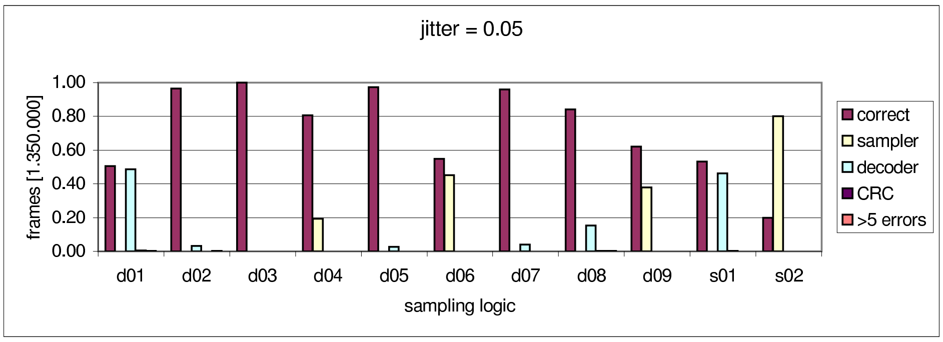

Figure 5.14: Frames with a 0.05 BCD Jitter

5.7.3 Asymmetric Delays

In this part of the simulation runs the asymmetric delay was only introduced to examine how other fault types behave in combination with asymmetric delays. Since asymmetric delays have a heavy impact on the simulation results they are examined more detailed in chapter 5.8.

5.7.4 Multiple Delayed/Early Edges (Offset)

Figure 5.15 and figure 5.16 show the results for two different groups of simulation runs. For figure 5.15 the simulation was run without delay (offset) of groups of edges so figure 5.15 shows the simulation result under normal conditions as defined in chapter 5.5.2.1. Figure 5.16 shows the simulation result when a delay of 0.5 BCD on the 12th and each subsequent edge is applied. All other simulation parameters are the same in both groups.

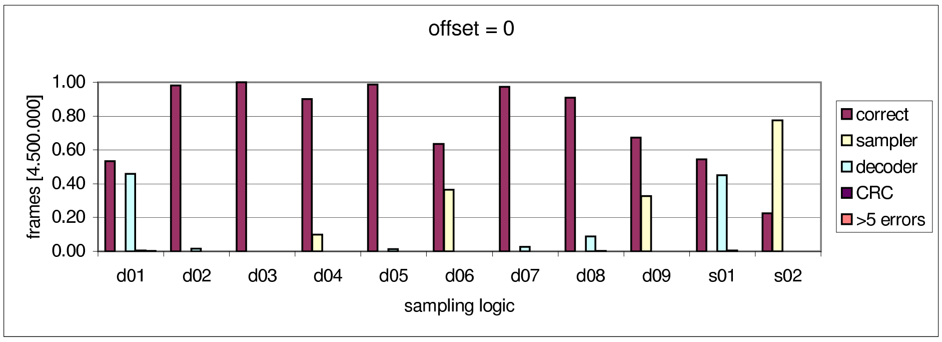

Figure 5.15: Sample logic behaviour with offset=0

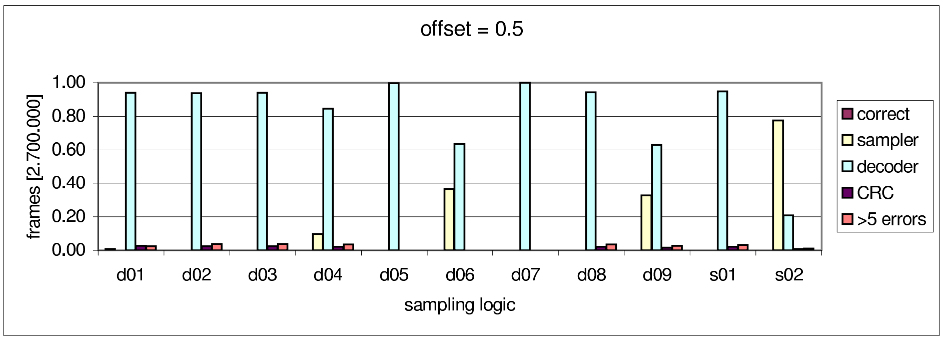

Figure 5.16: Sample logic behaviour with offset=0.5

Figure 5.16 shows that the number of correctly transmitted frames drops to zero in this case. In this special case, when the offset is a multiple of 0.5, the sampling logic cannot detect the error. From figure 5.16 we see that most of the errors are detected in the Xerxes decoder. A small part of the errors (about 3%) remain undetected. In other cases, when the offset value is not a multiple of 0.5 the sampling logic would also detect some errors. Delays larger than 1.5 `BCD` will always be detected by the Xerxes decoder. When all edges come too early by an uniform amount of time, the simulations give similar values.

This simulation was performed even if sudden offsets of large parts of the frame are very unlikely to happen. This type of fault can not be caused by the transmission line but it could be caused by an insulate contact of the oscillator, either on the sending or on the receiving controller. This fault is very hard to detect.

5.7.5 Signal forced to High/Low

This simulation injects a pulse that could flip two subsequent bits, which will produce a distortion similar to the fault that caused error propagation in chapter 4.3.2. The first simulation forces a high pulse on the physical line, the second simulation forces a high pulse followed by a low pulse. Since both results are very similar only the result the first simulation is shown here.

As expected, based on the findings of chapter 4.3.2, errors with more than five distorted bits can occur. Figure 5.17 shows the detected errors under the conditions defined in chapter 5.5.2.1. Figure 5.18 shows the detected errors after forcing the signal to high for 0.8 `BCD` in the 15th bit cell. While most of the other sampling logics let the erroneous sampled stream pass to the decoder d04, d06, d09 and s02 are able to detect the error, at the cost of more dropped frames under normal conditions. The rate of correctly received frames would drop further when the error injection would ensure that the signal is really modified instead of just forcing it to high, in fact this seems to be the explanation that the number of correct frames is rather high. In real environments the superposition of glitches is generated randomly too, so this solution seems acceptable.

5.7.6 Two subsequent delayed Edges

Two delayed edges can also lead to synchronisation loss, at least with eight times oversampling. This simulation checks the sensitivity of various sampling logics in respect to two delayed edges. The delay of the edges is 0.12 `BCD` and 0.24 `BCD`. This should cause the synchronisation logic to fail in most cases.

Comparing the reference histogram figure 5.19 to the simulation with two delayed edges in figure 5.20 it can be seen that the sampling logic does not detect this failure and there is

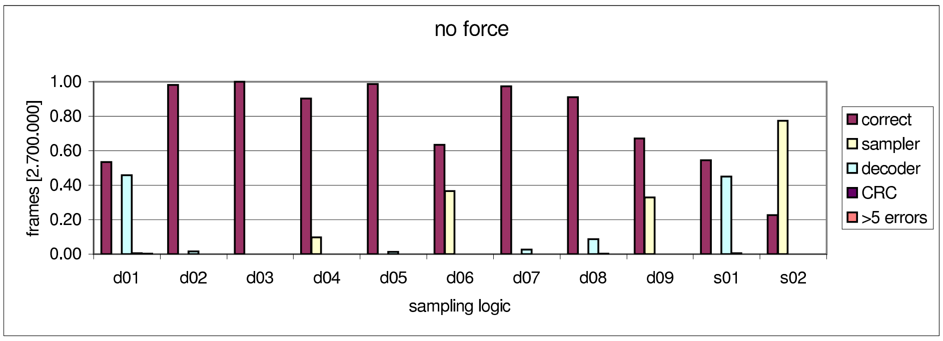

Figure 5.17: No Disturbance on transmitted Signal

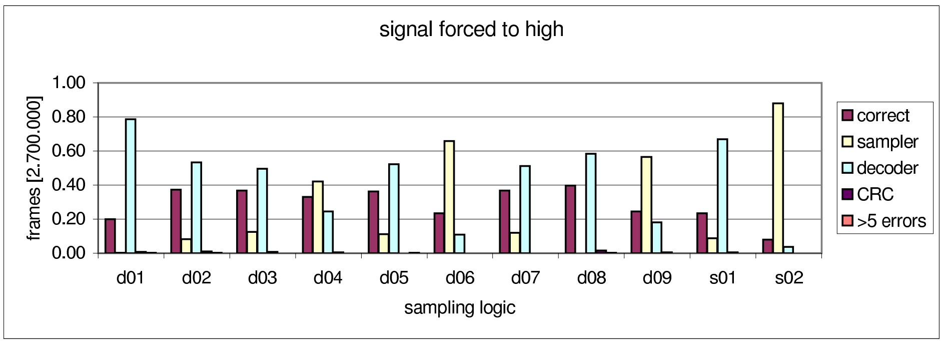

Figure 5.18: transmitted Signal forced to high for 0.8 `BCD`

a high number of errors that are detected in the decoder. The number of CRC errors and undetectable errors is also very high. Detailed numbers can be found in appendix C.

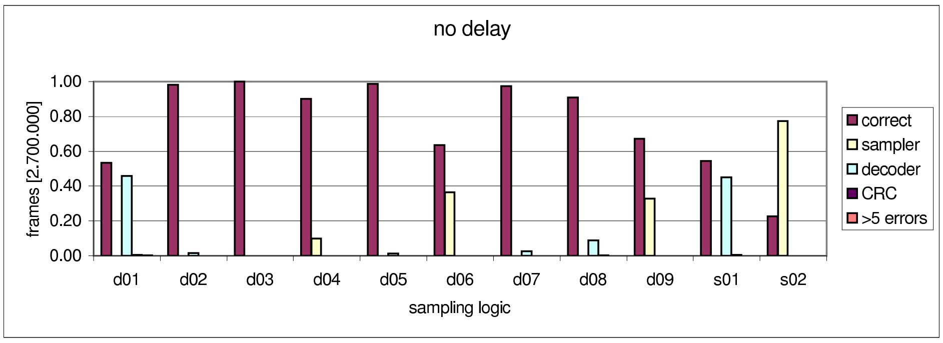

Figure 5.19: No delayed Edges

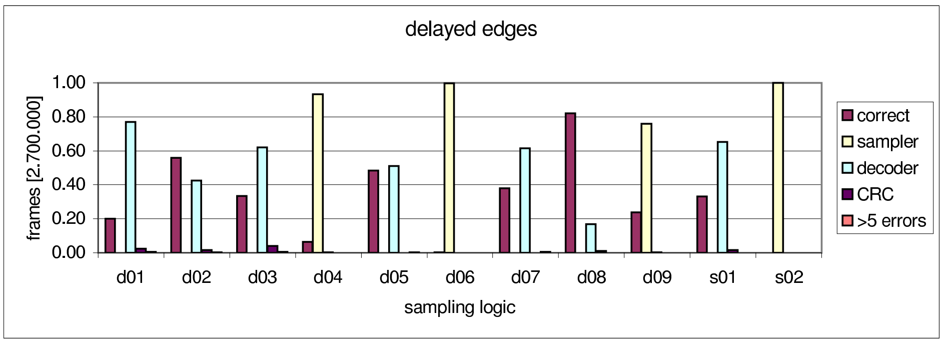

Figure 5.20: Two subsequent Edges delayed

5.8 Fault Simulations II

The simulation that were performed for chapter 5.7 have soon shown that Xerxes is very sensitive to asymmetric delays, in fact, asymmetric delays of edges or full frames are the main reason for erroneous resynchronisation. Therefore the impact of asymmetric delays will be examined in this chapter very detailed.

5.8.1 Single asymmetric delayed Edge

Asymmetric delays of single edges may, according to the half bit fault model from chapter 4.1.1, cause only one erroneous bit. This assumption is only right in a non-timed simulation. In timed simulations and under the consideration of sampling one asymmetric delayed edge could cause false resynchronisation, affecting the whole frame.

Figure 5.21 to figure 5.31 show the results of the simulation runs for the different sampling logics. The y-axis shows the number of frames, normalised to the absolute number of frames simulated. The x-axis shows where the error is detected. "`co`" designates the correct frames, "`sa`" the errors detected in the sampling logic, "`xe`" denotes errors detected in the decoder and "`1`" to "`>=6`" show the number of bit errors in the decoded message. The results for one particular delay are complementary and can be added up to 1. The detailed results can be found in appendix C since very small values are not shown on the histograms.

The number of correct frames using d01 and d02 drops to 50% at an asymmetry of 0.2, d03, d05 and d07 achieve higher frame rates and d04 and d06 provide lower correct frames. Receiver logic d08 accomplishes generally less correct frames but does not suffer from asymmetric delayed edges. The number of frames with six or more errors is also very high with sampling logic d08. d09 recognises only erroneous frames in this simulation.

It can also be seen from the simulation results, that error intervals can improve the recognition of erroneous frames significantly. Many errors can be detected in the sampling logic with the receivers d04, d06 and d09.

