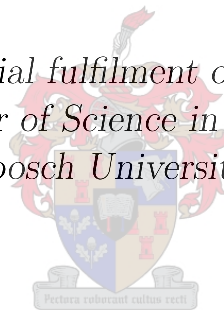
Implementation of a Protocol and Channel Coding Strategy for use in Ground-Satellite Applications

by

Riaan Wiid



Thesis presented in partial fulfilment of the requirements for the degree Master of Science in Engineering at Stellenbosch University

Supervisor : Dr. R. Wolhuter
Department of Electrical & Electronic Engineering

March 2012

Stellenbosch University http://scholar.sun.ac.za

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date : March 2012

Stellenbosch University http://scholar.sun.ac.za

Abstract

Implementation of a Protocol and Channel Coding Strategy for use in Ground-Satellite Applications

R. Wiid

Department of Electrical and Electronic Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MScEng (E&E)

March 2012

A collaboration between the Katholieke Universiteit van Leuven (KUL) and Stellenbosch University (SU), resulted in the development of a satellite based platform for use in agricultural sensing applications. This will primarily serve as a test platform for a digitally beam-steerable antenna array (SAA) that was developed by KUL. SU developed all flight - and ground station based hardware and software, enabling ground to flight communications and interfacing with the KUL SAA. Although most components had already been completed at the start of this *M.Sc.Eng.* project, final systems integration was still unfinished. Modules necessary for communication were also outstanding. This project implemented an automatic repeat and request (ARQ) strategy for reliable file transfer across the wireless link. Channel coding has also been implemented on a field programmable gate array (FPGA). This layer includes an advanced forward error correction (FEC) scheme i.e. a low-density parity-check (LDPC), which outperforms traditional FEC techniques. A flexible architecture for channel coding has been designed that allows speed and complexity trade-offs on the FPGA. All components have successfully been implemented, tested and integrated. Simulations of LDPC on the FPGA have been shown to provide excellent error correcting performance. The prototype has been completed and recently successfully demonstrated at KUL. Data has been reliably transferred between the satellite platform and a ground station, during this event.

Stellenbosch University http://scholar.sun.ac.za

Uittreksel

Implementasie van ’n Kommunikasie Protokol en Kanaalkoderingstrategie vir Gebruik in Grond-Satelliet Toepassings

R. Wiid

Departement Elektries en Elektroniese Ingenieurswese,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.

Tesis: MScIng (E&E)

Maart 2012

Tydens ’n samewerkingsooreenkoms tussen die Katholieke Universiteit van Leuven (KUL) en die Universiteit van Stellenbosch (US) is ’n satelliet stelsel ontwikkel vir sensor-netwerk toepassings in die landbou bedryf. Hierdie stelsel sal hoofsaaklik dien as ’n toetsmedium vir ’n digitaal stuurbare antenna (SAA) wat deur KUL ontwikkel is. Die US het alle hardeware en sagteware komponente ontwikkel om kommunikasie d.m.v die SAA tussen die satelliet en ’n grondstasie te bewerkstellig. Sedert die begin van hierdie *M.Sc.Ing.* projek was die meeste komponente alreeds ontwikkel en geïmplementeer, maar finale stelselsintegrasie moes nog voltooi word. Modules wat kommunikasie sou bewerkstellig was ook nog uistaande. Hierdie projek het ’n ARQ protokol geïmplementeer wat data betroubaar tussen die satelliet en ’n grondstasie kon oordra. Kanaalkodering is ook op ’n veld programmeerbare hekskikking (FPGA) geïmplementeer. ’n Gevorderde foutkorrigeringstelsel, naamlik ’n lae digtheids pariteit toetskode (LDPC), wat tradisionele foutkorrigeringstelsels se doeltreffendheid oortref, word op hierdie FPGA geïmplementeer. ’n Kanaalkoderingsargitektuur is ook ontwikkel om die verwerkingspoed van data en die hoeveelheid FPGA logika wat gebruik word, teenoor mekaar op te weeg. Alle komponente is suksesvol geïmplementeer, getoets en geïntegreer met die hele stelsel. Simulasies van LDPC op die FPGA het uistekende foutkorrigeringsresultate gelewer. ’n Werkende prototipe is onlangs voltooi en suksesvol gedemonstreer by KUL. Betroubare data oordrag tussen die satelliet en die grondstasie is tydens hierdie demonstrasie bevestig.

Stellenbosch University http://scholar.sun.ac.za

Acknowledgements

I would like to express my gratitude towards the following persons :

- God for always inspiring me to give my best.
- Dr. Riaan Wolhuter, my study leader, for his wisdom and guidance during difficult challenges of this project.
- Dr. Gert-Jan van Rooyen who provided many insights during the software design of this project.
- Rob Anderson for helping to track down and identify countless RF problems.
- Project colleagues Ewald van der Westhuizen, Wynand van Eden and Kobus Botha who provided many technical assistance during systems integration.
- Dr. Vladimir Volski and Hadi Aliakbarian for helping to successfully demonstrate this project in Leuven.
- Jaco du Toit for sharing his insights of LDPC FEC with me.
- All my friends and colleagues from the DSP lab for interesting conversations over a cup of coffee.
- My parents for constantly supporting and motivating me during the course of this Masters degree.

Stellenbosch University http://scholar.sun.ac.za

Contents

Abstract **ii**

Uittreksel **iii**

Acknowledgements **iv**

Contents **v**

List of Figures **vii**

List of Tables **xi**

List of Abbreviations **xii**

Nomenclature **xv**

1 Introduction **1**
1.1 Background 1
1.2 Motivation for Work 2
1.3 Project Objectives 3
1.4 Project Contributions and Summary 3
1.5 Outline of Thesis 4

2 Previous Work and Literature Review **5**
2.1 IS-HS 2 Theory of Operation 6
2.2 Existing Work on IS-HS 2 6
2.3 Protocols 10
2.4 Inter Protocol Layer Communication 11
2.5 Error Control Strategies 17
2.6 Wireless Channels 24
2.7 Summary 27

3 Detail Design **29**
3.1 Block Error Probability Analysis 29
3.2 Channel Coding Design 34

Stellenbosch University http://scholar.sun.ac.za

3.3 BCH FEC Design . . . 38
3.4 LDPC FEC Design . . . 44
3.5 IPC Design . . . 52
3.6 Software Protocol Layers . . . 53
3.7 FEC Block Error Rate Simulation . . . 59
3.8 Summary . . . 64

4 Implementation **65**
4.1 Existing Hardware Layout . . . 65
4.2 Channel Coding Implementation . . . 67
4.3 BCH Implementation . . . 71
4.4 LDPC Implementation . . . 74
4.5 IPC Implementation . . . 87
4.6 TM Implementation . . . 88
4.7 ARQ Implementation . . . 92
4.8 Summary . . . 98

5 Testing, Results and Discussion **99**
5.1 Channel Coding . . . 99
5.2 BCH . . . 102
5.3 LDPC . . . 105
5.4 TM and ARQ Protocols . . . 109
5.5 Belgium Demonstration . . . 110
5.6 Summary . . . 113

6 Conclusion, Contributions and Recommendations **114**
6.1 Conclusion and Summary . . . 114
6.2 Contributions to the Project . . . 115
6.3 Recommendations . . . 116

References **118**

Appendices **122**

A Mathematical Derivations **123**
A.1 QPSK Bit Error Probability Analysis . . . 123
A.2 Signal-to-Noise Ratio for Simulations . . . 129

Stellenbosch University http://scholar.sun.ac.za

List of Figures

2.1 Communications channel block diagram. 5
2.2 Intended interaction between the satellite and ground station platforms. 6
2.3 Block diagram of both ground station and satellite platforms. . . . 8
2.4 Layers of the OSI model [1]. 11
2.5 Shared memory between 2 processes. 12
2.6 QNX message passing between a client and server process. 14
2.7 A message queue shared between two processes. 15
2.8 A FEC encoder and decoder in a communications channel. 18
2.9 A (7,4) Hamming code's parity bit dependency diagram. 19
2.10 Visual representation of a parity check matrix. 23
(a) Parity check matrix of the Tanner graph in Fig. 2.10b. . . 23
(b) Tanner graph of parity check matrix in Fig. 2.10a 23
2.11 A transmitter and receiver communicating over a wireless channel. . 24
2.12 Section of a communications channel included in a data error probability model. 25
2.13 A BSC model. 26
2.14 A BEC model. 26
2.15 A BI-AWGN channel model. 27

3.1 QPSK symbol decision - and error regions. 30
(a) A QPSK signal constellation. 30
(b) Error region for symbol S_1 in Fig. 3.1a. 30
3.2 Gaussian white noise added to S_1. 30
3.3 An aircraft passing over a ground station at altitude $h = 3$ km. . . 32
3.4 Codeword error probability vs. SNR for BCH when using block length $n = 511$ bits. 33
3.5 Interaction between channel coding modules on the FPGA for both ground station and satellite platforms. 34
3.6 CRC encoding procedure. 35
3.7 Pseudo random sequence LFSR. 37
3.8 LFSR computed by the Berlekamp-Massey algorithm [2]. 41
3.9 A Chien search circuit [3]. 44
3.10 A QC-LDPC parity matrix structure. 45

Stellenbosch University http://scholar.sun.ac.za

(a) Circulant composition of a QC-LDPC parity check marix **H**. 45
(b) An example of a 5x5 circulant permutation matrix. 45
3.11 Message passing along the edges of a Tanner graph. 47
3.12 Dimensions of sub-matrices within **H**. 51
3.13 Layout of **H** using the template in Fig. 3.12. 51
3.14 Message passing IPC using POSIX semaphores and shared memory. 52
3.15 Interaction between OSI software layers. 54
3.16 TM frame layout. 56
3.17 TM header layout [1]. 56
3.18 Setting FHP when ARQ packet spans multiple TM frames. 57
3.19 An ARQ packet structure. 58
3.20 Packet round trip time measurement. 60
3.21 General operation of simulator. 61
3.22 Bit probability decision making. 62
3.23 Hardware based BER simulation. 63

4.1 Channel coding on the satellite platform. 65
4.2 Channel coding on the ground station platform. 67
4.3 Interface of a general channel coding module. 68
4.4 Timing diagram of the channel coding module from Fig. 4.3. 68
(a) Timing diagram when receiving data on *dat_in*. 68
(b) Timing diagram when outputting data on *dat_out*. 68
4.5 State machine diagram of the module in Fig. 4.3. 69
4.6 Serial implementation of a polynomial division LFSR. 71
4.7 Codeword as constructed by BCH encoder. 72
4.8 State machine diagram of a BCH decoder. 73
4.9 A BCH decoder's hardware layout. 74
4.10 Cyclic matrix multiplier architecture from [4]. 75
4.11 A reduced complexity cyclic matrix multiplier architecture. 75
4.12 A parallelised implemenation of Fig. 4.11. 76
4.13 A modified TM frame structure for half code rate LDPC. 77
4.14 State machine diagram of a LDPC encoder. 78
4.15 LDPC encoder hardware layout. 79
4.16 State machine diagram of a LDPC decoder. 80
4.17 General hardware layout of a LDPC decoder. 82
4.18 A 5-bit LLR value. 83
4.19 A CNU's hardware layout. 83
4.20 A VNU's hardware layout. 84
4.21 RAM block configuration for **H**. 85
4.22 A square permutation matrix stored as a row vector. 86
4.23 Address partitioning of a 9-kbit RAM block. 86
4.24 Logical to physical address translation. 86
4.25 Startup of a TM module. 89