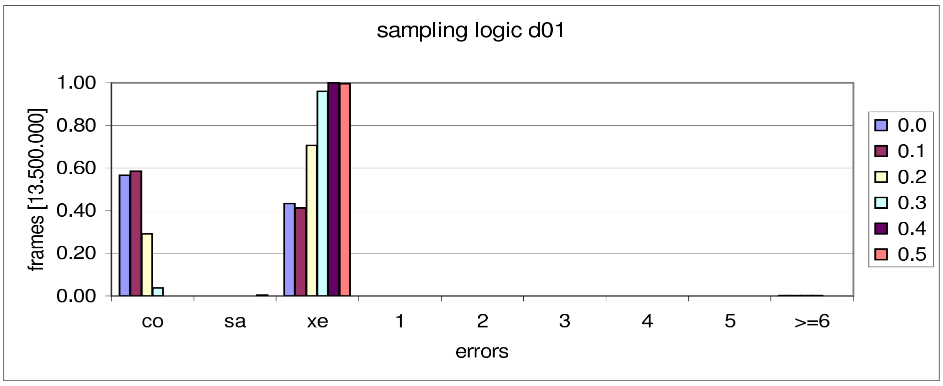

Figure 5.21: Asymmetric delayed Edge, Sampling Logic d01

Figure 5.32 shows the probabilities for more than six undetected defective bits caused by a single asymmetric edge. The x-axis shows the different sampling logics, the y-axis designates the probabilities with the residual error probabilities of the CRC ($3 * 10^{-5}$) already included and the series shown in the legend show the asymmetric delay.

5.8.2 Asymmetric delayed Frame

Asymmetric delayed frames can occur more likely than single asymmetric delayed edges. This is not only a fault but also a feature of some transceivers.