Stellenbosch University http://scholar.sun.ac.za

4.26 The receive thread of a TM module. 90
4.27 The transmit thread of a TM module. 91
4.28 Startup of an ARQ module. 92
4.29 Round trip time measurements. 93
4.30 The receive thread of an ARQ module. 95
4.31 The acknowledge procedure from Fig. 4.30. 96
4.32 The transmit thread of an ARQ module. 97

5.1 Hardware loopback test in FPGA for channel coding modules. . . . 100
5.2 Measurement of channel coding's processing delay. 101
(a) Encoding delay measurement. 101
(b) Decoding delay measurement. 101
5.3 BLER plot of a (511,484) BCH code. 103
5.4 BLER plot of a (511,259) BCH code. 104
5.5 Optimal α search for the $(512, 256)$ code. 106
5.6 Optimal $\alpha = 0.9$ compared against $\alpha = 1$. 106
5.7 Termination of $\alpha = 0.9$ scaling at different iteration counts. 107
5.8 Comparison between $ET = 15$ iterations and no ET when using $\alpha = 0.9$. 107
5.9 Comparison between FPGA and Matlab simulations for $\alpha = 0.9$ and $ET = 15$ iterations. 108
5.10 Bit error rate comparison between half rate BCH and LDPC implementations from Matlab. LDPC uses $ET = 15$ and $\alpha = 0.9$. . . . 108
5.11 TM and ARQ testing procedure. 110
5.12 IS-HS 2 demo setup in Belgium. 111
5.13 Application receiving files from ARQ on ground station FIT-PC. . . 111
5.14 CRC error rate while moving from A to B in Fig. 5.12. 112
5.15 Images received on the ground station after file transfer from the satellite platform. 113
(a) 113
(b) 113
(c) 113
(d) 113

A.1 QPSK symbol decision - and error regions. 123
(a) A QPSK signal constellation. 123
(b) Error region for symbol S_1 in Fig. A.1a. 123
A.2 Vector representation of Gaussian noise added to symbol S_1. 124
(a) Gaussian white noise added to S_1. 124
(b) Unit area of integration when using polar coordinates. . . . 124
A.3 Gray coding scheme for the symbols of a QPSK constellation. . . . 127
A.4 Time domain BPSK signal. 129
A.5 Frequency domain of a BPSK bit. 129

Stellenbosch University http://scholar.sun.ac.za

A.6 A QPSK symbol amplitude i.t.o two BPSK symbols on channels I and Q. 131

Stellenbosch University http://scholar.sun.ac.za

List of Tables

2.1 Description of OSI layers in Fig. 2.4. 11

3.1 Link budget parameters for the uplink when using aircraft altitude $h = 3$ km and elevation angle $\theta = 30°$. 32
3.2 PC to FPGA control byte values. 64

4.1 SH4 to FPGA expansion port lines and their description. 66
4.2 Message passing IPC functions for Linux Ubuntu 7.10. 87

5.1 Channel coding FPGA module implementation details. 100
5.2 BCH FPGA module implementation details. 102
5.3 LDPC FPGA module implementation details. 105

Stellenbosch University http://scholar.sun.ac.za

List of Abbreviations

A/D	Analogue to Digital
API	Application Programming Interface
ARQ	Automatic Repeat-Request
ASE	Aircraft Satellite Emulator
ASM	Attached Synchronisation Marker
AWGN	Additive White Gaussian Noise
BCH	Bose Chaudhuri Hocquenghem
BEC	Binary Erasure Channel
BEP	Bit Error Probability
BER	Bit Error Ratio
BI-AWGN	Binary AWGN
BLEP	Block Error Probability
BLER	Block Error Rate
BMA	Berlekamp Massey Algorithm
BP	Belief Propagation
BPSK	Binary Phase Shift Keying
BSC	Binary Symmetric Channel
CAN	Controller-Area Network
CCSDS	Consultative Committee for Space Data Systems
CFDP	CCSDS File Delivery Protocol
CNU	Check Node Update
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DSP	Digital Signal Processing
DVB-S	Digital Video Broadcast Satellite
EA	Euclidean Algorithm
EDAC	Error Detection And Correction
ECSS	European Cooperation for Space Standardization
ET	Early Termination
FEC	Forward Error Correction
FER	Frame Error Rate
FHP	First Header Pointer
FIFO	First In First Out

Stellenbosch University http://scholar.sun.ac.za

FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FTDI	Future Technology Devices Incorporated
GEO	Geostationary Orbit
GF	Galois Field
GPS	Global Positioning System
GS	Ground Station
GUI	Graphical User Interface
ID	Identifier
I/O	Input and Output
IP	Internet Protocol
IPC	Interprocess Communication
ISE	Integrated Software Environment
IS-HS	In-Situ-Hyperspectral
ISM	Industrial Scientific and Medical
ISO	International Organisation for Standardization
KUL	Katholieke Universiteit van Leuven
LCM	Least Common Multiple
LDPC	Low Density Parity Check
LE	Logic Element
LEO	Low Earth Orbit
LFSR	Linear Feedback Shift Register
LLR	Log Likelihood Ratio
LUT	Lookup Table
MEO	Medium Earth Orbit
MPA	Message Passing Algorithm
MS	Minimum-Sum
OBC	On-Board Computer
OS	Operating System
OSI	Open Systems Interconnection
PC	Personal Computer
PDF	Probability Distribution Function
PDF	Probability Density Function
POSIX	Portable Operating System Interface for Unix
QC	Quasi-Cyclic
QPSK	Quadrature Phase Shift Keying
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RF	Radio Frequency
RTT	Round Trip Time
RX	Receive
SAA	Steerable Antenna Array
SCPS-TP	Space Communications Protocol Specification Transport Protocol
SCSS	Satellite Communication Software System

Stellenbosch University http://scholar.sun.ac.za

SDR	Software Defined Radio
SP	Sum-Product
SNR	Signal-to-Noise Ratio
TC	Telecommand
TCP	Transmission Control Protocol
TM	Telemetry
TX	Transmit
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language
VNU	Variable Node Update
XOR	Exclusive-OR

Stellenbosch University http://scholar.sun.ac.za

Nomenclature

Greek Letters:

σ	Standard deviation
σ^2	Variance
ρ	Probability density function
ω_r	Row weight
ω_c	Column weight

Matrices and Vectors:

Matrices and vectors will always describe the following unless otherwise specified.

$\mathbf{G}$	Generator matrix
$\mathbf{H}$	Parity check matrix
$\mathbf{I}$	Identity matrix
$\mathbf{P}$	Parity matrix
$\mathbf{c}$	Codeword vector
c(X)	BCH codeword vector in polynomial format
d(X)	BCH message vector in polynomial format
e(X)	BCH error vector in polynomial format
g(X)	BCH generator polynomial
$\mathbf{s}$	Syndrome vector
$\mathbf{r}$	Remainder after division
r(X)	BCH remainder after division in polynomial format
s(X)	BCH syndrome vector in polynomial format
$\mathbf{x}$	Message vector
ϕ(X)	BCH minimal polynomial

Stellenbosch University http://scholar.sun.ac.za

Subscripts and Superscripts:

Subscripts and superscripts will always describe the following unless otherwise specified.

i	Parity matrix column index
j	Parity matrix row index
T	Matrix transpose

Units:

bps	bits per second
dB	Decibel
Hz	Hertz
m	meter
s	second
W	Watt

Variables:

Variables will always describe the following unless otherwise specified.

A	Amplitude
C_{cap}	Channel capacity
E_b	Bit energy
E_s	Symbol energy
$\mathbf{I}$	In-phase axis
k	Message length
L_{ji}	LLR check node to variable node message
N_o	Noise spectral density
n	Codeword length
$\mathbf{Q}$	Quadrature axis
q_{ij}	Variable node to check node message
R	Code rate
r_{ji}	Check node to variable node message
t	Number of correctable errors by BCH
Z_{ij}	LLR variable node to check node message

Stellenbosch University http://scholar.sun.ac.za

Chapter 1

Introduction

1.1 Background

Agricultural institutions are often required to track changes in the physical environment, on a regular basis. These include air temperature, ground moisture levels and many more. Sensor stations collecting this data are sometimes situated in very remote areas, making it difficult to reach by foot or vehicle. Large areas might also have too much sensor data to collect manually. A possible solution is to deploy a telemetry system that wirelessly gathers data from sensor array stations. Collected data will then be routed to a central station for further processing. This technique is known as remote sensing and can be implemented via either a terrestrial network or a satellite system.

A micro-satellite network can offer a significant number of advantages over a terrestrial network, amongst others offering better coverage over a large area. It can be controlled through a single operator and offers a low cost of adding additional ground stations to the network [5]. Satellites operate at a number of different orbital patterns around the earth. These include Low Earth Orbit (LEO), Medium Earth Orbit (MEO) and Geostationary Earth Orbit (GEO). Typical altitudes for LEO, MEO and GEO are 500-1000 km, 10000 km and 35786 km respectively [6]. Most telemetry and communication satellites operate in LEO. Transmit power requirements are the lowest here, making it ideal for relatively cheap communication satellites. However, a LEO satellite's orbital period is typically shorter than those from other satellites. This results in a smaller communications time window with a particular ground station, hence communication have to be efficient.

Communication quality is typically determined by Doppler frequency shift effects and low signal-to-noise ratios (SNRs) at the satellite's receiver. It would therefore be highly desirable to include technologies that could circumvent these problems at a minimum cost. As part of a project known as the In-Situ Hyper-Spectral (IS-HS) 2, a micro-satellite platform has been developed that addresses the aforementioned communication problems. It is primarily

Stellenbosch University http://scholar.sun.ac.za

intended for use in agricultural research, but not limited to that. The satellite will gather in-situ sensor data from a ground station as well as hyper-spectral imagery of the area. Collected data will then be downloaded to a central server station for further processing.

This project is a collaboration between Stellenbosch University (SU) and the Katholieke Universiteit van Leuven (KUL). SU designed the digital signal processing part of the system as well as all software and hardware, except for the steerable antenna array (SAA). Innovative technologies such as a software defined radio (SDR) modem and channel coding, which includes forward error correction (FEC), are present in the design. The SDR actively changes its demodulation frequency to compensate for Doppler frequency shift while the FEC ensures reliable communication at very low SNRs. KUL designed a SAA that ensures maximum signal gain at the receiver of the satellite. The antenna's angle of maximum gain is constantly directed towards the ground station as the satellite passes over it. Using this in conjunction with FEC allows for efficient usage of the time limited communications window.

Feasibility studies and prototypes for all technology to be used on the IS-HS 2 satellite, have been completed. This thesis will focus on implementing all the components necessary to facilitate reliable data exchange between a ground station and the satellite platform. These components were integrated with existing subsystems to create a functioning demonstration platform.

1.2 Motivation for Work

An eventual flight model for the project has to be preceded by a fully functional engineering model and it is around the latter that the work encompassed by this project, has been centred.

At commencement of the project, systems integration of the IS-HS 2 configuration was in progress, but incomplete. The SDR has been implemented on a digital signal processor (DSP) development board. Software such as the satellite communication software system (SCSS) have also been completed. The SCSS schedules communication with a particular ground station after which data is exchanged by file transfer. No means existed to facilitate the file transfer and had to be implemented.

Although a communications protocol adhering to the OSI standards has been basically selected initially, the individual layers and overall implementation were outstanding. Allowance for integration of specific FEC schemes, such as LDPC into the structure, was also still required.

Channel coding, especially the FEC, is computationally too expensive to run on the satellite's OBC. Therefore, it was proposed to be implemented on a field programmable gate array (FPGA) connected to the OBC. Fast, or parallel, designs use vast quantities of logic, while slower serial designs tend to be more compact. Given the FPGAs selected for this project, a trade-off

Stellenbosch University http://scholar.sun.ac.za

between speed and complexity had to be made since other components also have to fit onto the FPGA.

1.3 Project Objectives

As reliable file transfer between the satellite platform and ground station is an essential requirement, the objectives of this work documented herein, were defined as follows :

- Implement an efficient file transfer protocol between the ground- and flight based hosts.
- Ensure compatibility and portability of the relevant protocol layers between two different operating systems (OS). A platform independent design was required.
- Layers of implemented communication modules conform to OSI specifications.
- Design and implementation of data processing routines for each corresponding layer of the OSI model.
- Ensure reliability of file transfer, by implementing firmware (FPGA) based channel coding.
- Choosing a suitable block length for FEC.
- Choice of a FEC scheme based on the findings in earlier work [1].
- A verification technique that the implemented FEC is performing as expected.
- Ensuring complete and stable integration with all other system components.
- Practical field testing of the complete system.

1.4 Project Contributions and Summary

The following contributions specific to the project have been made :

- Development of a POSIX compliant inter process communication (IPC) message passing library for Linux Ubuntu. It hides OS specific IPC implementation details and allows for bidirectional communication between protocol layers.

Stellenbosch University http://scholar.sun.ac.za

- Protocol layers in software are dynamically executable. This allows for new OSI layers to be easily added for experimentation purposes.
- Synthesisable architectures for Bose Chaudhuri and Hoquenghem (BCH) and low-density parity-check (LDPC) FEC have been developed in VHDL.
- Architecture used for LDPC is configurable for speed and complexity trade-offs.
- A testing procedure has been devised using Matlab, that confirms FEC performance results on the FPGA, for a particular FEC scheme.
- Integrating all software protocol layers and channel coding modules into the final prototype for the IS-HS 2.
- Confirmation of reliable file transfer between the satellite - and ground station platforms.

1.5 Outline of Thesis

Chapter 2 provides a short overview of work that has been previously completed for the IS-HS 2. Some background regarding FEC and communication protocols are also given. Chapter 3 presents design details for each component implemented in this project. A BCH code capable of correcting three random bits, is chosen for initial implementation. A decoder is designed for LDPC using the hardware friendly minimum-sum (MS) decoding algorithm. Frame and packet structures for the protocol layers are also given. Finally, a simulation strategy using a C based test application and FPGA is described, which will compare the performance between Matlab and hardware FEC implementations. Chapter 4 provides a flexible channel coding FPGA module architecture, which allows for easy removal or addition of new modules. The layouts for both BCH and LDPC modules, are also given. Packet and frame processing routines are presented as flow charts for all protocol layers in software. Chapter 5 confirms the findings of [1], that LDPC outperforms BCH. It is also found that the BCH design from Chapter 3 is adequate for this implementation. A demonstration of the final system at KUL in Belgium proved that the final system is capable of transferring files reliably between a ground station and the satellite platform. Lastly, Chapter 6 concludes the results of this work and makes recommendations in terms of the current and next generation designs.

Stellenbosch University http://scholar.sun.ac.za

Chapter 2

Previous Work and Literature Review

The constituent components present in the communication system of IS-HS 2 are block diagrammatically presented in Fig. 2.1. Key concepts necessary to understand the design and implementation of components A,B,F and G will be discussed in this chapter. Properties of D, the physical communications channel, as required for bit error probability analysis, are also dealt with. The next section illustrates the intended interaction between satellite and ground station platforms. This is followed by a short overview of components previously implemented for the IS-HS 2 project.

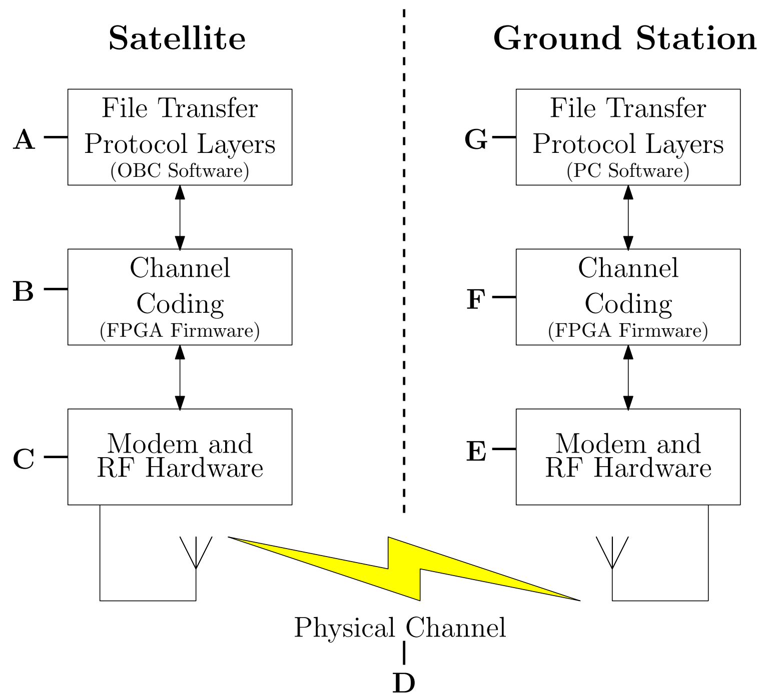


Figure 2.1: Communications channel block diagram.

Stellenbosch University http://scholar.sun.ac.za

2.1 IS-HS 2 Theory of Operation

Fig. 2.2 shows an example of a scenario where the satellite interacts with sensor ground stations $GS1$ and $GS2$. Sensor data is uploaded when the satellite passes over $GS1$. The SAA stays directed towards $GS1$ during this transaction. Upon completion, $GS2$ will be scheduled for communication where the SAA is redirected towards its position. After collecting all ground station information, it is downloaded to a central ground station where this data will be processed.

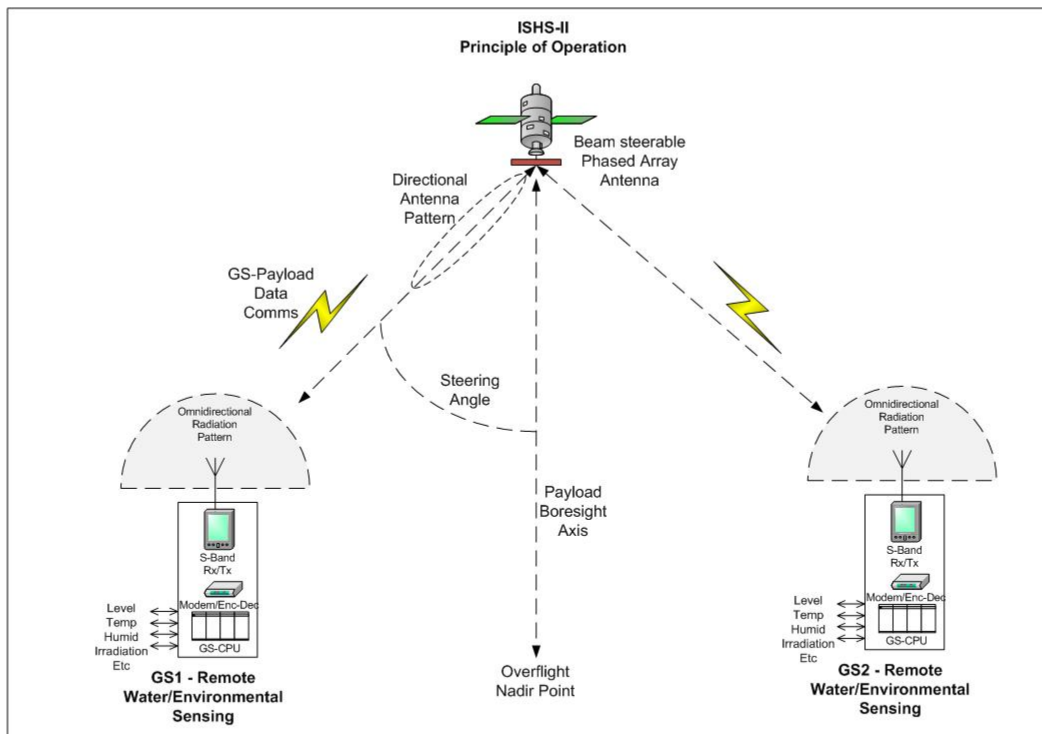


Figure 2.2: Intended interaction between the satellite and ground station platforms.

2.2 Existing Work on IS-HS 2

The architectures of both ground station and satellite platforms, are shown in Fig. 2.3. The system consists of an uplink and a downlink, allowing bidirectional communication between the satellite and a ground station. The uplink uses quadrature phase shift keying (QPSK) modulation and has a throughput of 19200 baud or 38400 bps. Commercial off-the-shelf (COTS) radios are used for the downlink, which operate at 115200 bps. All IS-HS 2 technologies relevant to communication, including the SDR, SAA and channel coding, are used on the uplink. Therefore, the uplink will serve as an evaluation platform for these technologies in the current project. Eventually, the flight model of this

Stellenbosch University http://scholar.sun.ac.za

satellite prototype will implement the same SDR, SAA and channel coding technologies on the downlink as well.

Green component blocks indicate the components as implemented in this work. Red component blocks were also implemented, but not as part of this thesis. Dashed connector lines with arrow heads indicate the different communication paths and directions.

2.2.1 Satellite Platform

The satellite platform in Fig. 2.3 consists of 5 major components :

- An aircraft OBC that supplies avionics information.
- A PC running aircraft satellite emulator (ASE) software.
- The IS-HS 2 communications payload.
- An uplink steerable antenna array (SAA) for receiving data from the ground station.
- A downlink radio which transmits data to the ground station.

In order to verify that all subsystems of the IS-HS 2 project are working correctly, a test involving a real satellite passing over a ground station would be required. Since this cannot be done in the engineering model, it has to be simulated. Mounting the satellite platform on an aircraft and following a particular flight path over a ground station would be the most realistic simulation, as proposed by [7]. The ASE along with the aircraft avionics OBC forms part of this simulation strategy. Parameters such as ground station GPS coordinates and the satellite's simulated LEO altitude can be entered into the ASE. A flight plan for the aircraft is then generated. This plan includes the aircraft's flight path, air speed as well as its altitude above sea level. Adherence to this flight plan emulates a real satellite passing over a ground station at an altitude of approximately 600 km. While this is still very much a viable option, such a test was not finally implemented, due to peripheral project timelines and constraints. Final testing was ground based, as covered in later sections.

The IS-HS 2 communications payload contains an OBC, SDR modem running on a DSP, FPGA which performs general data marshalling and channel coding processing, as well as the RF modules. These components initiate and control communications over the satellite link.