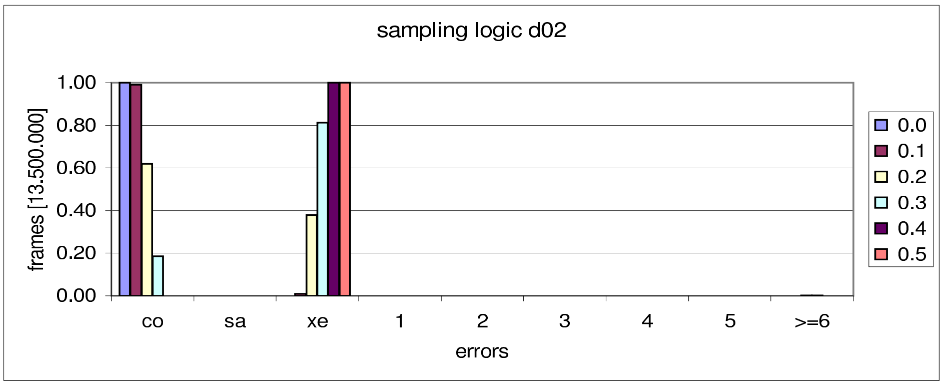

Figure 5.22: Asymmetric delayed Edge, Sampling Logic d02

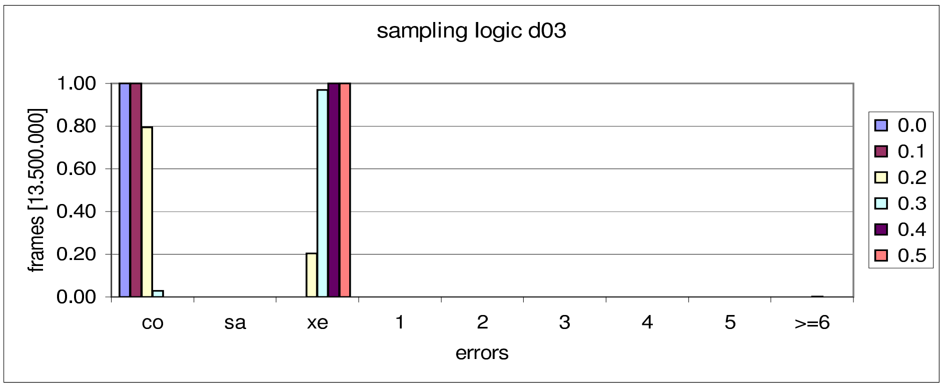

Figure 5.23: Asymmetric delayed Edge, Sampling Logic d03

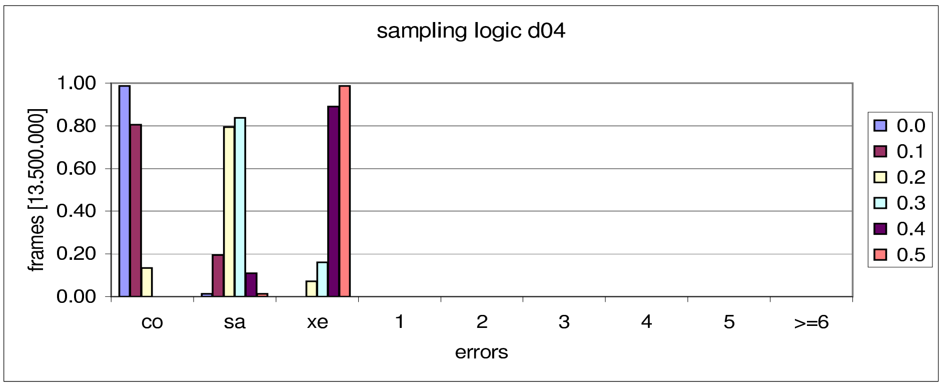

Figure 5.24: Asymmetric delayed Edge, Sampling Logic d04

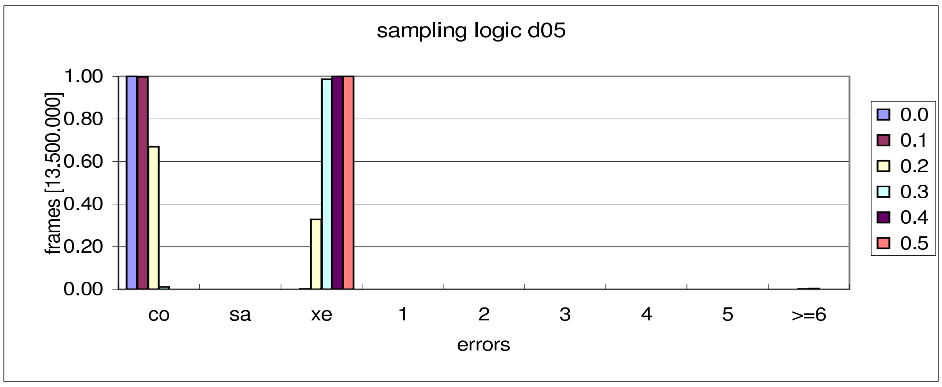

Figure 5.25: Asymmetric delayed Edge, Sampling Logic d05

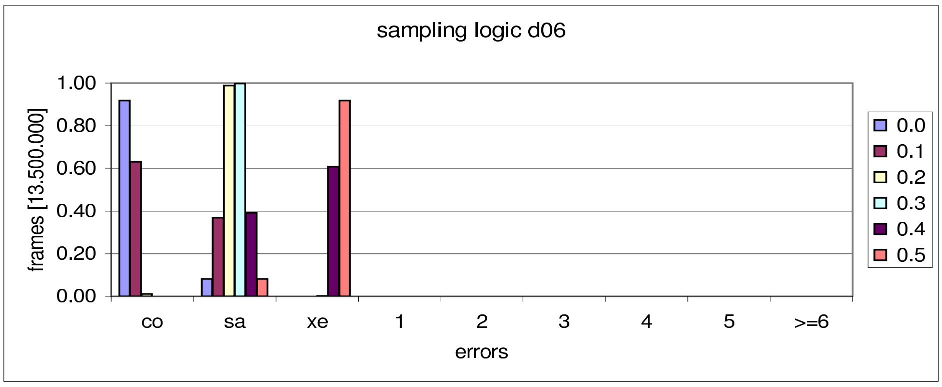

Figure 5.26: Asymmetric delayed Edge, Sampling Logic d06

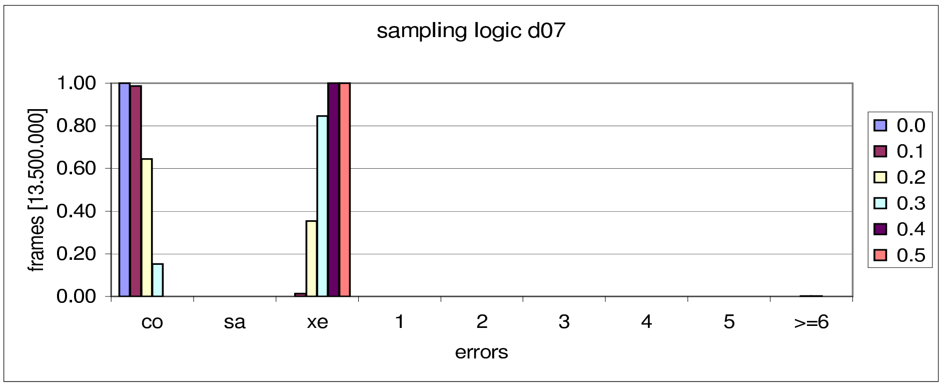

Figure 5.27: Asymmetric delayed Edge, Sampling Logic d07

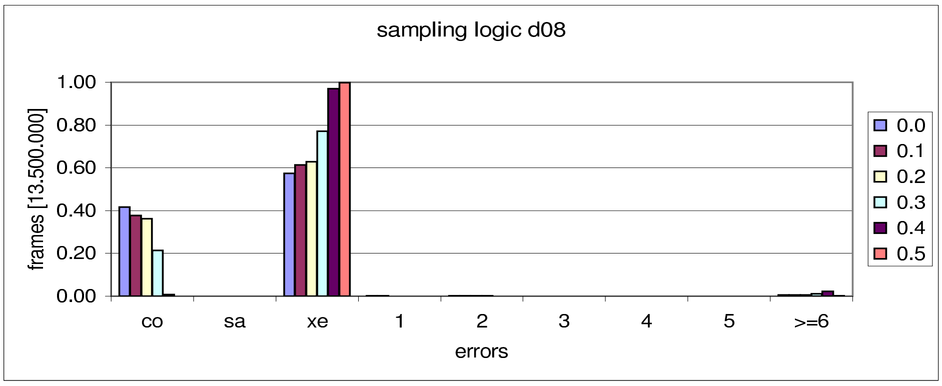

Figure 5.28: Asymmetric delayed Edge, Sampling Logic d08

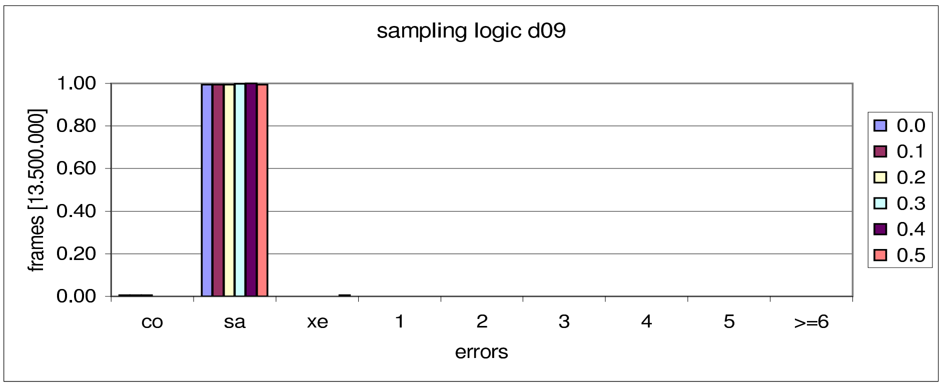

Figure 5.29: Asymmetric delayed Edge, Sampling Logic d09

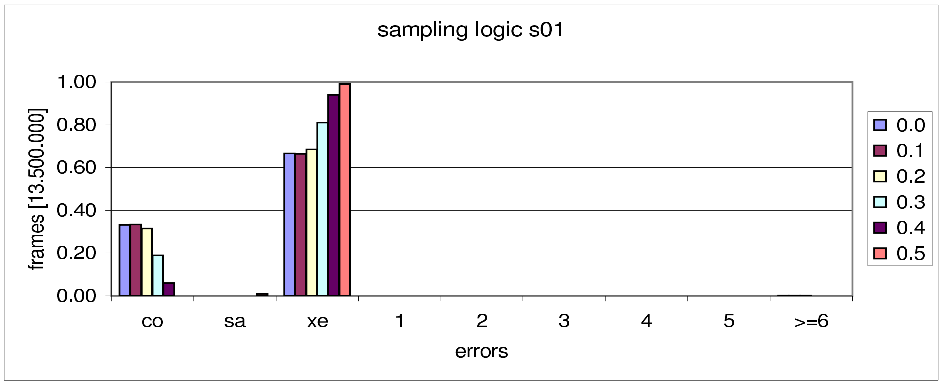

Figure 5.30: Asymmetric delayed Edge, Sampling Logic s01

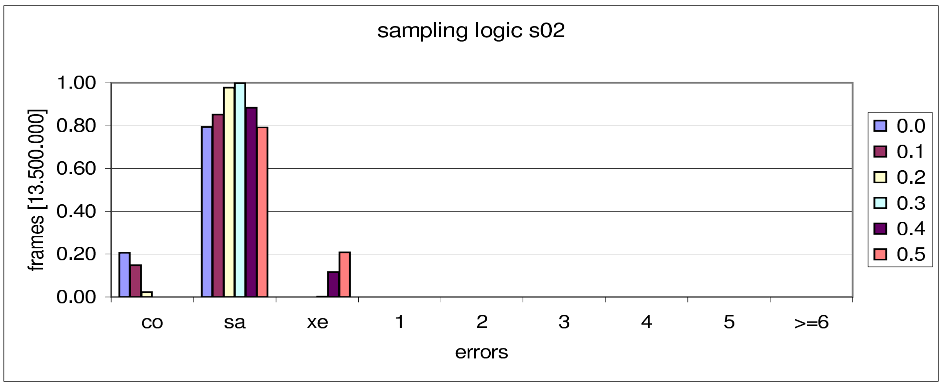

Figure 5.31: Asymmetric delayed Edge, Sampling Logic s02

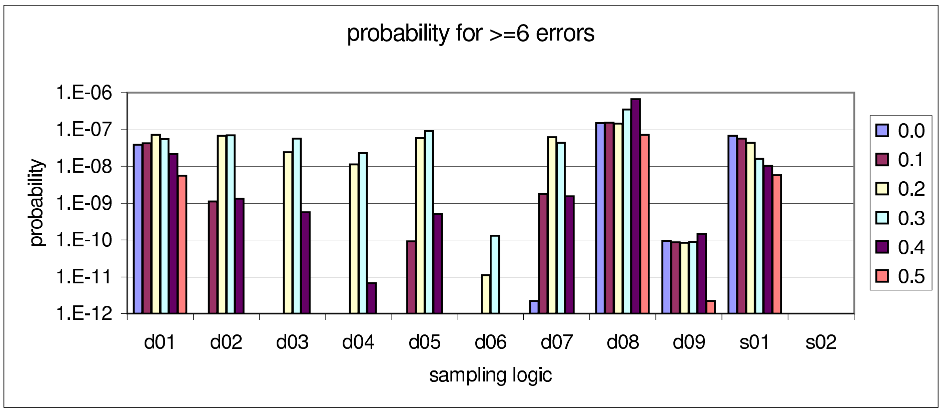

Figure 5.32: Probabilities for >5 Errors caused by a single asymmetric delayed Edge

Figure 5.33 to figure 5.43 show the results of simulation runs for delays from 0.0 to 0.5 `BCD`. It is obvious that the number of correct frames drops much faster with rising asymmetry than for single asymmetric edges, beginning with asymmetries of 0.1 `BCD`. The exceptions from this observation are the sampling logics d08 and d09, which were specially designed to achieve good results with asymmetric delays. Like in chapter 5.8.1 it can be seen that error intervals will improve the error detection.

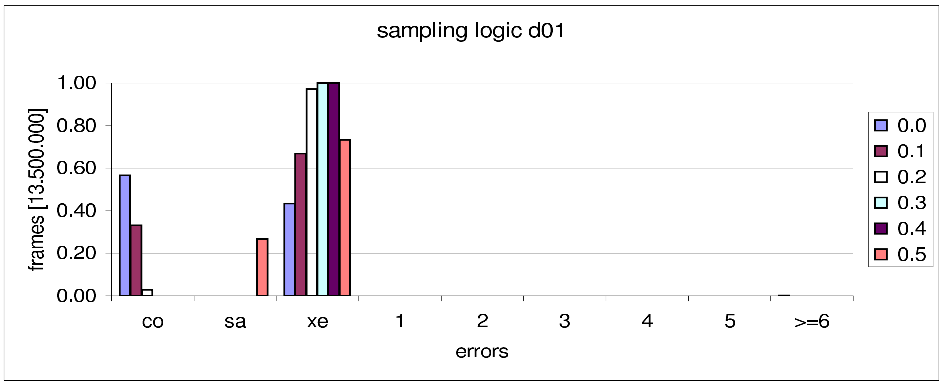

Figure 5.33: Asymmetric delayed Frame, Sampling Logic d01

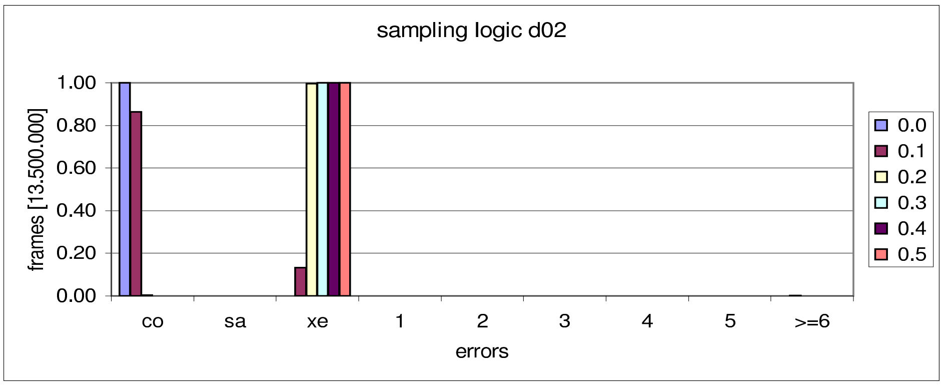

Figure 5.34: Asymmetric delayed Frame, Sampling Logic d02

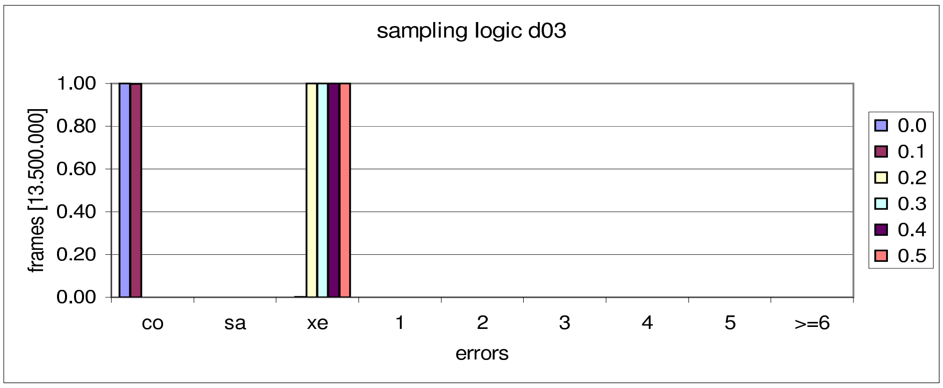

Figure 5.35: Asymmetric delayed Frame, Sampling Logic d03

Figure 5.44 shows the probabilities for more than six undetected defective bits caused by an asymmetric delay of the full frame. The axis labelling is the same as in figure 5.32 (probabilities include CRC residual error probabilities). The series show the different asymmetric delays on the falling edge of the full frame. Small values in this diagram are not a guarantee for a good receiver logic: The number of unrecognised faults with sampling logic d06 is zero but the number of correct decoded frames is also very small (see figure 5.38).

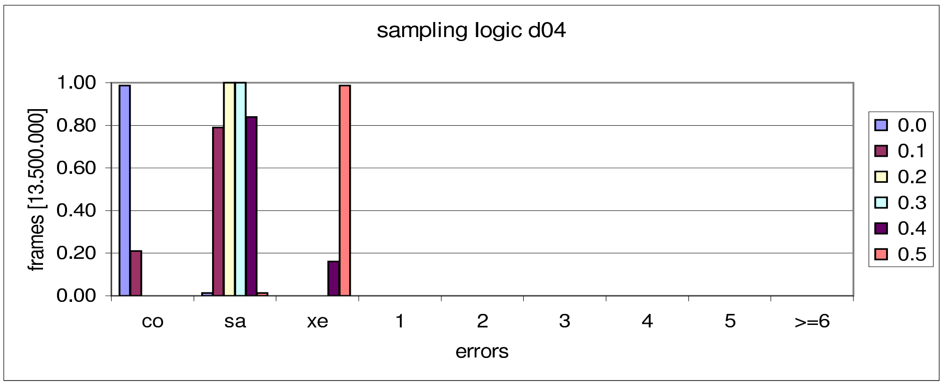

Figure 5.36: Asymmetric delayed Frame, Sampling Logic d04

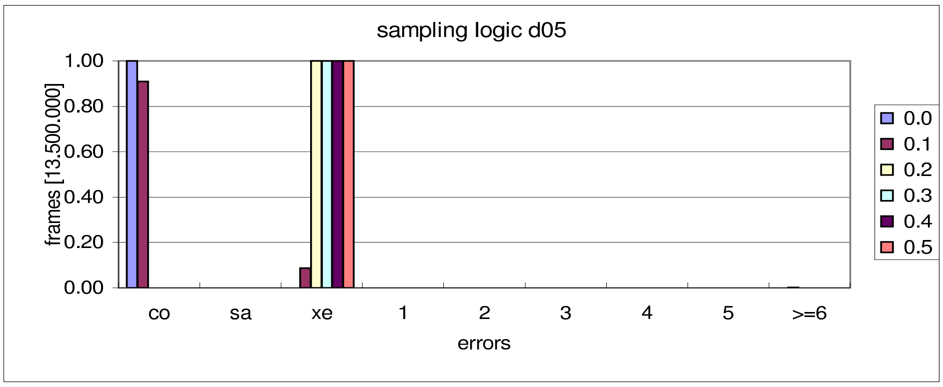

Figure 5.37: Asymmetric delayed Frame, Sampling Logic d05

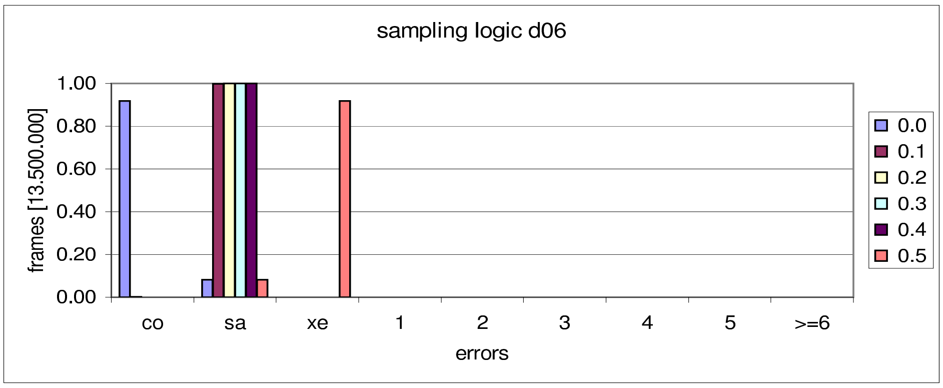

Figure 5.38: Asymmetric delayed Frame, Sampling Logic d06

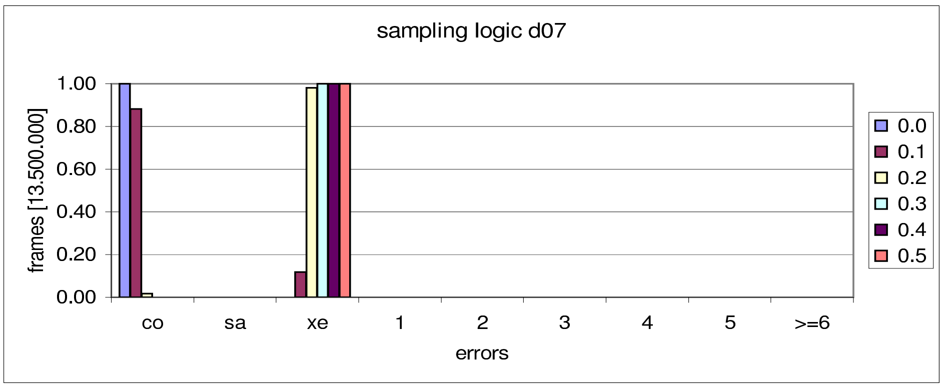

Figure 5.39: Asymmetric delayed Frame, Sampling Logic d07

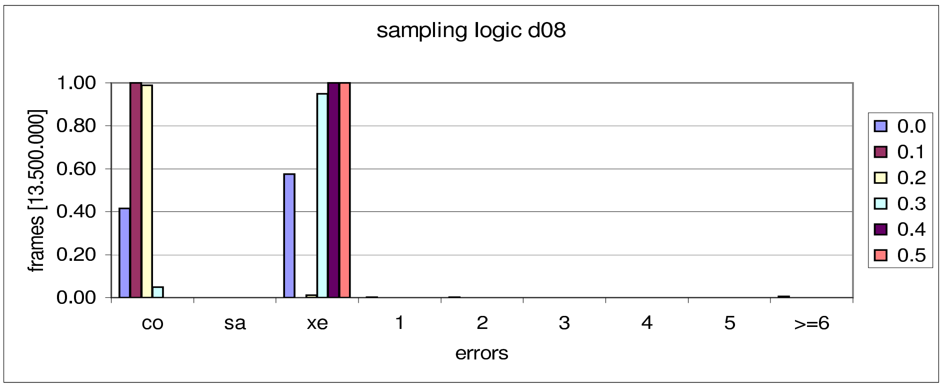

Figure 5.40: Asymmetric delayed Frame, Sampling Logic d08

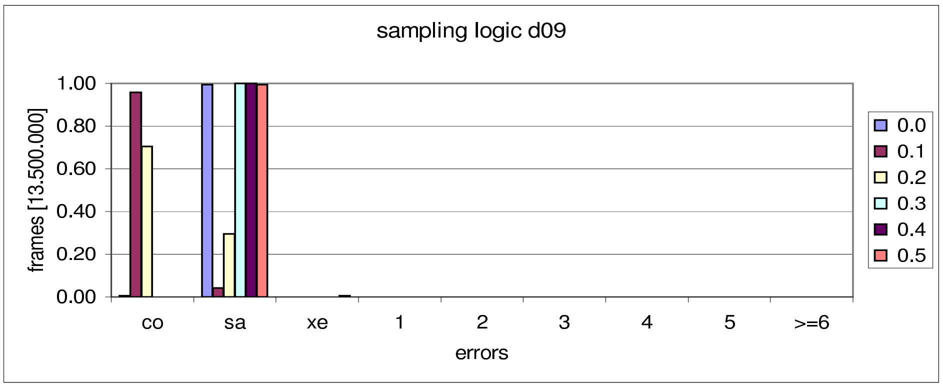

Figure 5.41: Asymmetric delayed Frame, Sampling Logic d09

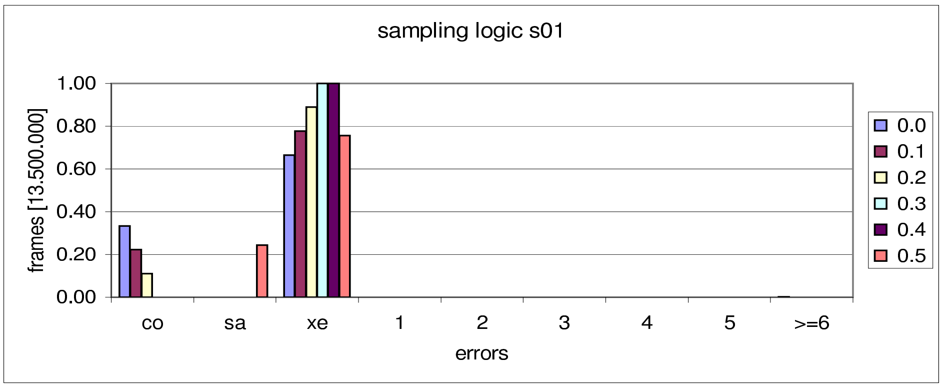

Figure 5.42: Asymmetric delayed Frame, Sampling Logic s01

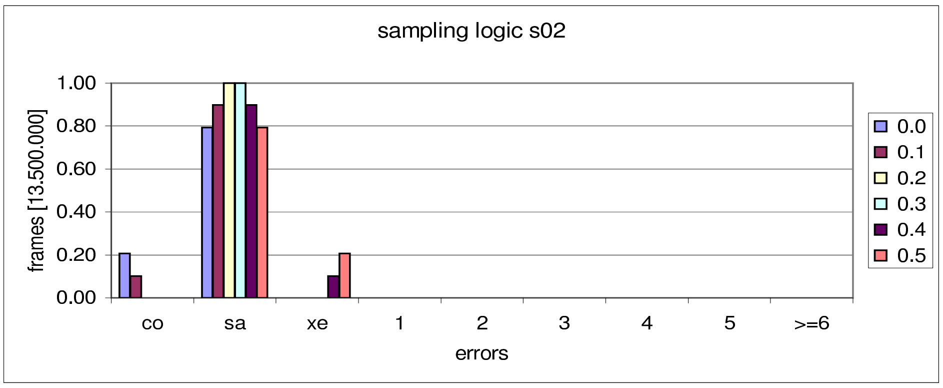

Figure 5.43: Asymmetric delayed Frame, Sampling Logic s02

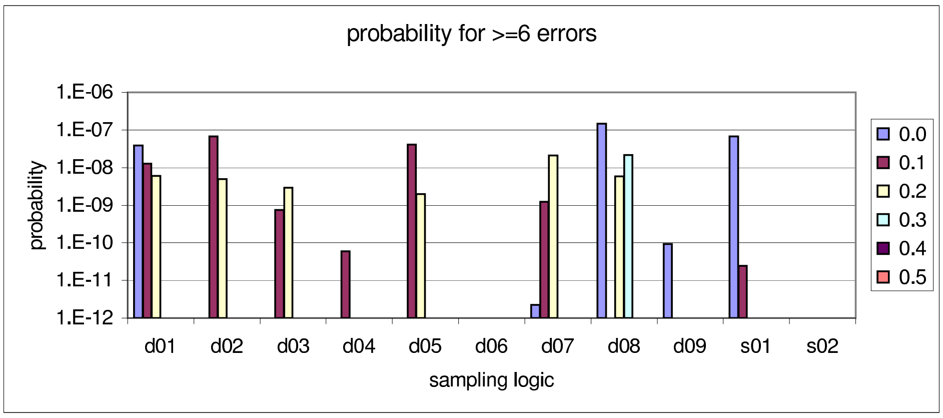

Figure 5.44: Probabilities for >5 Errors with an asymmetric delayed Frame

5.9 Evaluation

The theoretical considerations from chapter 5.2 and simulation results from the previous chapter show that the synchronisation can be lost due to injected faults on the physical line. This is a major problem since this fault can only be detected with the residual value of the CRC.

Synchronisation can be lost due to all of the faults specified in the fault model, depending on the receiver. A detailed description of each receiver type is given below.

S01: Since this sampling logic is static it suffers from long frame lengths with contemporaneous clock drifts. With a receive interval of 0.5 `BCD` the edges can move to a different sampling point slowly when the clock drifts are to high. This error cannot be detected with the CRC.

S02: Is more sensitive to clock drifts than s01 (since the receive window is only half the size) but does not produce as many undetectable errors.

D01: Was introduced to see if a sampling logic with an oversampling rate of four behaves with Xerxes encoding. D01 can only compensate clock drifts when the clock of the sender is slower or equal. Since there is one preferred direction for resynchronisation this sampling logic is worse than S01 and S02.

D02: This receiver lacks an error interval and therefore defective frames can only be recognised in the Xerxes decoder or by the CRC. As a result there can occur a large number of errors affecting more than five bits.

D03: This sampling logic is the same as d02 but with a sync interval that is only half the size of the sync interval of d02. This sampling logic gives better results than d02 for frames with asymmetric delay but inferior results with single asymmetric delayed edges. D03 gives more correct frames than d02 and less errors with five or more defective bits.

D04: This sampling logic is similar to d03 with an error interval and the sync interval symmetrically aligned around the sample points. Compared to d03 d04 recognises more frames as erroneous but decodes less frames with bit errors.

D05: This sampling logic uses ten times oversampling. There seems to be no gain from the higher oversampling rate compared to eight times oversampling. The receiver behaves similar to sampling logic d02.

D06: This sampling logic uses ten times oversampling with an error interval of 0.2 `BCD`. The performance of this sampling logic is very poor for all simulations.

D07: Ten times oversampling with a sync. Interval of 0.3 `BCD` and no error interval. The behaviour is similar to d02 with slightly less erroneous frames.

D08: d02 with built in asymmetry compensation. This is a good choice for known asymmetric delays of whole frames but gives bad results without asymmetric frame delay. The results are comparable to d02 with an asymmetric delay of 0 `BCD` even if an asymmetric delay is present. If no asymmetric delay is present, this receiver logic is a bad choice.

D09: d04 with built in asymmetry compensation. For this receiver logic the same comments apply as for d08.

The sampling logic has significant influence on the reliability of a network protocol. Some issues have been shown by the simulations:

Reliability vs. Availability (error intervals): Reliable error detection results in more erroneous frames. This can be seen by comparing d03 and d04. D03 decodes frames correctly when d04 produces an error in the sampling logic while d04 recognises most of the erroneous frames which produce more than five errors with sampling logic d03.

Oversampling rates: The simulations using d01 have shown that oversampling rates of four are unusable for an edge triggered encoding scheme. With only four intervals resynchronisation works only very poor (and only in one direction). The gain from oversampling rates higher than eight is low, compared to the effort caused to implement those oversampling rates in hardware. Ten times oversampling shows virtually no difference to eight times oversampling.

Adjustable asymmetric delay: When working with transceivers that feature a given constant asymmetric delay, a compensation for asymmetric delays gives a great gain in availability and measurable gain in reliability.

Limiting resynchronisations: an other idea, that could keep the number of erroneous resynchronisations small is to limit the number of resynchronisations to a given number (i.e. 2 resynchronisations for each 64 bits) or to introduce additional synchronisation bits in the frame. This could reduce the number of erroneous resynchronisations but has not been simulated. Additionally it would also be interesting if there is a correlation between the number of resynchronisations and the number of errors.

Concluding remarks: Since the physical layer is not specified, a concrete fault model cannot be given. Therefore, the injected faults were chosen to cover a wide range of possible failure conditions. Furthermore, the number and occurrence of injected faults per frame were significantly increased in order to minimise the simulation effort (higher occurrence of the fault types under investigation). Therefore, the diagrams do not reflect the occurrence of the faults during real operation, rather they show the distribution between tolerated (undetectable) and detected faults. Detected faults are further classified by the unit detecting the fault (sampler, decoder, CRC).

Chapter 6

Conclusion

Regarding the code stability, it has been shown in chapter 4 that the occurrence of error propagation in the Xerxes encoding scheme highly depends on the fault models. The two investigated fault models half-bit and full-bit led to two completely different results. With the half bit fault model no error propagation occurs for up to six injected faults. With the full bit fault model, error propagation is possible. Two injected faults may lead to four bit errors in the decoded stream. Since the physical layer is not yet specified, both fault models are valid. The residual error probability is, interestingly, equal for both fault models applying stochastic injected faults as shown in a detailed comparison.

A further interesting result of the code stability investigation is that even numbers of errors can always be detected by the Xerxes decoder. This feature cannot be used with the current FlexRay™ implementation since it requires a distinctive end delimiter. In addition, it has been shown that error detection probability for such small noise pulses, i.e., up to eight injected faults, is reasonably high. More than five semi or full bit errors were detected with a probability of more than 0.99998 solely by the Xerxes decoder.

Regarding the resynchronisation, it has been shown that a single injected fault can cause false resynchronisation and therefore six or more bit errors in the decoded stream. Asymmetric delays of single edges or full frames caused the most undetectable transmission errors. Furthermore, simulation runs have shown that asymmetric delays of up to $\frac{1}{4}$ `BCD` can be compensated through the sampling logic but with increasing asymmetric delays the transmitted frame gets very sensitive to small noise pulses. Simulations have also shown that compensation of asymmetric delays within the sampler logic decreases the need for dynamic resynchronisation and results in a significant increment of correctly received frames.

The results in this study are intended to help to choose a sampling logic and to define requirements for the physical layer and the bit sampling logic. Currently there is no physical layer defined, so the error probabilities presented in this study are error probabilities for a wide range of injected faults but not expected error probabilities in real applications. Therefore it would make sense to rerun the simulations performed in this study with a more detailed fault model after a physical layer has been defined. With these simulations concrete error probability values for real applications could be given.

Bibliography

[Tran 99] Eushiuan Tran, Dr. Philip Koopman, Carnegie Mellon University, “Multi-Bit Error Vulnerabilities in the Controller Area Network Protocol”, 1999

[Koopman 98] Philip Koopman, Eushiuan Tran, Geoff Hendrey, “Toward Middleware Fault Injection for Automotive Networks”, Fast Abstract, 28th International Symposium on Fault-Tolerant Somputing Systems, Munich, Germany, 1998

[Forster 01] Wolfgang Forster, Diploma Thesis, Technikum Wien, “Xerxes-Kodierung”, 2001

[Miller 80] Jerry W. Miller, Melno Park, U.S. Pat. #4234897, — “DC free Encodings for Data Transmission”, 1980

[FR_PS 01] Mathias Rausch, Mark Jordan, “FlexRay - Deterministic Automotive Protocol Specification v.0.434b”, 2001

[FRpre 01] Hubert Kirrmann, Ecole Polytechnique, Lausanne, CH, Philip Koopmann, Carnegie Mellon, Pittsburgh, US, “A Preliminary Analysis of the FlexRay Protocol”, 2001

[Freq 02] Ralf Belschner, Josef Berwanger, Christian Ebner, Harald Eisele, Sven Fluhrer, Thomas Forest, Thomas Führer, Florian Hartwich, Bernd Hedenetz, Robert Hugel, Andreas Knapp, Josef Krammer, Arnold Millsap, Bernd Müller, Martin Peller, Anton Schedl, BMW AG, DaimlerChrysler AG, Robert Bosch GmbH, General Motors/Opel, “FlexRay Requirements Specification, Vers. 1.9.9a”, 2000-2002

[DBI] Tim Bunce, J. Douglas Dunlop, Jonathan Leffler et al., DBI manual page, “DBI - Database independent interface for Perl”, 2002

[PSQL] The PostgreSQL Global development Group, “PostgreSQL 7.3.1 User’s Guide”, www.postgresql.org, 2002

[C++Ref] Bjarne Stroustru, “The C++ Programming Language (Special Edition)”, Addison Wesley. Reading Mass., ISBN 0-201-70073-5., 2000