The OBC is a Sun Space and Information Systems design. The South-African designed Sumbandila satellite launched in 2008 also uses this OBC. It has a SH7750R CPU based on the Renesas SH-4 family of 32-bit RISC architectures [8]. Other features include 8 MB of S-RAM and a CAN bus for reliable communication with external components. This OBC will henceforth be referred to as the SH4. The SH4 runs the Unix based QNX OS. Programs

Stellenbosch University http://scholar.sun.ac.za

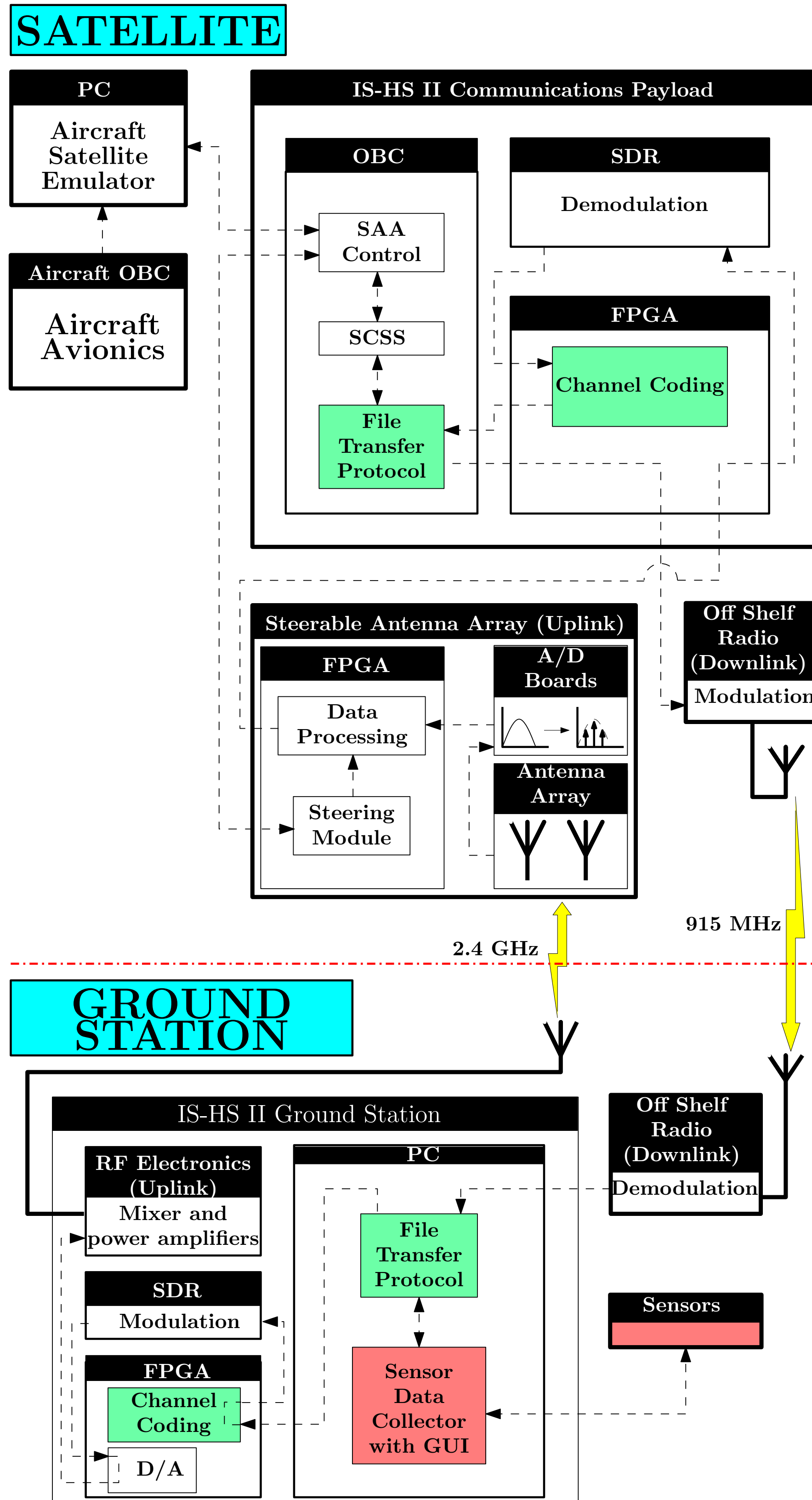


Figure 2.3: Block diagram of both ground station and satellite platforms.

Stellenbosch University http://scholar.sun.ac.za

running under this OS, will be referred to as processes. The SAA control, SCSS and file transfer protocols are processes necessary for communications.

The SAA control process regularly polls the ASE for updated aircraft avionics information. After receiving this data, a new steering angle for the SAA is calculated. This angle is then written to the SAA's FPGA which in turn steers the antenna. Communication with a particular ground station is scheduled by the SCSS process. It initiates a communications transaction, based on the avionics information it receives from the SAA control process. After sensor data has been uploaded, the next ground station is placed on its schedule.

Information exchange between a ground station and the SCSS happens in the form of files. File transfer protocols must segment these files into smaller packets before transmitting it over the link. Once a packet has been sent, the ground station has to acknowledge its reception. After receiving an acknowledge the next packet is transmitted, otherwise the current packet will be retransmitted. This procedure will repeat until the complete file has been transferred.

Base band QPSK data received from the SAA is conveyed by the FPGA to the SDR for demodulation. The FPGA then routes demodulated data from the SDR to the channel coding modules. After successful decoding, data is sent to the protocol layers on the SH4. Should data be corrupted and irrecoverable by the FEC, it gets rejected on the FPGA.

An off-the-shelf data radio that operates in the 915 MHz license free industrial scientific and medical (ISM) band is used for the satellite's downlink. It accepts data from a UART connected to the FPGA, after which it gets transmitted to the ground station. No additional channel coding are required since these radios already implement error control schemes. The SAA constructed by KUL, operates in the 2.4 GHz ISM band and consists of a four by four array of circularly polarised antennas, called elements. It mixes the 2.4 GHz signal down to base band QPSK before getting sampled by the A/D boards. A signal received by one element, only differs in phase from the others. These phases are manipulated on the FPGA before summing it all together. Correctly manipulating these phases leads to an optimal signal angle at all times. This processing scheme effectively beam steers the antenna.

2.2.2 Ground Station Platform

An IS-HS 2 ground station platform contains the following components :

- A downlink radio which receives data from the satellite.
- Sensors that collect agricultural - or similar types of data.
- A PC for storing sensor data and controlling communication.

Stellenbosch University http://scholar.sun.ac.za

- FPGA for channel coding and data marshalling between the SDR and RF electronics.
- A SDR for QPSK modulation.
- RF electronics including a quadrature mixer and power amplifiers.

Central to the operation of the ground station is the PC. Known as a FIT-PC, its compact design allows for constructing a small and energy efficient ground station. It hosts the Linux Ubuntu 7.10 OS. Processes that will run on this OS include the file transfer protocol and the sensor data collector. Sensor data to be uploaded to the satellite are generated by the sensor data collector process. Once implemented, this process will contain a graphical user interface (GUI) that enables a user to access collected information. This includes sensor measurements and link information such as time stamps of the last satellite pass.

Data coming from the file transfer protocol are sent to the FPGA. Here the channel coding module adds redundancy for the FEC. Data are then forwarded to the SDR for QPSK modulation. Modulated data are mixed up to 2.4 GHz by the RF electronics before being amplified. Finally, the 2.4 GHz signal is transmitted over the link to the satellite. A downlink radio, similar to the one used on the satellite, receives data from the satellite. It passes data to the file transfer protocols if no errors are present on the received data. The radio will discard data if it contains errors.

2.3 Protocols

Previous work [1] suggested that IS-HS 2 communication protocols should conform to OSI specifications. This standard is maintained by the International Organization for Standardization (ISO) [9]. The OSI model defines communication software i.t.o layers where each layer provides a different service such as end-to-end reliability [9]. Data is transferred from one system to another through interaction between these layers.

Fig. 2.4 shows the OSI layers described by [1]. Descriptions of these layers are given in Table 2.1. Note that the data link layer has been split into two sub sections. This allows any type of FEC to be implemented without having to adopt a new data link layer standard.
The SCSS and SAA control processes on the SH4 forms the application layer. Similarly, the ground station PC has the sensor data collector process on this layer. Hardware such as the SDR and RF electronics creates the physical layer on both satellite and ground station platforms. Transport as well as data link layers still had to be implemented as part of the present work.

A data link layer at the receiving platform guarantees the data it passes to a transport layer, to be error free. Since these erroneous frames are rejected,

Stellenbosch University http://scholar.sun.ac.za

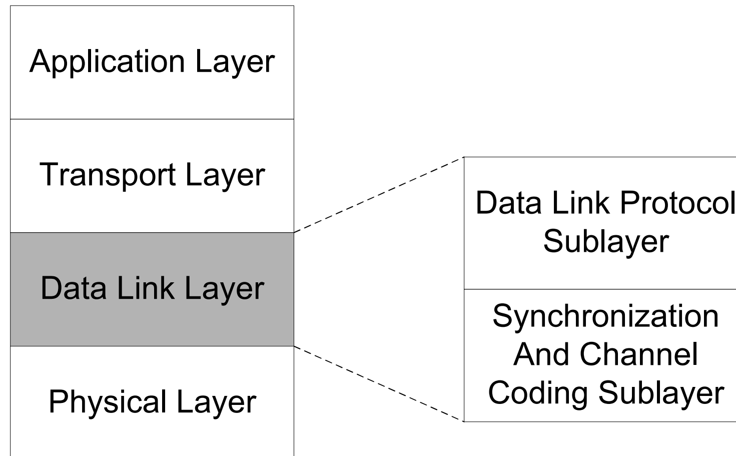


Figure 2.4: Layers of the OSI model [1].

OSI layer	Layer description
Application	User software that controls and initiates communication transactions over the satellite link.
Transport	Ensures end-to-end reliability when transmitting data.
Data Link	Provides error detection and correction services.
Physical	Hardware for transmitting or receiving data on the satellite link.

Table 2.1: Description of OSI layers in Fig. 2.4.

this layer cannot guarantee data to be successfully received every time. At the transmitting platform, a data link layer receives data from a transport layer. Here transport layer data are divided into fixed length data frames to be transmitted sequentially.

The transport layer is responsible for transmitting high level data such as files. It guarantees a sent file to be correctly assembled on the receive side. The transmitting platform breaks a file into packets before sending these to the data link layer. Should a packet get lost due to errors in the data link layer, the transport layer will retransmit this lost packet. An Automatic Repeat reQuest (ARQ) strategy can provide the functionality expected from this layer [1].

2.4 Inter Protocol Layer Communication

The OSI layers from Section 2.3 have to interact with each other. Layers running on the SH4 and FIT-PC will typically use memory resources to communicate with each other. Hardware based layers use electrical interfaces to communicate with adjacent layers. Unlike the electrical interfaces, a software inter layer communication strategy have not yet been designed at project ini-

Stellenbosch University http://scholar.sun.ac.za

tiation.

Different processes on an OS provide unique services such as TCP/IP networking and file system support [10]. Processes may contact each other to request a particular service and exchange data if necessary. This is done via interprocess communication (IPC) facilities provided by the OS.
The OSI layers are regarded as different processes running on an OS. Layers such as the transport layer must be able to send and receive files as discussed in Section 2.3. By using threads, both transmit and receive functionalities can be implemented on a single process. Threads reside within a process and are the most basic units to be scheduled for execution on a CPU [10]. The OS rapidly switches between threads on the CPU, when a process is scheduled to run. This allows transmit and receive functionalities to run simultaneously and independently of each other.

2.4.1 IPC Schemes

2.4.1.1 Shared Memory

In Fig. 2.5 a section of memory is shared between two different processes. Processes 1 and 2 call the OS kernel to map this shared memory region into their separate address spaces. No kernel call is necessary to modify data in this memory, making it the fastest way to exchange large quantities of data between two processes [10]. By using routine memory access procedures to modify data [10], changes are instantly available to the other process.