Appendix A

Abbreviations

ASIC	Application Specific Integrated Circuit, a custom IC design
BCD	Bit Cell Duration, the duration needed to transmit a bit, 100 ns for 10MBit transer rate
ber	bit error rate, probability that a single full bit is flipped
CAN	Controller Area Network, a communication bus used in the automotive industry
CRC	Cyclic Redundancy Check, checksum generation algorithm for the verification of binary data
CVAB	Code Violation Avoidance Bit, a bit in the FlexRay™ frame used to be able to generate Xerxes-compliant frames
DBI	Data Base Interface, the Perl Data Base Interface
FCB	Frame Completion Bit, a bit in the FlexRay™ frame used to be able to generate Xerxes-compliant frames
FPGA	Field Programmable Gate Array, a programable logic device often used for prototyping
FSS	Frame Start Sequence, a part of the FlexRay™ frame
GNU	Gnu is Not Unix, a free Software Project
hber	half bit error rate, probability that a single half bit is flipped
LIN	Local Interconnect Network, an new low-cost bus protocol targeted at the automotive market
MFM	Modified Frequency Modulation, a bit coding scheme
NRZ	Non Return to Zero, a encoding scheme where bits are encoded by a high or low level for a whole bit cell
NRZ8N1	spezial kind of NRZ encoding with a start bit before and a stop bit after each group of eight transfered bits
RDS	Running Digital Sum, the integral of the signal value over time is a measurement for the DC component of a signal
STL	`C++` standard library providing containers, algorithms and streams
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Table A.1: Abbreviations

Appendix B

Simulation Results for Chapter 4

Table B.1 shows the results for figure 4.6 figure 4.6 in chapter 4.3.3.2. It can be seen that the error rates for the half bit fault model are similar to the error rates of the full bit fault model with the double error rate. The errors detected in the decoder, frames with less than six errors (detected by CRC) and possible undetectable errors (more than six errors) are denoted by “`decoder`”, “`<6err.`” and “`>=6err.`”.

	half bit			full bit		
error rate	decoder	< 6err.	$>= 6$err.	decoder	< 6err.	$>= 6$err.
$1 * 10^{-8}$	25	0	0	13	0	0
$2 * 10^{-8}$	58	0	0	29	0	0
$5 * 10^{-8}$	144	0	0	75	0	0
$1 * 10^{-7}$	288	0	0	151	0	0
$2 * 10^{-7}$	603	2	0	291	1	0
$5 * 10^{-7}$	1.508	3	0	755	2	0
$1 * 10^{-6}$	2.993	11	0	1.510	6	0
$2 * 10^{-6}$	6.100	25	0	3.100	17	0
$5 * 10^{-6}$	15.267	62	0	7.584	31	0
$1 * 10^{-5}$	30.483	116	0	15.253	51	0
$2 * 10^{-5}$	60.793	247	0	30.479	142	0
$5 * 10^{-5}$	151.709	644	0	76.250	309	0
$1 * 10^{-4}$	300.258	1.289	0	150.970	621	0
$2 * 10^{-4}$	591.927	2.573	0	300.641	1.189	0
$5 * 10^{-4}$	1.413.956	6.453	0	734.586	3.243	0
$1 * 10^{-3}$	2.626.587	12.369	0	1.415.369	6.089	0
$2 * 10^{-3}$	4.560.241	22.332	0	2.629.914	10.434	0
$5 * 10^{-3}$	7.810.757	34.440	1	5.338.444	24.514	1
$1.0 * 10^{-2}$	9.515.659	23.874	4	7.831.930	30.333	3
$1.2 * 10^{-2}$	9.734.687	17.924	4	8.406.079	29.610	12
$1.5 * 10^{-2}$	9.891.798	10.895	9	8.993.837	26.528	13
$2.0 * 10^{-2}$	9.975.443	4.080	12	9.535.009	19.906	18
$2.5 * 10^{-2}$	9.994.365	1.357	14	9.784.606	14.097	12
$3.0 * 10^{-2}$	9.998.704	425	11	9.900.639	9.137	28
$3.5 * 10^{-2}$	9.999.704	116	5	9.954.349	5.327	35
$4.0 * 10^{-2}$	9.999.935	31	2	9.978.840	3.315	21
$4.5 * 10^{-2}$	9.999.985	9	1	9.990.130	1.975	20
$5.0 * 10^{-2}$	9.999.995	0	1	9.995.601	1.009	12
$0.1 * 10^{-1}$	10.000.000	0	0	9.999.999	1	0
$0.2 * 10^{-1}$	10.000.000	0	0	10.000.000	0	0

Table B.1: Simulation Results for Chapter 4.3.3.2

Appendix C

Simulation Results for Chapter 5

Since the tables in chapter 5.6 do not show very small values well the numerical simulation results can be found here for all simulations performed in chapter 5. In studies like this small numbers are in fact very important. The diagrams were made in Excel but unfortunately Excel provides no means to show if a column is really zero or if the value is so small that it looks like zero when it is illustrated in the diagram.

All numbers are relative to the total number of simulated frames. Simulated frames of different lengths have been accumulated in the evaluation. This does not make a big difference: the results were very similar with different frame lengths. The only exception was seen using big clock drifts with both static sampling logics `s01` and `s02` and the first dynamic sampling logic `d01`. For this three cases the results are similar to the results for frame lengths of 128 Bytes.

scale=1

receiver	correct	sampler	decoder	CRC	>5 errors
d01	0.70200556	0.00000000	0.29657222	0.00077037	0.00065185
d02	0.99260185	0.00000000	0.00701296	0.00017222	0.00021296
d03	0.99996852	0.00000000	0.00003148	0.00000000	0.00000000
d04	0.92537222	0.07462407	0.00000370	0.00000000	0.00000000
d05	0.99443333	0.00000000	0.00519444	0.00017222	0.00020000
d06	0.45877407	0.54122593	0.00000000	0.00000000	0.00000000
d07	0.99990370	0.00000000	0.00009444	0.00000000	0.00000185
d08	0.96631296	0.00000000	0.03272593	0.00062593	0.00033519
d09	0.71814074	0.28176481	0.00008889	0.00000370	0.00000185
s01	0.86811852	0.00000000	0.13177593	0.00010556	0.00000000
s02	0.49774074	0.50225926	0.00000000	0.00000000	0.00000000

scale=1.01

receiver	correct	sampler	decoder	CRC	>5 errors
d01	0.77515741	0.00000000	0.22468333	0.00007222	0.00008704
d02	0.98696481	0.00000000	0.01203519	0.00046667	0.00053333
d03	0.99991852	0.00000000	0.00007963	0.00000000	0.00000185
d04	0.86396852	0.13601111	0.00002037	0.00000000	0.00000000
d05	0.98395000	0.00000000	0.01500000	0.00051667	0.00053333
d06	0.39300741	0.60699259	0.00000000	0.00000000	0.00000000
d07	0.99961111	0.00000000	0.00037407	0.00000926	0.00000556
d08	0.91585000	0.00000000	0.08125370	0.00205000	0.00084630
d09	0.61092963	0.38904074	0.00002778	0.00000185	0.00000000
s01	0.21627222	0.00000000	0.76714259	0.01379815	0.00278704
s02	0.00017407	0.99982593	0.00000000	0.00000000	0.00000000

scale=0.99

receiver	correct	sampler	decoder	CRC	>5 errors
d01	0.00902407	0.00000000	0.96473889	0.01634630	0.00989074
d02	0.98293704	0.00000000	0.01555556	0.00068704	0.00082037
d03	0.99973704	0.00000000	0.00025926	0.00000185	0.00000185
d04	0.88347222	0.11652593	0.00000185	0.00000000	0.00000000
d05	0.99864444	0.00000000	0.00125556	0.00003333	0.00006667
d06	0.42240741	0.57759259	0.00000000	0.00000000	0.00000000
d07	0.99931296	0.00000000	0.00063889	0.00001852	0.00002963
d08	0.90882778	0.00000000	0.08724630	0.00285185	0.00107407
d09	0.64374259	0.35625741	0.00000000	0.00000000	0.00000000
s01	0.06076852	0.00000000	0.92645000	0.00921852	0.00356296
s02	0.00000370	0.99999630	0.00000000	0.00000000	0.00000000

Table C.1: Simulation Results for Chapter 5.7.1

jitter=0.01

receiver	correct	sampler	decoder	CRC	>5 errors
d01	0.56318000	0.00000000	0.42756444	0.00583407	0.00342148
d02	1.00000000	0.00000000	0.00000000	0.00000000	0.00000000
d03	1.00000000	0.00000000	0.00000000	0.00000000	0.00000000
d04	0.99624963	0.00375037	0.00000000	0.00000000	0.00000000
d05	1.00000000	0.00000000	0.00000000	0.00000000	0.00000000
d06	0.48723852	0.51276148	0.00000000	0.00000000	0.00000000
d07	1.00000000	0.00000000	0.00000000	0.00000000	0.00000000
d08	0.97671037	0.00000000	0.02269259	0.00044963	0.00014741
d09	0.72330519	0.27669481	0.00000000	0.00000000	0.00000000
s01	0.55480444	0.00000000	0.43775852	0.00590963	0.00152741
s02	0.25064889	0.74935111	0.00000000	0.00000000	0.00000000

jitter=0.05

receiver	correct	sampler	decoder	CRC	>5 errors
d01	0.50484815	0.00000000	0.48691037	0.00487259	0.00336889
d02	0.96395630	0.00000000	0.03350148	0.00111556	0.00142667
d03	0.99980889	0.00000000	0.00018741	0.00000148	0.00000222
d04	0.80762222	0.19236370	0.00001407	0.00000000	0.00000000
d05	0.98645333	0.00000000	0.01268444	0.00039556	0.00046667
d06	0.41108148	0.58891852	0.00000000	0.00000000	0.00000000
d07	0.99944741	0.00000000	0.00052593	0.00001111	0.00001556
d08	0.84237778	0.00000000	0.15266148	0.00357778	0.00138296
d09	0.61992593	0.38000667	0.00006370	0.00000296	0.00000074
s01	0.53292370	0.00000000	0.46223481	0.00380444	0.00103704
s02	0.20006741	0.79993259	0.00000000	0.00000000	0.00000000

Table C.2: Simulation Results for Chapter 5.7.2

offs=0

receiver	correct	sampler	decoder	CRC	>5 errors
d01	0.53401407	0.00000000	0.45723741	0.00535333	0.00339519
d02	0.98197815	0.00000000	0.01675074	0.00055778	0.00071333
d03	0.99990444	0.00000000	0.00009370	0.00000074	0.00000111
d04	0.90193593	0.09805704	0.00000704	0.00000000	0.00000000
d05	0.99322667	0.00000000	0.00634222	0.00019778	0.00023333
d06	0.44916000	0.55084000	0.00000000	0.00000000	0.00000000
d07	0.99972370	0.00000000	0.00026296	0.00000556	0.00000778
d08	0.90954407	0.00000000	0.08767704	0.00201370	0.00076519
d09	0.67161556	0.32835074	0.00003185	0.00000148	0.00000037
s01	0.54386407	0.00000000	0.44999667	0.00485704	0.00128222
s02	0.22535815	0.77464185	0.00000000	0.00000000	0.00000000

offs=0.3

receiver	correct	sampler	decoder	CRC	>5 errors
d01	0.42437037	0.00000000	0.55725296	0.01030185	0.00807481
d02	0.30593111	0.00000000	0.65080185	0.01636259	0.02690444
d03	0.62798296	0.00000000	0.34985815	0.00838148	0.01377741
d04	0.00201667	0.55683185	0.41428481	0.01043296	0.01643370
d05	0.19945889	0.00000000	0.75190778	0.01834333	0.03029000
d06	0.00000519	0.89132074	0.10217185	0.00256741	0.00393481
d07	0.32114444	0.00000000	0.63801222	0.01581889	0.02502444
d08	0.05205370	0.00000000	0.89389185	0.02108370	0.03297074
d09	0.00000815	0.43799259	0.52684407	0.01383481	0.02132037
s01	0.24603148	0.00000000	0.72910852	0.01127926	0.01358074
s02	0.00000185	0.99486037	0.00422111	0.00046852	0.00044815

offs=0.5

receiver	correct	sampler	decoder	CRC	>5 errors
d01	0.00833444	0.00000000	0.94067926	0.02705481	0.02393148
d02	0.00000000	0.00000000	0.93775778	0.02353000	0.03871222
d03	0.00000259	0.00000000	0.93998704	0.02333630	0.03667407
d04	0.00000000	0.09805704	0.84448259	0.02238296	0.03507741
d05	0.00000000	0.00000000	0.93898556	0.02346222	0.03755222
d06	0.00000000	0.55084000	0.41943037	0.01162963	0.01810000
d07	0.00000185	0.00000000	0.93997444	0.02334148	0.03668222
d08	0.00000000	0.00000000	0.94182037	0.02268185	0.03549778
d09	0.00000000	0.32835074	0.62885889	0.01663926	0.02615111
s01	0.00035704	0.00000000	0.94724222	0.02122704	0.03117370
s02	0.00000000	0.77464185	0.20778407	0.00682444	0.01074963

Table C.3: Simulation Results for Chapter 5.7.4

no forced signal

receiver	correct	sampler	decoder	CRC	>5 errors
d01	0.53401407	0.00000000	0.45723741	0.00535333	0.00339519
d02	0.98197815	0.00000000	0.01675074	0.00055778	0.00071333
d03	0.99990444	0.00000000	0.00009370	0.00000074	0.00000111
d04	0.90193593	0.09805704	0.00000704	0.00000000	0.00000000
d05	0.99322667	0.00000000	0.00634222	0.00019778	0.00023333
d06	0.44916000	0.55084000	0.00000000	0.00000000	0.00000000
d07	0.99972370	0.00000000	0.00026296	0.00000556	0.00000778
d08	0.90954407	0.00000000	0.08767704	0.00201370	0.00076519
d09	0.67161556	0.32835074	0.00003185	0.00000148	0.00000037
s01	0.54386407	0.00000000	0.44999667	0.00485704	0.00128222
s02	0.22535815	0.77464185	0.00000000	0.00000000	0.00000000

signal forced to high

receiver	correct	sampler	decoder	CRC	>5 errors
d01	0.20086407	0.00366074	0.78535778	0.00861852	0.00149889
d02	0.37368926	0.08201185	0.53314222	0.00978185	0.00137481
d03	0.36768963	0.12623370	0.49694185	0.00894222	0.00019259
d04	0.32944259	0.42069481	0.24478630	0.00479222	0.00028407
d05	0.36457222	0.13087333	0.49370556	0.00956667	0.00128222
d06	0.16348370	0.76044815	0.07428963	0.00177852	0.00000000
d07	0.37333074	0.14402926	0.47377370	0.00842407	0.00044222
d08	0.39680407	0.00038333	0.58439889	0.01485481	0.00355889
d09	0.24651852	0.56424148	0.18201926	0.00601963	0.00120111
s01	0.23550037	0.08869926	0.66901556	0.00621889	0.00056593
s02	0.08119889	0.88124222	0.03655037	0.00100852	0.00000000

signal forced to high and low

receiver	correct	sampler	decoder	CRC	>5 errors
d01	0.12276704	0.03887481	0.81479926	0.02193148	0.00162741
d02	0.22569148	0.12409963	0.59878148	0.04550333	0.00592407
d03	0.22997963	0.19456815	0.52681444	0.04734519	0.00129259
d04	0.20698519	0.57600370	0.19392815	0.02194444	0.00113852
d05	0.22837778	0.16843333	0.54170778	0.05399778	0.00748333
d06	0.10272667	0.82141481	0.06819926	0.00765852	0.00000074
d07	0.22994111	0.15767481	0.56242111	0.04618481	0.00377815
d08	0.20844185	0.20512741	0.53788185	0.04609111	0.00245778
d09	0.15385481	0.70389704	0.12839444	0.01380556	0.00004815
s01	0.12264667	0.17329296	0.67369926	0.03005259	0.00030852
s02	0.05144778	0.90908704	0.03443000	0.00503519	0.00000000

Table C.4: Simulation Results for Chapter 5.7.5

no delayed edge

receiver	correct	sampler	decoder	CRC	>5 errors
d01	0.53401407	0.00000000	0.45723741	0.00535333	0.00339519
d02	0.98197815	0.00000000	0.01675074	0.00055778	0.00071333
d03	0.99990444	0.00000000	0.00009370	0.00000074	0.00000111
d04	0.90193593	0.09805704	0.00000704	0.00000000	0.00000000
d05	0.99322667	0.00000000	0.00634222	0.00019778	0.00023333
d06	0.44916000	0.55084000	0.00000000	0.00000000	0.00000000
d07	0.99972370	0.00000000	0.00026296	0.00000556	0.00000778
d08	0.90954407	0.00000000	0.08767704	0.00201370	0.00076519
d09	0.67161556	0.32835074	0.00003185	0.00000148	0.00000037
s01	0.54386407	0.00000000	0.44999667	0.00485704	0.00128222
s02	0.22535815	0.77464185	0.00000000	0.00000000	0.00000000

two delayed edges

receiver	correct	sampler	decoder	CRC	>5 errors
d01	0.20071185	0.00000000	0.76938407	0.02500630	0.00489778
d02	0.55795148	0.00000000	0.42535481	0.01528444	0.00140926
d03	0.33453667	0.00000000	0.62066111	0.04077333	0.00402889
d04	0.06507148	0.93297481	0.00190741	0.00004407	0.00000222
d05	0.50760889	0.00000000	0.46699222	0.02281667	0.00258222
d06	0.00335407	0.99664296	0.00000296	0.00000000	0.00000000
d07	0.37761481	0.00000000	0.58452000	0.03425148	0.00361370
d08	0.82033111	0.00000000	0.16808852	0.01090556	0.00067481
d09	0.23903111	0.75876963	0.00166519	0.00053370	0.00000037
s01	0.33059704	0.00000000	0.65374148	0.01501148	0.00065000
s02	0.00093370	0.99906444	0.00000185	0.00000000	0.00000000

Table C.5: Simulation Results for Chapter 5.7.6

rcv.	asym	correct	sampler	xerxes	b01	b02	b03	b04	b05	>5 err.
d01	0.0	0.5657556296	0.0000000000	0.4328385185	0.0000177037	0.0000358519	0.0000159259	0.0000328889	0.0000123704	0.0012569630
d01	0.1	0.5854918519	0.0000000000	0.4129432593	0.0000237778	0.0000485926	0.0000196296	0.0000395556	0.0000163704	0.0013736296
d01	0.2	0.2915338519	0.0000000000	0.7059662222	0.0000237778	0.0000485926	0.0000196296	0.0000396296	0.0000164444	0.0023085185
d01	0.3	0.0381331852	0.0000000000	0.9598811111	0.0000234074	0.0000475556	0.0000191111	0.0000392593	0.0000157778	0.0017984444
d01	0.4	0.0000014074	0.0000003704	0.9992794815	0.0000000741	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0007186667
d01	0.5	0.0000000000	0.0037644444	0.9960515556	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0001840000
d02	0.0	0.9999999259	0.0000000000	0.0000000741	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d02	0.1	0.9912051852	0.0000000000	0.0087581481	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000366667
d02	0.2	0.6186078519	0.0000000000	0.3791695556	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0022225926
d02	0.3	0.1857323704	0.0000000000	0.8119637778	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0023038519
d02	0.4	0.0006183704	0.0000000000	0.9993378519	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000437778
d02	0.5	0.0000000000	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d03	0.0	0.9999999259	0.0000000000	0.0000000741	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d03	0.1	0.9999778519	0.0000000000	0.0000221481	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d03	0.2	0.7950254074	0.0000000000	0.2041604444	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0008141481
d03	0.3	0.0285743704	0.0000000000	0.9695548148	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0018708148
d03	0.4	0.0000000741	0.0000000000	0.9999811111	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000188148
d03	0.5	0.0000000000	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d04	0.0	0.9859764444	0.0140235556	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d04	0.1	0.8060780741	0.1939202222	0.0000017037	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d04	0.2	0.1348468148	0.7930080000	0.0717701481	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0003750370
d04	0.3	0.0000236296	0.8377596296	0.1614567407	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0007600000
d04	0.4	0.0000000000	0.1101625185	0.8898372593	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000002222
d04	0.5	0.0000000000	0.0139297778	0.9860702222	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d05	0.0	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d05	0.1	0.9987105185	0.0000000000	0.0012864444	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000030370
d05	0.2	0.6695268148	0.0000000000	0.3285134815	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0019597037
d05	0.3	0.0110698519	0.0000000000	0.9859226667	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0030074815
d05	0.4	0.0000000000	0.0000000000	0.9999831852	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000168148
d05	0.5	0.0000000000	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d06	0.0	0.9180927407	0.0819072593	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d06	0.1	0.6307091111	0.3692908889	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d06	0.2	0.0109732593	0.9887642222	0.0002621481	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000003704
d06	0.3	0.0000000000	0.9977955556	0.0022000741	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000043704
d06	0.4	0.0000000000	0.3923794815	0.6076205185	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d06	0.5	0.0000000000	0.0819072593	0.9180927407	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d07	0.0	0.9999750370	0.0000000000	0.0000246667	0.0000000741	0.0000000000	0.0000001481	0.0000000000	0.0000000000	0.0000000741
d07	0.1	0.9866508148	0.0000000000	0.0132899259	0.0000000741	0.0000000000	0.0000001481	0.0000000000	0.0000000000	0.0000590370
d07	0.2	0.6442567407	0.0000000000	0.3536978519	0.0000000741	0.0000001481	0.0000000000	0.0000000000	0.0000000000	0.0020451852
d07	0.3	0.1516470370	0.0000000000	0.8469188148	0.0000000000	0.0000002222	0.0000000000	0.0000000000	0.0000000000	0.0014339259
d07	0.4	0.0000060000	0.0000000000	0.9999426667	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000513333
d07	0.5	0.0000000000	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d08	0.0	0.4163277778	0.0000000000	0.5743936296	0.0011433333	0.0012481481	0.0008105185	0.0008100741	0.0003688148	0.0044924444
d08	0.1	0.3773711111	0.0000000000	0.6133740000	0.0009989630	0.0010664444	0.0007846667	0.0007738519	0.0004814815	0.0046159259
d08	0.2	0.3621645185	0.0000000000	0.6291436296	0.0009182222	0.0009696296	0.0007563704	0.0007460000	0.0004654815	0.0043162222
d08	0.3	0.2140072593	0.0000000000	0.7710371111	0.0005264444	0.0017419259	0.0004771852	0.0004591852	0.0003222222	0.0110535556
d08	0.4	0.0074543704	0.0000000000	0.9693048148	0.0000193333	0.0007278519	0.0000348148	0.0000345185	0.0000394815	0.0223304444
d08	0.5	0.0000000741	0.0000000000	0.9976432593	0.0000000741	0.0000002963	0.0000001481	0.0000005926	0.0000000000	0.0023555556
d09	0.0	0.0050550370	0.9948360741	0.0001029630	0.0000006667	0.0000006667	0.0000007407	0.0000003704	0.0000003704	0.0000025185
d09	0.1	0.0055533333	0.9943240000	0.0001171852	0.0000005926	0.0000006667	0.0000005185	0.0000003704	0.0000004444	0.0000024444
d09	0.2	0.0048730370	0.9950144444	0.0001074074	0.0000005926	0.0000005926	0.0000004444	0.0000003704	0.0000002963	0.0000024444
d09	0.3	0.0008140741	0.9990104444	0.0001722963	0.0000000741	0.0000000000	0.0000000741	0.0000000741	0.0000000000	0.0000029630
d09	0.4	0.0000000000	0.9994591852	0.0005360000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000048148
d09	0.5	0.0000000000	0.9947137778	0.0052861481	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000741
s01	0.0	0.3325543704	0.0000000741	0.6652133333	0.0000029630	0.0000014815	0.0000000000	0.0000000000	0.0000000000	0.0022277778
s01	0.1	0.3331698519	0.0000000000	0.6649376296	0.0000008889	0.0000002222	0.0000000000	0.0000000000	0.0000000000	0.0018914074
s01	0.2	0.3142731111	0.0000000000	0.6843014815	0.0000008889	0.0000002222	0.0000000000	0.0000000000	0.0000000000	0.0014242963
s01	0.3	0.1889844444	0.0000000000	0.8104874074	0.0000008889	0.0000002222	0.0000000000	0.0000000000	0.0000000000	0.0005270370
s01	0.4	0.0600760000	0.0000002963	0.9395842222	0.0000008889	0.0000002222	0.0000000000	0.0000000000	0.0000000000	0.0003383704
s01	0.5	0.0000011111	0.0084580741	0.9913495556	0.0000000000	0.0000008889	0.0000000000	0.0000000000	0.0000000000	0.0001903704
s02	0.0	0.2071903704	0.7928096296	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
s02	0.1	0.1477547407	0.8522452593	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
s02	0.2	0.0227605185	0.9772394815	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
s02	0.3	0.0000000000	0.9976562222	0.0023437778	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
s02	0.4	0.0000000000	0.8843247407	0.1156752593	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
s02	0.5	0.0000000000	0.7921990370	0.2078009630	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000