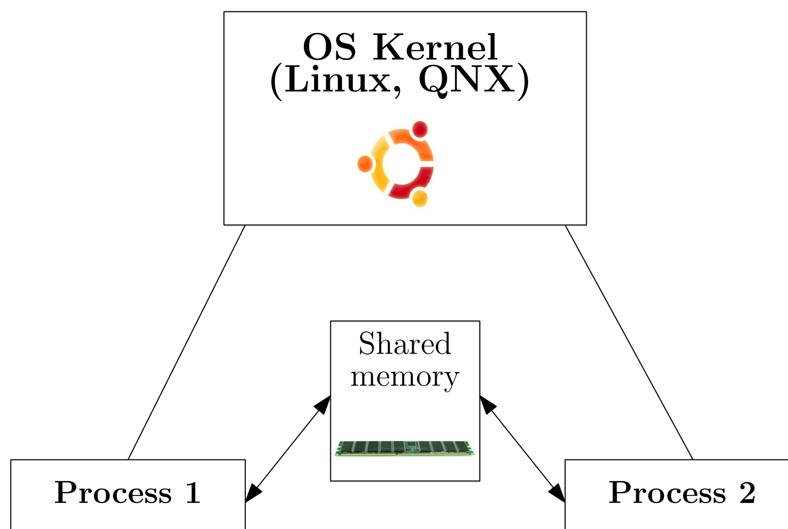


Figure 2.5: Shared memory between 2 processes.

No synchronisation services are provided by default with shared memory. A receiving process should only read data once a sending process is done modi-

Stellenbosch University http://scholar.sun.ac.za

fying the region of interest. Therefore, processes 1 and 2 have to either agree on a concurrent access technique or resort to OS synchronisation services, discussed in Section 2.4.2. Typical shared memory applications include bulk data transfer to a video display device driver [11].

2.4.1.2 Message Passing

Processes communicate by sending and receiving messages via a mailbox. The mailbox is a data structure that allows messages to be placed in or removed from [10]. A mailbox resides either in the OS kernel or as shared memory outside the kernel. A process sends message M to mailbox A by calling $send(A, M)$. The receiving process may call $receive(A)$ to receive this message from mailbox A. Unlike shared memory IPC, an OS kernel can provide a variety of synchronisation services when calling $send()$ and $receive()$ [10]. These services include :

- Blocking send() : The sending process sleeps until the receiving process collects the message. Alternatively the sender could also be unblocked once a receiver sends a reply message.

- Blocking receive() : A receiving process waits for an available message. The OS kernel notifies the receiver to wake up if a message is available.

Message passing is the primary IPC technique used in QNX [11]. It uses a client-server relationship between communicating processes as shown in Fig. 2.6. The client process sends the server process a message by calling $send()$. The client then blocks until it receives a reply message from the server. After processing the received message, the server sends a reply by calling $reply()$. This reply may contain processed data or could just be a notification that message processing is done.

2.4.1.3 Pipes

A pipe is a file with a predetermined size that only exists in memory [12]. Reading and writing operations are performed in memory and not via the file system. Therefore, it acts as a type of shared memory. Pipes only allows one way communication between two processes, hence a pair of pipes are required for bidirectional communication.

The sending process calls $write()$ to add data to one end of the pipe. A process calling $read()$ on the other end, receives data from this pipe. The sender is blocked when a pipe becomes full. Similarly, a receiving process is suspended if no data is available.

Processes sometimes spawn new processes, called child processes. The process that spawns, called a parent, typically communicates with a child by using an unnamed pipe. This pipe is only visible between the parent and child

Stellenbosch University http://scholar.sun.ac.za

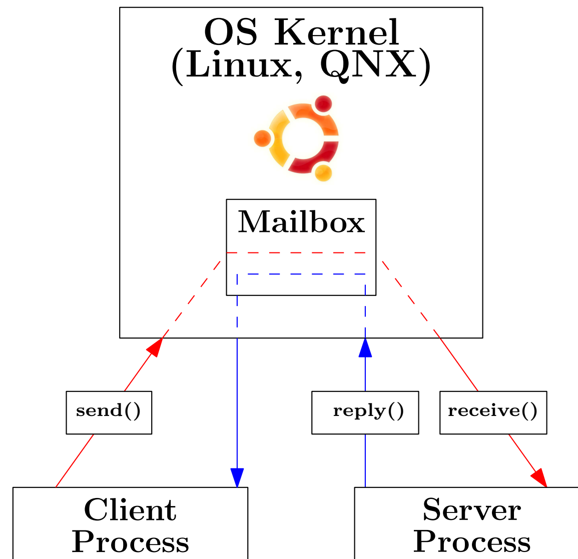


Figure 2.6: QNX message passing between a client and server process.

processes. Operating systems such as Linux uses this communication scheme between a parent and a child [13]. Named pipes, also known as FIFOs, are visible to all processes. A reference to the pipe's memory mapped file is placed as an entry in the file system [13]. Processes may open this file, allowing them to connect to this pipe.

2.4.1.4 Message Queues

A message queue is a linked list of messages that exists inside the OS kernel as shown in Fig. 2.7. The queue uses a FIFO principle : messages M_1 to M_3 are removed in the same order they have been added. Bidirectional communication are allowed between two processes that share the same queue.
Receiving processes call *receive*() to retrieve the first message from the queue. This process is blocked if no messages are available. Similarly, a sending process is blocked if the queue becomes full. It is unblocked once the receiving process removes a message. On the Linux OS, a receiving process can be asynchronously notified of an incoming message [14]. This allows the process to perform other tasks instead of having to regularly poll or wait for incoming messages. Productivity of a process is therefore increased.

2.4.1.5 Sockets

Sockets offer connection orientated communication between two processes. Unlike the other IPC in this chapter, sockets allow processes to communicate over a network. Sockets may also be used between processes on the same computer. These local sockets are known as Unix domain sockets [12].

Stellenbosch University http://scholar.sun.ac.za

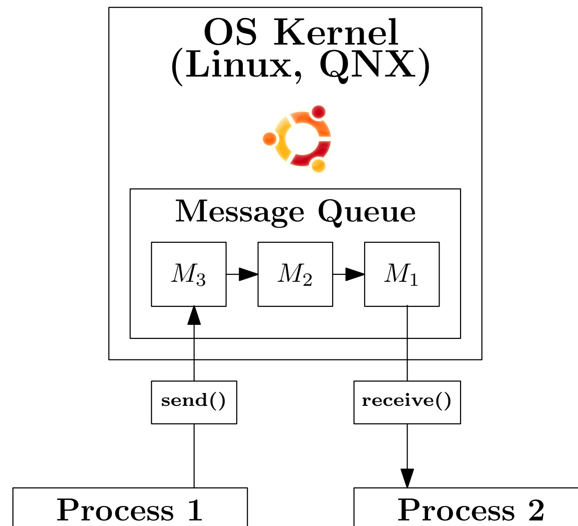


Figure 2.7: A message queue shared between two processes.

Bidirectional communication is possible between two processes. Similar to pipe IPC, data is written to one end of a socket while data is removed from the other end. Memory mapped files buffer data on both communication paths [12]. A client-server model is adopted between communicating processes. The server process creates a socket and listens for incoming connections. Client processes can connect to a server after which a bidirectional connection is established. Many different clients may connect to the same server process.

2.4.1.6 Files

This is the simplest form of IPC between two processes. By default no synchronisation services are provided by the OS to control file access, hence data consistency cannot be guaranteed. Communication happens via disk I/O or in memory through memory mapped files. The latter is faster since modification happens at memory access speeds. By contrast, disk I/O happens at much slower speeds.

File locks control file access while modifying a particular section of the file and are either mandatory or advisory. Mandatory locks deny read and write access to all processes other than the one holding the lock. Advisory locks indicate that the file is being modified but does not deny read and write permissions to the calling process.

2.4.2 Synchronisation Schemes

Shared data have to be accessed concurrently between processes or threads. No other process or thread are allowed to access this data while being modified

Stellenbosch University http://scholar.sun.ac.za

by another process or thread. Operating systems provide a variety of solutions to this synchronisation problem.

2.4.2.1 Semaphores

A semaphore controls access to a shared region by using locks. Processes must request a lock from this semaphore before access is granted to the shared region. This lock is returned after modifying the shared region. Semaphores implement these locks by using an integer [10]. This integer gets initialised to a specific positive value upon creation of the semaphore. Each process requesting a lock, decrements this integer until it reaches zero. Lock requests are denied when this integer is zero, followed by the requesting process being blocked. A blocked processes is waken when another process returns its lock.

Typically a lock is requested by calling *sem_wait*() and is released by calling *sem_signal*(). These functions are executed *atomically*. An *atomic* call guarantees that only one process will modify the semaphore at a time when multiple processes call *sem_wait*() simultaneously [10]. This prevents a situation where two processes obtain a lock whereas only one should have received a lock.

A binary semaphore is a specific implementation that only contains one lock. Linux for example uses binary semaphores to synchronise threads. These are known as *mutex* locks, since they mutually exclude multiple threads from simultaneously accessing a resource protected by this lock [12].

2.4.2.2 Signals

Signals informs a process of an event that is in progress or has just taken place [12]. These events could be external or internal with regard to a process. An internal event could be something such as an illegal memory access attempt [10]. Events like these are synchronous since the process that caused them is signalled immediately. External events cause a signal to be delivered asynchronously to a process. This may happen if some process have modified a file and wishes to inform another process that its modifications are done.

Processes implement routines for handling the different signals delivered to it. Some signals cannot be processed since the handler routines may override its original purpose. Examples include Linux's kill signal - called *SIG_KILL* - which cannot be processed by a signal handling routine [13]. It forces a process to terminate even if some resources owned by the process have not yet been released.

2.4.2.3 File Locks

These locks are critical to synchronise access to a file. A file could be locked using either a lock-file or a system call that associates a lock with an open file's descriptor inside the OS kernel. Lock-files are typically empty files that

Stellenbosch University http://scholar.sun.ac.za

are created alongside the file being modified. Its presence is an indication that another process is busy modifying the desired file [12]. After modification is complete the lock file is removed or unlinked, allowing other processes to recreate a lock-file and modify the same file. The Linux OS creates files atomically, meaning that only one of multiple competing processes will be able to create the lock-file at a time. This ensures concurrent access to the file being modified. By calling $fcntl()$ in QNX and Linux, a mandatory lock is placed on a file that is already open. Either the whole file or a specific section could be locked [12]. This system call is performed atomically.

2.5 Error Control Strategies

Error control is a term referring to both error detection and correction (EDAC) schemes [15]. An error detection scheme can identify whether data is corrupted based on the received information. Corrupted data is discarded before requesting retransmission of the same information. Forward error correction (FEC) attempts to identify and correct all errors in the received information. Failing to do so will result in a retransmission of the same information. Since FEC can identify erroneous data, it also serves as an error detection scheme.

The noisy channel coding theorem has been part of the work done by Claude E. Shannon in 1948 [3]. It states that error free communication is possible over a channel containing Gaussian noise. Specifically, if the rate of information transmission, R_{inf}, is less than the channel's capacity C_{cap} then error free communication is possible. Units of R_{inf} and C_{cap} are both in bits per second (bps).

Error correction adds additional information to a message that needs to be transmitted. This redundancy along with the original message is known as a codeword. Redundancy effectively spreads a message's information across the whole codeword which averages the effect of noise [16]. Burst errors for example, may corrupt a certain location in a codeword. However, that section may be recoverable using the redundancy of the codeword.