Table C.6: Simulation Results for Chapter 5.8.1

Table C.7: Simulation Results for Chapter 5.8.2

rcv.	asym	correct	sampler	xerxes	b01	b02	b03	b04	b05	>5 err.
d01	0.0	0.5657556296	0.0000000000	0.4328385185	0.0000177037	0.0000358519	0.0000159259	0.0000328889	0.0000123704	0.0012569630
d01	0.1	0.3314774815	0.0000000000	0.6678979259	0.0000414815	0.0000629630	0.0000240000	0.0000499259	0.0000191852	0.0003640741
d01	0.2	0.0285468148	0.0000000000	0.9711355556	0.0000227407	0.0000417778	0.0000151111	0.0000301481	0.0000103704	0.0001663704
d01	0.3	0.0000000000	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d01	0.4	0.0000000000	0.0000017778	0.9999982222	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d01	0.5	0.0000000000	0.2678260000	0.7321740000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d02	0.0	0.9999999259	0.0000000000	0.0000000741	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d02	0.1	0.8634422222	0.0000000000	0.1326434815	0.0004211852	0.0004441481	0.0003242963	0.0003094815	0.0001607407	0.0021095556
d02	0.2	0.0031345185	0.0000000000	0.9965818519	0.0000312593	0.0000305185	0.0000226667	0.0000197037	0.0000142963	0.0001496296
d02	0.3	0.0000000000	0.0000001481	0.9999998519	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d02	0.4	0.0000000000	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d02	0.5	0.0000000000	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d03	0.0	0.9999999259	0.0000000000	0.0000000741	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d03	0.1	0.9981903704	0.0000000000	0.0017662963	0.0000027407	0.0000040741	0.0000040000	0.0000045926	0.0000027407	0.0000214074
d03	0.2	0.0007122963	0.0000001481	0.9991285926	0.0000143704	0.0000171111	0.0000108148	0.0000120000	0.0000067407	0.0000863704
d03	0.3	0.0000000000	0.0000001481	0.9999998519	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d03	0.4	0.0000000000	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d03	0.5	0.0000000000	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d04	0.0	0.9859764444	0.0140235556	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d04	0.1	0.2094741481	0.7904685185	0.0000544444	0.0000003704	0.0000002222	0.0000001481	0.0000000741	0.0000000741	0.0000018519
d04	0.2	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d04	0.3	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d04	0.4	0.0000000000	0.8396128148	0.1603871852	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d04	0.5	0.0000000000	0.0140235556	0.9859764444	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d05	0.0	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d05	0.1	0.9101166667	0.0000000000	0.0876858519	0.0001795556	0.0001931111	0.0001767407	0.0001789630	0.0001189630	0.0012344444
d05	0.2	0.0003868148	0.0000000000	0.9995041481	0.0000120741	0.0000100000	0.0000071852	0.0000080000	0.0000050370	0.0000611111
d05	0.3	0.0000000000	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d05	0.4	0.0000000000	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d05	0.5	0.0000000000	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d06	0.0	0.9180927407	0.0819072593	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d06	0.1	0.0017382963	0.9982617037	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d06	0.2	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d06	0.3	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d06	0.4	0.0000000000	0.9990724444	0.0009275556	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d06	0.5	0.0000000000	0.0819072593	0.9180927407	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d07	0.0	0.9999750370	0.0000000000	0.0000246667	0.0000000741	0.0000000000	0.0000001481	0.0000000000	0.0000000000	0.0000000741
d07	0.1	0.8820974074	0.0000000000	0.1178336296	0.0000080741	0.0000075556	0.0000039259	0.0000043704	0.0000034815	0.0000385926
d07	0.2	0.0180031852	0.0000000000	0.9808568148	0.0001030370	0.0001184444	0.0000730370	0.0000765185	0.0000628889	0.0006382222
d07	0.3	0.0000000000	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d07	0.4	0.0000000000	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d07	0.5	0.0000000000	0.0000016923	0.9999983077	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d08	0.0	0.4163277778	0.0000000000	0.5743936296	0.0011433333	0.0012481481	0.0008105185	0.0008100741	0.0003688148	0.0044924444
d08	0.1	0.9999939259	0.0000000000	0.0000060000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000741	0.0000000000
d08	0.2	0.9885620741	0.0000000000	0.0111143704	0.0000329630	0.0000328889	0.0000242222	0.0000224444	0.0000147407	0.0001834074
d08	0.3	0.0496011852	0.0000000000	0.9492053333	0.0001125926	0.0001191111	0.0000924444	0.0000827407	0.0000614815	0.0006636296
d08	0.4	0.0000000000	0.0000000741	0.9999999259	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d08	0.5	0.0000000000	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d09	0.0	0.0050550370	0.9948360741	0.0001029630	0.0000006667	0.0000006667	0.0000007407	0.0000003704	0.0000003704	0.0000025185
d09	0.1	0.9572285926	0.0427714074	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d09	0.2	0.7052231111	0.2947749630	0.0000019259	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d09	0.3	0.0000000000	1.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d09	0.4	0.0000000000	0.9999945926	0.0000054074	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
d09	0.5	0.0000000000	0.9948360741	0.0051639259	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
s01	0.0	0.3325543704	0.0000000741	0.6652133333	0.0000029630	0.0000014815	0.0000000000	0.0000000000	0.0000000000	0.0022277778
s01	0.1	0.2221680000	0.0000001481	0.7778254074	0.0000056296	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000008148
s01	0.2	0.1110568889	0.0000001481	0.8889373333	0.0000056296	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
s01	0.3	0.0004082222	0.0000000741	0.9995877778	0.0000039259	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
s01	0.4	0.0000000000	0.0000066667	0.9999933333	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
s01	0.5	0.0000000000	0.2438390370	0.7561609630	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
s02	0.0	0.2071903704	0.7928096296	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
s02	0.1	0.1007003704	0.8992996296	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
s02	0.2	0.0000004444	0.9999995556	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
s02	0.3	0.0000000000	0.9999716296	0.0000283704	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
s02	0.4	0.0000000000	0.8988037037	0.1011962963	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000
s02	0.5	0.0000000000	0.7928096296	0.2071903704	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000	0.0000000000

Appendix D

Code Examples

This chapter contains code examples that were too big to be placed elsewhere in this paper.

D.1 Implementation of the `XerxesDecoder` Class

Listing D.1 shows the implementation of the Xerxes Decoder introduced in chapter 3.3.2.2. In this listing the header file and the implementation have been combined.

```
class Decoder {
 public:
    virtual void decode(const CodeStream& CS, BinStream& BS) = 0;
    static const char* errstring(int _reason);
    static const int CodeError    = 1;
    static const int SamplerError = 2;
};

class Xerxes_Decoder : public Decoder {
 public:
    Xerxes_Decoder();
    void decode(const CodeStream& CS, BinStream& BS);
    static  void init();
 protected:
    static void addTransition(const byte  _c_state, const byte  _n_state,
                              const char* _c_sym,   const char* _n_sym,
                              const byte  _bit);
    inline byte do_transition(byte& _State, byte _curr, byte _next);
    static const byte xSta  =  1;
    static const byte xA    =  2;
    static const byte xB    =  3;
    static const byte xC    =  4;
    static const byte xD    =  5;
    static const byte xW    =  6;
    static const byte xX    =  7;
    static const byte xY    =  8;
    static const byte xZ    =  9;
    static const byte xSto  = 10;
    static byte m_transitions[256];
};

byte Xerxes_Decoder::m_transitions[256];

class _Decoder_init { // static initialisation object for XD
public:
    _Decoder_init() { Xerxes_Decoder::init(); }
} __Decoder_init;
```

```

const char* Decoder::errstring(int _reason)
{
    switch (_reason) {
      case CodeError:    return "Xerxes Code Error";
      case SamplerError: return "Xerxes Symbol Error (Sampler)";
      default:           return "Unknown Error";
    }
}

void Xerxes_Decoder::addTransition(const byte  _c_state,
                                   const byte  _n_state,
                                   const char* _c_sym,
                                   const char* _n_sym,
                                   const byte  _bit)
{
    while(*_c_sym != 0) {
        byte c = CodeStream::from_char(*_c_sym);
        const char* next = _n_sym;
        while(*next != 0) {
            byte n = CodeStream::from_char(*next);
            m_transitions[(_c_state & 0x0f) | ((n & 3)<<4) | ((c & 3)<<6)] =
                (_n_state & 0x0f) | ((_bit & 3) << 4);
            next++;
        }
        _c_sym++;
    }
}

void Xerxes_Decoder::init()
{
    for(int i=0; i<256; m_transitions[i++]=0);
    addTransition(xSta,  xSta,   "I",   "I",    2);
    addTransition(xSta,  xA,     "I",   "D",    2);
    addTransition(xSta,  xD,     "I",   "C",    2);
    addTransition(xA,    xA,     "D",   "DN",   1);
    addTransition(xA,    xX,     "N",   "CDN",  0);
    addTransition(xD,    xX,     "C",   "CDN",  0);
    addTransition(xD,    xA,     "D",   "DN",   1);
    addTransition(xC,    xA,     "D",   "DN",   1);
    addTransition(xC,    xX,     "C",   "CD",   0);
    addTransition(xX,    xD,     "C",   "CD",   0);
    addTransition(xX,    xW,     "D",   "NI",   1);
    addTransition(xX,    xY,     "C",   "N",    1);
    addTransition(xW,    xC,     "N",   "CD",   0);
    addTransition(xY,    xZ,     "N",   "CD",   1);
    addTransition(xZ,    xY,     "C",   "N",    1);
    addTransition(xZ,    xW,     "D",   "NI",   1);
    addTransition(xZ,    xB,     "C",   "CD",   0);
    addTransition(xB,    xA,     "D",   "DN",   1);
    addTransition(xB,    xX,     "C",   "CD",   0);
    addTransition(xB,    xSto,   "C",   "I",    0);
    addTransition(xB,    xSto,   "I",   "I",    2);
    addTransition(xD,    xSto,   "C",   "I",    0);
    addTransition(xD,    xSto,   "I",   "I",    2);
    addTransition(xW,    xSto,   "NI",  "I",    0);
    addTransition(xX,    xSto,   "C",   "I",    0);
    addTransition(xX,    xSto,   "I",   "I",    2);
    addTransition(xZ,    xSto,   "C",   "I",    0);
    addTransition(xZ,    xSto,   "I",   "I",    2);
    addTransition(xSto,  xSto,   "I",   "I",    0);
}
```

```

Xerxes_Decoder::Xerxes_Decoder()
{
}

inline byte Xerxes_Decoder::do_transition(byte& _State, byte _curr, byte _next)
{
    byte rv = m_transitions[(_State & 0x0f) | ((_next & 3)<<4) |
                            ((_curr & 3)<<6)];
    if(rv==0) throw int(CodeError);
    _State = rv & 0x0f;
    byte code = (rv >> 4) & 7;
    return code;
}

void Xerxes_Decoder::decode(const CodeStream& CS, BinStream& BS)
{
    static byte idle  = CodeStream::from_char('I');
    byte        State = xSta;
    byte c = idle;
    byte n = idle;
    int soc_len = 0;
    while(CS.getb(soc_len)==CodeStream::D) soc_len++;
    if(CS.getb(soc_len++)!=CodeStream::N) throw int(CodeError);
    if(CS.getb(soc_len++)!=CodeStream::D) throw int(CodeError);
    if(CS.getb(soc_len++)!=CodeStream::N) throw int(CodeError);
    for(int i=0; i<CS.size()+3; i++) {
        n = CS.getb(i);
        if(n==CodeStream::E) throw int(SamplerError);
        // propagate error from Sampler
        if(n==CodeStream::B) throw int(CodeError);
        byte bit = do_transition(State, c, n);
        if(bit<2) BS.append(bit);
        c =  n;
    }
    BS.remove((BS.size() - soc_len) % 8);
}
```

Listing D.1: Implementation of the XerxesDecoder

D.2 Implementation of the Sampling Logic

Listing D.2 shows the sampling logic which is included in chapter 3.3.2.4 as pseudo code.

```
void TimeDecoder_Default::decode(const std::vector<double>& _timevec,
                                 CodeStream& _CS)
{
    std::vector<double>::const_iterator i = _timevec.begin();
    double swh = fabs(m_sync_win_half);
    double swoffs = fabs((fabs(m_sync_win_half)-m_sync_win_half))/2;
    double t_clock = 0, diff;
    double last;
    int    line=1;
    int    toggle;
    _CS.clear();
    t_clock = _timevec.front() - 0.5 - m_offs + m_granularity/2.0;
    while(i!=_timevec.end()) {
        bool clock = false;
        bool data  = false;
        if(t_clock > *i+5) throw int(Decoder::SamplerError);
        toggle = 1;
        while(i!=_timevec.end() && check_intervall(0.2500001, *i-t_clock)) {
            toggle ^= 1;
            if (toggle == 1) {
                if(*i > last+m_granularity)
                  throw int(Decoder::SamplerError);
            } else
                last   = *i;
            if(check_intervall(m_rcv_win_half, *i-t_clock)) clock = !clock;
            else throw int(Decoder::SamplerError);
            diff = *i-t_clock+swoffs;
            if(!check_intervall(swh, diff)) {
                if(m_granularity>1e-8)
                    diff = floor(diff / m_granularity) * m_granularity;
                t_clock += diff;
            }
            if(m_asym!=0) {
                double asym = static_cast<double>(m_asym);
                if(line==1) t_clock -= asym * m_granularity;
                else        t_clock += asym * m_granularity;
                line ^= 1;
            }
            ++i;
        }
        t_clock += 0.5;
        toggle = 1;
        while(i!=_timevec.end() && check_intervall(0.2500001, *i-t_clock)) {
            toggle ^= 1;
            if (toggle == 1) {
                if(*i > last+m_granularity)
                   throw int(Decoder::SamplerError);
            } else
                last   = *i;
            if(check_intervall(m_rcv_win_half, *i-t_clock)) data = !data;
            else throw int(Decoder::SamplerError);
            diff = *i-t_clock+swoffs;
            if(!check_intervall(swh, diff)) {
                if(m_granularity>1e-8)
                    diff = floor(diff / m_granularity) * m_granularity;
                t_clock += diff;
            }
            if(m_asym!=0) {
                double asym = static_cast<double>(m_asym);
                if(line==1) t_clock -= asym * m_granularity;
                else        t_clock += asym * m_granularity;
                line ^= 1;
            }
            ++i;
        }
        t_clock += 0.5;
        if(clock && data)  _CS.append_data(CodeStream::B);
        else if(clock)     _CS.append_data(CodeStream::C);
        else if(data)      _CS.append_data(CodeStream::D);
        else               _CS.append_data(CodeStream::N);
    }
    _CS.append_data(CodeStream::I);
}
```
Listing D.2: Implementation of the Sampling Logic