Two classes of error correction codes exist, namely convolution codes and linear block codes [3]. A convolution code operates on a continuous stream of data that enters the encoder. Internally it is synchronised to encode k-bit message sections to n-bit codewords. The decoder aligns itself with these n-bit sections before decoding the stream.

Linear block codes operate on fixed length messages. An encoder will wait for k message bits before encoding commences. Similarly the decoder will gather n codeword bits before performing error detection and correction. According to [1], BCH and LDPC linear block codes are to be considered for implementation in the IS-HS 2. The implementation of these linear block codes, formed part of the work as set out in this thesis.

Stellenbosch University http://scholar.sun.ac.za

2.5.1 Linear Block Codes

In linear block codes a k-bit message is encoded to form a n-bit codeword, referred to as a (n, k)-code [17]. A (n, k) code may have up to 2^k different codewords. Only binary codes are considered in this thesis. All addition and multiplication operations are done modulo-2.

Fig. 2.8 illustrates how information is processed by linear block codes. A 4-bit codeword **c** is created by multiplying message **x** with **G**, called the generator matrix. This step appends $(n - k) = 2$ redundant bits to **x** without altering **x**. A codeword having this type of structure is called a systematic code. The code rate $R = (k/n) = 0.5$ indicates how much usable information is contained in a codeword [17]. Noise gets added to **c** being sent across the channel to give **c'**, which may contain errors. Multiplying **c'** with $\mathbf{H}_T$, known as the parity check matrix, gives syndrome **s**. Only an all zero syndrome indicates that no errors are present in **c'**. After successfully correcting all errors in **c'**, the decoder strips this codeword's redundant bits and outputs the original message **x**.

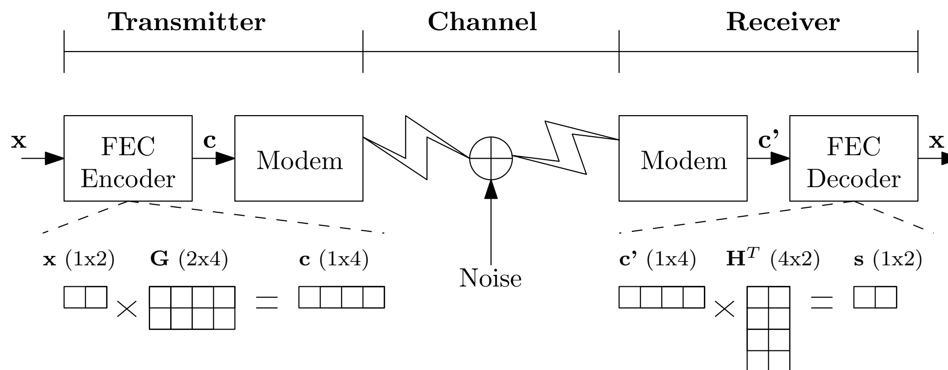


Figure 2.8: A FEC encoder and decoder in a communications channel.

Note that $R_{inf} = R \times C_{cap}$. Therefore, if less redundancy is used in FEC then R_{inf} would be higher as expected. A good FEC code corrects a lot of errors while allowing $R_{inf} \rightarrow C_{cap}$.

Linear combinations of codewords will result in another valid codeword [15]. An encoder can use this property to encode any message using a set of basic codewords. Eq. 2.5.1 shows a four bit message $[x_0, x_1, x_2, x_3]$ being encoded to a seven bit systematic codeword **c** by using basic codewords $\mathbf{c}_0$ to $\mathbf{c}_3$. Note that these basic codewords are all linearly independent.

$$\begin{aligned} \mathbf{c} &= [\ \mathbf{x} \mid \mathbf{p}\ ] \\ &= c_0 + c_1 + c_2 + c_3 \\ &= [\ x_0\ \ 0\ \ 0\ \ 0\ \ p_{00}\ \ p_{01}\ \ p_{02}\ ] + \end{aligned}$$

Stellenbosch University http://scholar.sun.ac.za

$$
\begin{aligned}
& [\ 0 \quad x_1 \quad 0 \quad 0 \quad p_{10} \quad p_{11} \quad p_{12}\ ] \ + \\
& [\ 0 \quad 0 \quad x_2 \quad 0 \quad p_{20} \quad p_{21} \quad p_{22}\ ] \ + \\
& [\ 0 \quad 0 \quad 0 \quad x_3 \quad p_{30} \quad p_{31} \quad p_{32}\ ] \\
= & [\ x_0 \quad x_1 \quad x_2 \quad x_3 \quad p_0 \quad p_1 \quad p_2\ ]
\end{aligned} \tag{2.5.1}
$$

A matrix notation can be used to represent the steps of Eq. 2.5.1. This is illustrated below.

$$
\begin{aligned}
\mathbf{c} &= \begin{bmatrix} x_0 & x_1 & x_2 & x_3 \end{bmatrix} \times \left[\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & p_{00} & p_{01} & p_{02} \\ 0 & 1 & 0 & 0 & p_{10} & p_{11} & p_{12} \\ 0 & 0 & 1 & 0 & p_{20} & p_{21} & p_{22} \\ 0 & 0 & 0 & 1 & p_{30} & p_{31} & p_{32} \end{array}\right] \\
&= \mathbf{x} \times [\ \mathbf{I} \,|\, \mathbf{P}\ ] \\
&= \mathbf{x} \times \mathbf{G}
\end{aligned} \tag{2.5.2}
$$

The matrix $\mathbf{G} = [\ \mathbf{I} \,|\, \mathbf{P}\ ]$ is the same as used in Fig. 2.8. An encoder requires this matrix to transform a message into a codeword. The identity matrix $\mathbf{I}$ ensures that message $\mathbf{x}$ is added unaltered to $\mathbf{c}$. Redundant, or parity bits, p_0 to p_2 are added to $\mathbf{x}$ by parity matrix $\mathbf{P}$.

A parity bit's value indicates whether a sequence of bits have an even or uneven number of ones. In even parity, modulo-2 summation of all these bit values ,including the parity's value, will produce zero. A (7,4) Hamming code's parity is illustrated in Fig. 2.9.

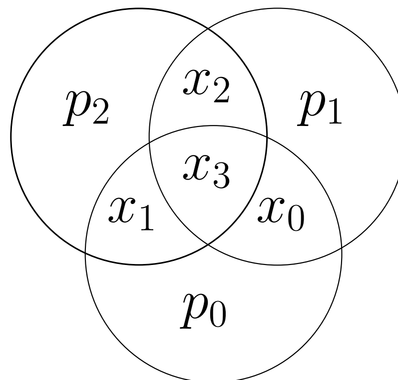


Figure 2.9: A (7,4) Hamming code's parity bit dependency diagram.

Each circle in the diagram contains three data bits and one parity bit. These illustrate which bits of message $\mathbf{x}$ are used to calculate a parity bit's value.

Stellenbosch University http://scholar.sun.ac.za

Parity bits p_0 to p_2 are calculated using Eqs. 2.5.3 to 2.5.5.

$$p_0 = x_0 + x_1 + x_3 \tag{2.5.3}$$
$$p_1 = x_0 + x_2 + x_3 \tag{2.5.4}$$
$$p_2 = x_1 + x_2 + x_3 \tag{2.5.5}$$

After sending a codeword across the channel, the FEC decoder receives a codeword **c'** $= [x_0' \quad x_1' \quad x_2' \quad x_3' \quad p_0' \quad p_1' \quad p_2']$ that may contain errors. Continuing the (7,4) Hamming example, the decoder evaluates the following equations :

$$s_0 = p_0' + x_0' + x_1' + x_3' \tag{2.5.6}$$
$$s_1 = p_1' + x_0' + x_2' + x_3' \tag{2.5.7}$$
$$s_2 = p_2' + x_1' + x_2' + x_3' \tag{2.5.8}$$

Eqs. 2.5.6 to 2.5.8 are known as parity check equations. Evaluation of these equations can be presented in matrix form as shown below.

$$\begin{aligned}
\mathbf{s} &= \begin{bmatrix} s_0 & s_1 & s_2 \end{bmatrix} \\
&= \mathbf{c'} \times \left[\begin{array}{ccc} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \\
&= \mathbf{c'} \times \left[\begin{array}{c} \mathbf{P} \\ \hline \mathbf{I} \end{array}\right] \\
&= \mathbf{c'} \times \mathbf{H}^T
\end{aligned} \tag{2.5.9}$$

Matrix **H** is the parity check matrix used in Fig. 2.8. A special property of linear block codes is that $\mathbf{G} \times \mathbf{H}^T = 0$ [15]. Since this is true, matrices **P** and **I** of Eq. 2.5.9 are the same as used in Eq. 2.5.2.

As mentioned before, vector **s** is the syndrome. A non-zero element's position in **s** indicates which parity check equation failed. Unfortunately, the syndrome provides no information regarding the exact location of a bit error in the codeword. In codes such as BCH, the syndrome requires further processing by the decoder to determine error locations [3].

Stellenbosch University http://scholar.sun.ac.za

2.5.2 BCH Codes

Named after inventors Bose, Chaudhuri and Hocquenghem (BCH), this code forms part of a powerful class of random error correcting codes [3]. First to invent BCH was Hocquenhem in 1959. Independent from his result, Bose and Chaudhuri also published their work regarding the code's design. Unlike the single error correcting Hamming code, BCH can specify the amount of correctable errors, t, when implementing a (n, k) code [15]. A Hamming code can therefore be seen as a single error correcting BCH with $t = 1$. Numerous communication standards includes BCH error correction. Among these are DVB-S2 [18], a digital satellite television standard, and the TC protocol used in space telemetry systems [19]. BCH codewords are viewed as polynomials. Each bit from $\mathbf{c} = [c_{n-1}, \cdots, c_0]$ represents the coefficient of a term in Eq. 2.5.10.

$$c(X) = c_{n-1}X^{n-1} + \cdots + c_1X^1 + c_0 \tag{2.5.10}$$

These codewords are also cyclic. A cyclic property allows creation of a new codeword, denoted $c_{new}(X)$, by rotating an existing codeword's elements. As an example, Eq. 2.5.10 has been rotated to the left by one bit :

$$c_{new}(X) = c_{n-2}X^{n-1} + \cdots + c_0X^1 + c_{n-1} \tag{2.5.11}$$

Cyclic codes also allow its codewords to be created by using a generator polynomial as shown in Eq. 2.5.12. The k-bit message to be encoded is represented by $d(X)$, which has order $k - 1$. Polynomial $g(X)$ is the generator of order $n - k$.

$$c(X) = d(X) \times g(X) \tag{2.5.12}$$

A BCH encoder applies either generator matrix $\mathbf{G}$ or polynomial $g(X)$ to encode a k-bit message $\mathbf{x}$. Note that Eq. 2.5.12 can be used to construct $\mathbf{G}$ from Eq. 2.5.2. Both $g(X)$ and $\mathbf{G}$ generate the same systematic codewords [17]. After computing the syndrome at the decoder, error locations in $c(X)$ are determined by using either linear algebra or iterative decoding techniques. The first BCH decoding algorithm has been introduced by Peterson in 1960 [3]. It involves solving a set of linear equations to identify error positions in $c(X)$. This technique becomes time consuming when using large codewords and results in a highly complex decoder [15]. Berlekamp and Massey later developed an iterative decoding technique that finds a polynomial $\sigma(X)$ of which the roots can be used to locate the errors in $c(X)$. This is computationally less expensive than Peterson's solution [3].

Stellenbosch University http://scholar.sun.ac.za

2.5.3 LDPC Codes

LDPC codes have originally been invented by Robert Gallager during the 1960's as part of the work done for his PhD [20]. However, a cost effective implementation of the codes was not possible at that stage due to limited computer technologies and the high complexity of Gallager's decoding algorithm [21]. After being forgotten for a few decades, the codes were accidentally rediscovered by MacKay and Neal [22] and Wiberg [23]. Here MacKay showed that LDPC can achieve near Shannon limit performance when using an optimized iterative decoding technique [21].

LDPC is a very powerful FEC code and can be found in numerous communications standards such as DVB-S2 [18] and IEEE 802.11n Wi-Fi [24]. Turbo Codes, another powerful FEC code and strong competitor to LDPC, can also be found in other modern communications standards such as DVB-RSC and 802.16e WiMAX [25]. What makes LDPC attractive compared to Turbo Codes, is that there are no patent issues surrounding the code [26]. Other advantages include better complexity-performance trade-off options [25], lower decoding complexity and very low error floors at low bit error rates (BER) [21].

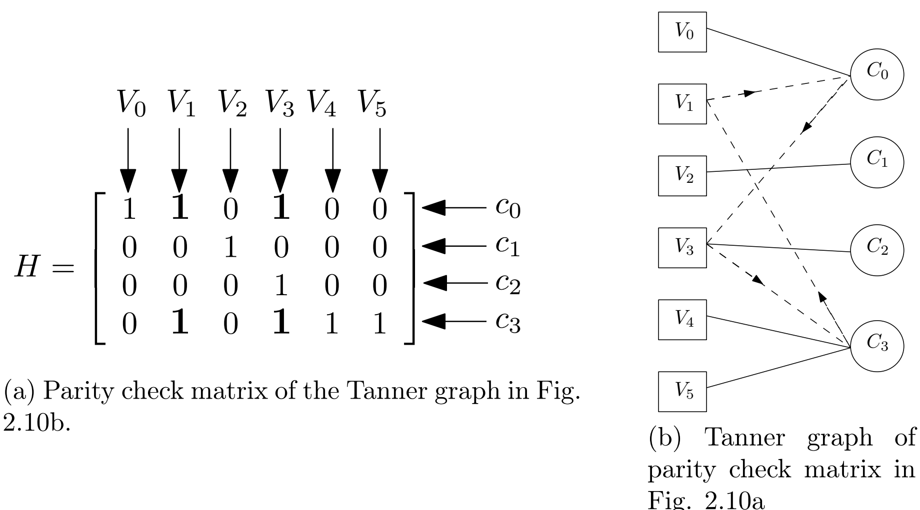
LDPC also falls in the category of linear block codes. Characteristic to LDPC is its sparse **H** matrix, hence the term low density parity check. A sparse matrix have less non-zero elements than zeros. **H** is described by both its column weight ω_c and row weight ω_r. The weight of a vector refers to the number of non-zero entries it contains, i.e. the number of ones contained in a binary vector. A (ω_c, ω_r)-regular LDPC code has the same ω_c for all its columns and the same ω_r for all its rows. An irregular code have different ω_c's and ω_r's for some or all of its columns and rows.

Matrix **H** is visually presented by a Tanner graph [27] shown in Fig. 2.10. The circles are known as check nodes which represent a parity check equation, and hence a row in **H**. The squares are called variable nodes and represent the columns of **H**. Column positions coincide with bit positions of the received codeword **c**'. Whenever H_{ij}=1 the associated variable node and check node of row i and column j are joined by a line, called an edge.

In general the BER performance of a LDPC code is governed by the length of its codewords as well as the techniques used to construct **H**. The minimum Hamming distance d_{min} increases as the codeword's length increases [15]. Hamming distance refers to the number of bits by which two codewords differ. Increasing this distance improves the error correction capabilities of a code. Constructing **H** to be as random as possible, delivers good BER results [15], but increases decoding complexity. This is due to lots of information regarding **H** being stored in memory. By using a structured code such as quasi-cyclic (QC) LDPC, lowers decoding complexity, but reduces BER performance [28].

Another important property of **H** that limits decoder performance is girth [27]. Starting at any check node or variable node in the Tanner graph, girth

Stellenbosch University http://scholar.sun.ac.za



(a) Parity check matrix of the Tanner graph in Fig. 2.10b.

(b) Tanner graph of parity check matrix in Fig. 2.10a

Figure 2.10: Visual representation of a parity check matrix.

is defined as the minimum number of edges to be traversed to reach the same starting node, without traversing any edge more than once. This is indicated by the dashed lines in Fig. 2.10b. Starting at V_1 , 4 lines must be traversed to reach V_1 again thus giving this code a girth of 4. This cycle can also be seen in **H**. Whenever two non-zero elements in two different columns are in the same two rows, a length 4 cycle is present. This is indicated by the bold ones in Fig. 2.10a. The presence of length 4 cycles severely deteriorates performance of the decoder, so that it takes more iterations to find the correct codeword [27].

2.5.3.1 Decoding

A LDPC decoder computes the syndrome $\mathbf{s} = \mathbf{c}'\mathbf{H}^T$ where **c**' is the received codeword with errors. Only when $\mathbf{s} \neq 0$, a decoding cycle is started to correct the errors in **c**'. Note that the syndrome is used here only as an error detection method. Decoding achieves the best BER performance when using an iterative message passing algorithm (MPA) [1]. These messages are either log-likelihood values or probability values exchanged between check nodes and variable nodes during an iteration. The MPA decoder's architecture imitates the structure of a Tanner graph [29]. An iteration begins with each variable node passing a message to its connected check nodes. This is followed by each check node passing a message to all its connected variable nodes. Note that messages travel along the Tanner's edges between connected nodes. Messages arriving at each variable node v_j for $0 \leq j \leq n$ are now used to modify the corresponding bit c'_j of codeword **c'** to form $\mathbf{c}_{new}$. The final step in an iteration recomputes the syndrome by using $\mathbf{s} = \mathbf{c}_{new}\mathbf{H}$. Decoding stops when $\mathbf{s} = 0$, otherwise a new iteration is started. This process continues until $\mathbf{s} = 0$ or when a predefined maximum number of iterations are reached.

Stellenbosch University http://scholar.sun.ac.za

2.5.3.2 Encoding

Encoding can be done using a generator matrix **G**. Although **H** is sparse, matrix **G** will not necessarily be sparse [15]. Therefore, encoding might have a time complexity of $O(n^2)$ with n the length of a codeword. However, another technique exists that lowers this complexity to $O(n)$ [30]. Section 3.4.3 provides more detail about this technique.

2.6 Wireless Channels

Various parameters are associated with a wireless communications channel. The most important to consider in a design are noise sources, multipath effects, transmit power and sources of signal attenuation. Taking these into account, a few concepts will now be explained that are necessary for error probability analysis of data sent over a wireless channel.

2.6.1 Link Margin

A link budget calculation is the first step in designing any wireless communications system. Important decisions regarding transmitter power and receiver sensitivity are made here. Fig. 2.11 shows a typical setup of both a transmitter and receiver communicating over a wireless channel.

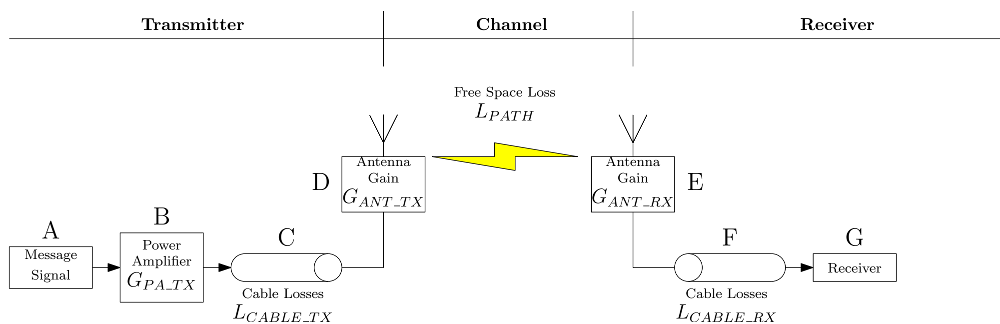


Figure 2.11: A transmitter and receiver communicating over a wireless channel.

The transmitter sends a signal from A to amplifier B. This signal continues through cable C with some loss after which it reaches antenna D. Depending how directional the antenna is, more gain is added to the transmit path. After losing most of its power over the channel, the signal reaches a receiving station. The antenna at E also adds some gain in the receive path. After experiencing more loss through RF cabling at F, the signal reaches the receiver at G. A signal being generated by A has unity power. Expressing all losses and gains in terms of decibels (dB), the signal power in dB reaching G can be expressed as :

Stellenbosch University http://scholar.sun.ac.za

$$\begin{aligned} P_G = & G_{PA_TX} - L_{CABLE_TX} + G_{ANT_TX} - L_{PATH} \\ & + G_{ANT_RX} - L_{CABLE_RX} \end{aligned} \tag{2.6.1}$$

Receivers typically have a lower bound on acceptable input signal levels, known as its sensitivity [31]. This value specifically accounts for thermal noise from the antenna and noise added by each amplifier in front of the receiver. Sensitivity thus specifies the minimum acceptable power level, after the antenna, of a received signal. Signals below this value will disappear into the noise floor of the receiver. Assuming that P_G given in Eq. 2.6.1 is greater than this sensitivity, the following term is formed :

$$P_{LINK} = P_G - P_{SENSITIVITY_G} \tag{2.6.2}$$

The term P_{LINK} is known as the link margin [32]. Since the receiver's sensitivity is equal to its noise floor, P_{LINK} can be seen as a signal-to-noise ratio (SNR). This SNR forms the lower bound on the SNR at which a FEC code must be able to deliver a low BER.

2.6.2 Channel Error Probability Model

A channel model mathematically describes the effects of disturbances such as noise on a transmitted signal [15]. Since these disturbances affects random segments of transmitted information, a statistical model is applied to each bit being transmitted. This model applies a certain weight to a bit's chance of being received correctly or incorrectly. In Fig. 2.12 a channel model encapsulates modem and RF components as well as the wireless channel. The FEC encoder inputs bits into this model. Bits are then flipped according a chosen statistical model after which bits are output to the FEC decoder. A few channel models are considered below.

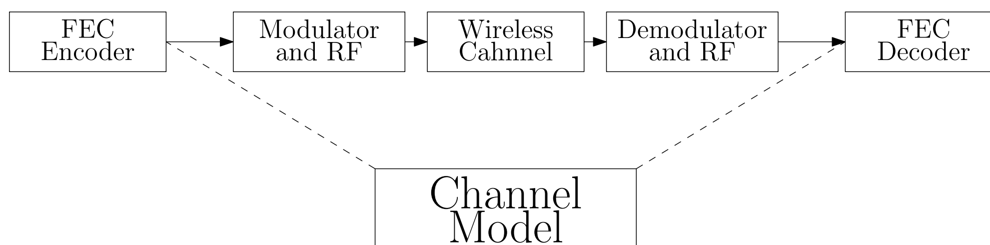


Figure 2.12: Section of a communications channel included in a data error probability model.