D.3 Implementation of the Filters

The following listings show the implementation of the Filter framework. All Filter classes but `Filter_Scale` and `Filter_Scale` have been removed from the listing to save space. The Filter Framework is used for fault injection and allows an arbitrary number of filters to be serialized in a filter chain.

```
class Filter {
 public:
    virtual ~Filter();
    virtual void filter(std::vector<double>& _timevec) = 0;
 protected:
    class FilterRange {
    public:
        FilterRange();
        void set_range(const std::string& _start, const std::string& _stop);
        void calc_range(std::vector<double>& _timevec);
        std::vector<double>::iterator begin();
        std::vector<double>::iterator end();
        char m_start_type;
        union { double dval; unsigned int uval; } m_start_time;
        char m_stop_type;
        union { double dval; unsigned int uval; } m_stop_time;
    private:
        std::vector<double>::iterator m_begin;
        std::vector<double>::iterator m_end;
        double m_begin_offset;
        double m_end_offset;
    };
};

class FilterFactory
{
 public:
    typedef Filter* createfunc (const std::string& _args);
    struct FilterInfo {
        const char*  m_name;
        const char*  m_helptext;
        const char*  m_defaults;
        createfunc*  m_create;
    };
    FilterFactory();
    void append(const std::string& _filter_desc);
    void clear();
    void filter(std::vector<double>& _timevec);
    std::string help() const;
 private:
    std::vector<FilterInfo> m_finfo;
    std::vector<Filter*> m_filters;
};
extern FilterFactory Filters;

class Filter_Jitter : public Filter {
 public:
    static Filter* create(const std::string& _arguments);
    static const   FilterFactory::FilterInfo info();
    void           filter(std::vector<double>& _timevec);
 private:
    Filter_Jitter();
    static const char* m_help;
    static const char* m_defaults;
    double              m_jitter;
    FilterRange         m_range;
};

class Filter_Scale : public Filter {
 public:
    static Filter* create(const std::string& _arguments);
    static const   FilterFactory::FilterInfo info();
    void           filter(std::vector<double>& _timevec);
 private:
    Filter_Scale();
    static const char* m_help;
    static const char* m_defaults;
    double              m_scale;
    FilterRange         m_range;
};

inline std::vector<double>::iterator Filter::FilterRange::begin()
{
    return  m_begin;
}

inline std::vector<double>::iterator Filter::FilterRange::end()
{
    return  m_end;
}
```

Listing D.3: Header File for the Filter implementation

```
FilterFactory Filters;

const char* Filter_Scale::m_help =
"    scale=scale[:start={@|+|*}time][:stop={@|+|*}time]\n"
"        scale the time axis around the edge given in start\n"
"        this filter can be used to simulate oscillator drifts\n";
const char* Filter_Scale::m_defaults = "scale=1.0:start=@0:stop=@0";

const char* Filter_Jitter::m_help =
"    jitter=jitterwidth[:start={@|+|*}time][:stop={@|+|*}time]\n"
"        add a random offset to each edge (normal distribution\n"
"        jitterwidth=3*sigma, values from -jw to +jw)\n";
const char* Filter_Jitter::m_defaults = "jitter=0.25/3.0:start=@0:stop=@0";

///////////////////////////////////////////////////////////////////////////

FilterFactory::FilterFactory()
{
    m_finfo.push_back(Filter_Shift::info());
    m_finfo.push_back(Filter_Scale::info());
    m_finfo.push_back(Filter_Jitter::info());
    m_finfo.push_back(Filter_Asym::info());
    m_finfo.push_back(Filter_High::info());
    m_finfo.push_back(Filter_Low::info());
}

std::string FilterFactory::help() const
{
    std::string help;
    help  = "\n";
    help += "FILTER HELP\n";
    help += "***********\n";
    help += "available filters:\n";
    for(std::vector<FilterInfo>::const_iterator i=m_finfo.begin();
          i!=m_finfo.end(); ++i)
        help += i->m_helptext;
    help += "\n";
    help += "arguments:\n";
    help += "    <arg>    mandatory arguments\n";
    help += "    [arg]    optional arguments\n";
    help += "    a|b      multiple possibilities\n";
    help += "    @n       nht edge starting from 1, (default all = @0,@0)\n";
    help += "    *5.5     abs. time in BCD units (frame starts at 5.0)\n";
    help += "    +8.9     rel. time in BCD units (from frame start)\n";
    help += "    #0.5     rel. time from first selected edge\n";
    help += "    :        argument seperator, may be omitted at the end\n";
    help += "examples for filter expressions:\n";
    help += "    scale=1.0001          creates a slow clock drift "
            "(oscillator drift)\n";
    help += "    offset=0.5:start=@2   delays each but the first "
            "edge by 0.5 BCD\n";
    return help;
}

void FilterFactory::append(const std::string& _filter_desc)
{
    std::vector<std::string> temp;
    stringtok(temp, _filter_desc, ":=", "", 2);
    if(temp.size()>=1)
        for(std::vector<FilterInfo>::const_iterator i=m_finfo.begin();
              i!=m_finfo.end(); ++i)
            if(i->m_name == temp[0])
                m_filters.push_back(i->m_create(_filter_desc));
}

void FilterFactory::clear()
{
    while(m_filters.size()!=0) {
        delete m_filters.back();
        m_filters.pop_back();
    }
}

void FilterFactory::filter(std::vector<double>& _timevec)
{
    for(std::vector<Filter*>::iterator i=m_filters.begin();
          i!=m_filters.end(); ++i)
        (*i)->filter(_timevec);
}

///////////////////////////////////////////////////////////////////////////

Filter::FilterRange::FilterRange()
{
}

void Filter::FilterRange::set_range(const std::string& _start,
                                    const std::string& _stop)
{
    m_start_type = _start[0];
    if(m_start_type=='@') {
        m_start_time.uval = atol(_start.c_str()+1);
        if(m_start_time.uval!=0) m_start_time.uval--;
    } else if (m_start_type=='*') {
        m_start_time.dval = strtod(_start.c_str()+1, NULL);
    } else if (m_start_type=='+') {
        m_start_time.dval = strtod(_start.c_str()+1, NULL);
    } else throw int(0);
    m_stop_type = _stop[0];
    if(m_stop_type=='@') {
        m_stop_time.uval = atol(_stop.c_str()+1);
    } else if (m_stop_type=='*') {
        m_stop_time.dval = strtod(_stop.c_str()+1, NULL);
    } else if (m_stop_type=='+') {
        m_stop_time.dval = strtod(_stop.c_str()+1, NULL);
    } else if (m_stop_type=='#') {
        m_stop_time.dval = strtod(_stop.c_str()+1, NULL);
    } else throw int(0);
}

void Filter::FilterRange::calc_range(std::vector<double>& _timevec)
{
    double temp;
    if(m_start_type=='@') {
        if(m_start_time.uval<_timevec.size())
            m_begin = _timevec.begin() + m_start_time.uval;
        else
            m_begin = _timevec.end();
    } else {
        if (m_start_type=='*') {
            temp = m_start_time.dval - 1e-8;
        } else {
            temp = m_start_time.dval + _timevec[0] - 0.5 - 1e-8;
        }
        for(m_begin = _timevec.begin();
            m_begin!=_timevec.end()  &&  *m_begin<temp; ++m_begin);
    }
    if(m_stop_type=='@') {
        if(m_stop_time.uval==0 || m_stop_time.uval>=_timevec.size())
            m_end = _timevec.end();
        else
            m_end = _timevec.begin() + m_stop_time.uval;
    } else {
        if (m_start_type=='*') {
            temp = m_stop_time.dval + 1e-8;
        } else if (m_start_type=='+') {
            temp = m_stop_time.dval + _timevec[0] - 0.5 + 1e-8;
        } else if (m_start_type=='#') {
            if(m_begin != _timevec.end())
                temp = m_stop_time.dval + *m_begin + 1e-8;
            else
                temp=-1;
        }
        for(m_end=m_begin; m_end!=_timevec.end()  &&  *m_end<temp; ++m_end);
    }
    if(m_begin>=m_end) {
        m_begin=_timevec.end();
        m_end=_timevec.end();
    }
}

Filter::~Filter()
{
}

///////////////////////////////////////////////////////////////////////////
```

```

Filter_Scale::Filter_Scale()
{
}

Filter* Filter_Scale::create(const std::string& _arguments)
{
    Filter_Scale* FS = new Filter_Scale;
    std::map<std::string,std::string> config;
    ParseCfg(config, m_defaults);
    ParseCfg(config, _arguments);
#ifdef DEBUG
    std::cerr << "Filter_Scale::create(" << _arguments << ")" << std::endl;
    for(map<std::string,std::string>::iterator ci=config.begin();
          ci!=config.end(); ++ci)
        std::cerr << ci->first << "=>" << ci->second << std::endl;
#endif
    FS->m_scale = strtod(config["scale"].c_str(), NULL);
    FS->m_range.set_range(config["start"], config["stop"]);
    return FS;
}

const FilterFactory::FilterInfo Filter_Scale::info()
{
    FilterFactory::FilterInfo fs;
    fs.m_name      = "scale";
    fs.m_helptext  = m_help;
    fs.m_defaults  = m_defaults;
    fs.m_create    = &create;
    return fs;
}

void Filter_Scale::filter(std::vector<double>& _timevec)
{
    m_range.calc_range(_timevec);
    double center = *m_range.begin();
    for(std::vector<double>::iterator i=m_range.begin(); i!=m_range.end();
          ++i)
        *i = (*i-center) * m_scale + center;
}

///////////////////////////////////////////////////////////////////////////

Filter_Jitter::Filter_Jitter()
{
}

Filter* Filter_Jitter::create(const std::string& _arguments)
{
    Filter_Jitter* FS = new Filter_Jitter;
    std::map<std::string,std::string> config;
    ParseCfg(config, m_defaults);
    ParseCfg(config, _arguments);
#ifdef DEBUG
    std::cerr << "Filter_Jitter::create(" << _arguments << ")" << std::endl;
    for(map<std::string,std::string>::iterator ci=config.begin();
          ci!=config.end(); ++ci)
        std::cerr << ci->first << "=>" << ci->second << std::endl;
#endif
    FS->m_jitter = strtod(config["jitter"].c_str(), NULL) / 3.0;
    FS->m_range.set_range(config["start"], config["stop"]);
    return FS;
}

const FilterFactory::FilterInfo Filter_Jitter::info()
{
    FilterFactory::FilterInfo fs;
    fs.m_name      = "jitter";
    fs.m_helptext  = m_help;
    fs.m_defaults  = m_defaults;
    fs.m_create    = &create;
    return fs;
}

void Filter_Jitter::filter(std::vector<double>& _timevec)
{
    m_range.calc_range(_timevec);
    for(std::vector<double>::iterator i=m_range.begin();
          i!=m_range.end(); ++i)
        *i = *i + gaussian_distribution(m_jitter);
}
```

Listing D.4: Source File for the Filter implementation

D.4 Implementation of the `time_frame` Function

Listing D.5 shows the implementation of the `time_frame2` function. This is function is called from the `time_frame` function to simulate `_framecount` frames until the desired number of frames has been simulated.

```
unsigned long time_frame2(BinStream& _frame, const int _framecount,
                          unsigned long& _rcv_det, unsigned long& _xer_det,
                          unsigned long& _crc_det, unsigned long& _not_det,
                          unsigned long errors[17])
{
    BinStream& BS_corr = _frame;
    CodeStream CS_corr;
    CodeStream CS_err;
    BinStream  BS_err;

    //BS_corr.append_crc(8);
    g_encoder->encode(BS_corr, CS_corr);

    std::vector<double> tv_corr;
    g_time_enc->encode(CS_corr, tv_corr);

    for(int i=0; i<_framecount; ++i) {

        std::vector<double> tv_err(tv_corr);
        bool caught = false;
        Filters.filter(tv_err);
        CS_err.clear();
        BS_err.clear();
        try {
            g_time_dec->decode(tv_err,CS_err);
            g_decoder->decode(CS_err,BS_err);
        } catch (int& e) {
            if(e==Decoder::CodeError)
                ++_xer_det;
            else
                ++_rcv_det;
            caught = true;
        }
        if(!caught) {
            int diff = diffrence(BS_corr, BS_err);
            if(diff>15) diff=16;
            errors[diff]++;
            if(diff>6) {
                _not_det++;
            } else if (diff!=0) _crc_det++;
        }
    }
    return _framecount;
}
```

Listing D.5: Implementation of the `time_frame` function