Stellenbosch University http://scholar.sun.ac.za

2.6.2.1 Binary Symmetric Channel

A binary symmetric channel (BSC) is shown in Fig. 2.13. Bits being transmitted move from left to right along the routes of the arrowed lines. A bit being sent has probability P_e of being changed. This is known as a crossover probability and is represented by the diagonal lines. In a binary channel the probability of successful transmission is $P_s = 1 - P_e$ as indicated by the horizontal lines. Error probabilities for both a 1 and 0 are the same. This model assumes that P_e is always the same for a certain channel.

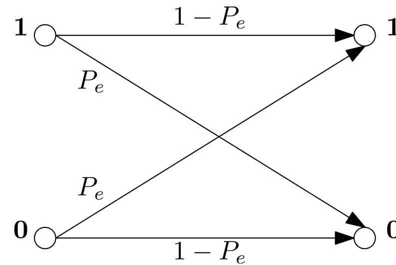


Figure 2.13: A BSC model.

2.6.2.2 Binary Erasure Channel

The binary erasure channel (BEC) allows bits to be received either correctly or as unknown. The demodulator marks a bit as erased if it is unsure whether a 1 or a 0 has been received. Erasures are marked as E in Fig. 2.14. Probability of an erasure is indicated as P_e. Similar to a BSC, the BEC model assumes a constant P_e for a channel. A SDR from this project outputs demodulated data according to the phase difference between subsequent received QPSK symbols. Demodulated data is never marked as unknown by the SDR, hence the BEC model will not be used in this thesis.

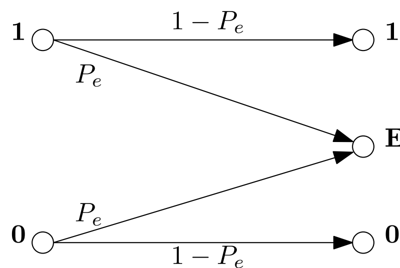


Figure 2.14: A BEC model.

Stellenbosch University http://scholar.sun.ac.za

2.6.2.3 Additive White Gaussian Noise Channel

Noise present on devices such as antennas are typically thermal noise of which the behaviour is modelled by a Gaussian random process [33]. An additive white Gaussian noise (AWGN) channel model gets noise values from a Gaussian random process [34]. Combining a BSC and the AWGN model is possible [15]. The BSC allows a bit to be in only one of two states while the AWGN provides an accurate description of the channel noise. This hybrid model replaces P_e of the BSC with a value from a Gaussian probability distribution function (PDF) having standard deviation σ. This hybrid model is called a binary AWGN (BI-AWGN) [34] and will be used in the rest of this thesis.

Binary AWGN channels map bit values 1 and 0 to values 1 and -1 respectively in Fig. 2.15. A Gaussian PDF with standard deviation σ is placed over each bit. At the thick vertical line between the PDFs, the demodulator decides between a 1 or a 0. Adding sufficient noise to bit -1 such that the decision point is crossed, the demodulator could interpret it as a 1. The converse is also true.

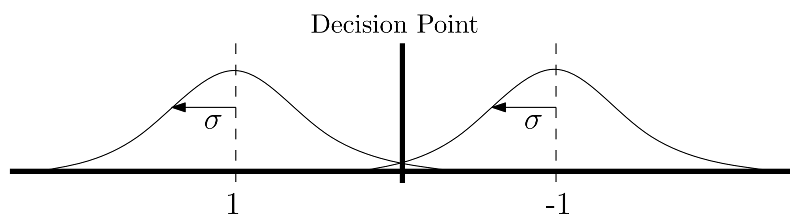


Figure 2.15: A BI-AWGN channel model.

2.7 Summary

A description of the interaction between a satellite and ground station platform has been presented. Previously implemented components of the IS-HS 2 have also been mentioned. By using FEC, the time-limited communications window of a satellite pass can be more efficiently utilised. Linear block codes will spread k information bits over a codeword of n information bits in order to lower the errors introduced by noise. A BCH code uses polynomial principles to achieve this, which is simple to implement in hardware. It provides good error correcting performance and a choice for the number of correctable errors in a codeword. By using the iterative BM decoding technique, a BCH decoder's hardware complexity can also be kept to a minimum. The LDPC code has been shown to outperform BCH [1] when using a random construction technique for **H**. A structured QC-LDPC code can provide the same performance as with random constructions, provided **H** has been properly designed. More importantly, a structured code will keep hardware complexity relatively low.

Stellenbosch University http://scholar.sun.ac.za

It has also been shown that the first step to implement FEC, is to determine a proper codeword length through error probability analysis. This calculation will require parameters from the testing scenario such as transmit power and receiver sensitivity. Finally, an overview of various IPC schemes has been given, which allow for interaction between adjacent protocol layers from the OSI model. It is clear that synchronisation is important when two processes share information. Shared memory is the fastest IPC medium, but doesn't guarantee concurrent data access. However, by using shared memory in conjunction with semaphores or signals for example, it can provide both fast and synchronised communication between two processes.

Determination of a proper block length for FEC, will be covered in Chapter 3. Details regarding encoding and decoding of both BCH and LDPC, are also handled. A proper IPC scheme is also chosen from the findings of this chapter.

Stellenbosch University http://scholar.sun.ac.za

Chapter 3

Detail Design

This chapter presents a more detailed description of the contents from Chapter 2. Referring to Fig. 2.1, a bottom up approach will be followed in this chapter. At first, an error probability analysis is performed for the wireless channel at D. This will indicate which FEC codeword length to use for a given BER constraint. Codeword length and block length will be used interchangeably in this thesis. The other modules including B, A, F and G will be designed around this chosen length. Firstly, all channel coding modules are designed after which the software protocol layers are handled. Finally, a block error rate (BLER) simulation application for BCH and LDPC is designed. Simulation strategies are discussed for both Matlab and FPGA platforms.

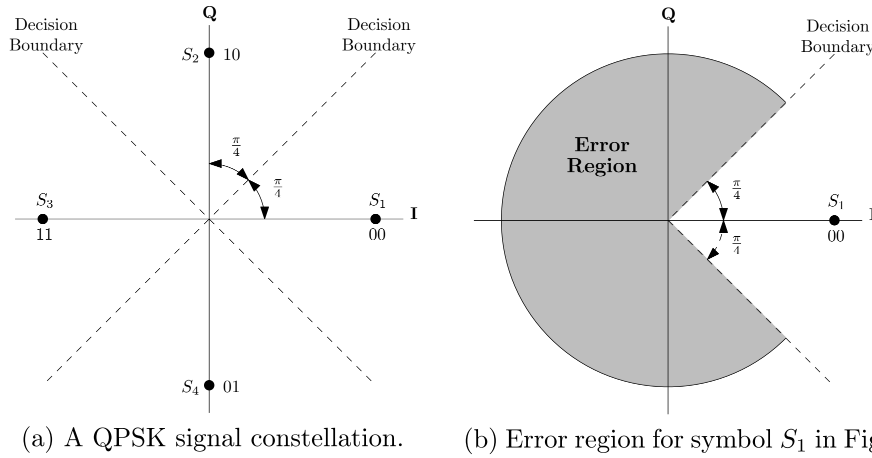
3.1 Block Error Probability Analysis

This section calculates a QPSK receiver's block error probability (BLEP) when using the BI-AWGN wireless channel model. A derivation for bit error probabilities and block error probabilities for such a receiver are covered in Appendix A.1. This section highlights some of these results and uses them to determine a suitable FEC block length for this implementation.

Fig. 3.1a shows a QPSK signal constellation with four two-bit symbols S_1 to S_4. Symbols are indicated as black dots on both the quadrature ($\mathbf{Q}$) and in-phase $\mathbf{I}$ axis. Each symbol differs by $\pi/2$ radians in phase from its neighbour. Phase noise added to a received symbol will cause it to deviate from its position in Fig. 3.1a. Should a symbol cross the dashed lines due to phase noise, the modem will make an error. For example, if symbol S_1's phase deviates into the grey area of Fig. 3.1b, it will be mistaken for another symbol. Since QPSK is a phase modulation scheme, only phase noise will be considered.

Suppose symbol S_1 having amplitude A has been received with some noise. It is now positioned at S_{noise} in Fig. 3.2. Gaussian noise is represented by $n(t)$ which has both in-phase, n_i, and quadrature, n_q, components. Adding vectors

Stellenbosch University http://scholar.sun.ac.za



(a) A QPSK signal constellation. (b) Error region for symbol S_1 in Fig. 3.1a.

Figure 3.1: QPSK symbol decision - and error regions.

S_1 and $n(t)$ results in S_{noise} of amplitude E and phase θ. Angle θ represents phase noise, which is of interest.

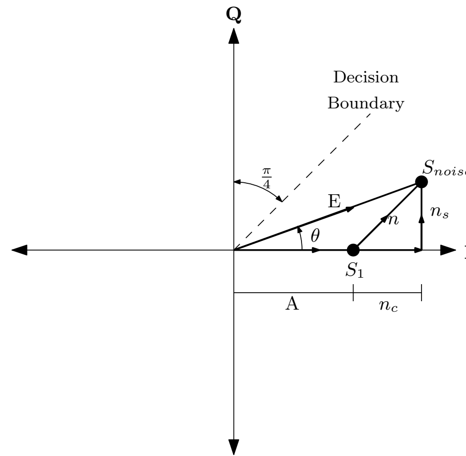


Figure 3.2: Gaussian white noise added to S_1.

Phase noise, θ, is a Gaussian random variable and its probability density function (PDF) is given by :

$$\rho_\theta(\theta) = \frac{1}{2\pi} e^{\frac{-A^2}{2\sigma^2}} \left[1 + \left(\frac{Acos(\theta)}{\sigma} \right) \left(e^{\frac{A^2 cos^2(\theta)}{2\sigma^2}} \right) \left(\sqrt{2\pi} \right) \times \left(1 - Q\left(\frac{Acos(\theta)}{\sigma} \right) \right) \right] \tag{3.1.1}$$

where

Stellenbosch University http://scholar.sun.ac.za

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_{x}^{\infty} \left(e^{\frac{-a^2}{2}} \right) da \tag{3.1.2}$$

Integrating $\rho_\theta(\theta)$ over the white region in Fig. 3.1b leads to a probability of successfully receiving a symbol :

$$P_{success} = \int_{\frac{-\pi}{4}}^{\frac{\pi}{4}} \rho_\theta(\theta) d\theta \tag{3.1.3}$$

This $P_{success}$ is used to determine the probability of receiving a symbol incorrectly :

$$P_{error} = 1 - \int_{\frac{-\pi}{4}}^{\frac{\pi}{4}} \rho_\theta(\theta) d\theta \tag{3.1.4}$$

Assuming both symbol bits have equal probability of being changed by noise, a bit error probability (BEP) is produced :

$$P_{bep} = \frac{P_{error}}{2} \tag{3.1.5}$$

Finally, having FEC correct up to t bit errors in a n-bit block leads to block error probability :

$$P_{blep} = \sum_{i=t+1}^{n} \binom{n}{i} P_{bep}^{i} (1 - P_{bep})^{n-i} \tag{3.1.6}$$

Eq. 3.1.6 is evaluated over a range of E_b/N_o SNR values. A SNR is expressed as either A^2/σ^2 or E_b/N_o. The former is a ratio of average signal power to average noise power whereas the latter is a ratio of energy per transmitted bit to noise power spectral density. Eq. 3.1.7 shows the relationship between these two SNRs for a QPSK receiver. Using this relationship, Eq. 3.1.1 can written i.t.o E_b/N_o. Relationship 3.1.7 is derived in Appendix A.2.

$$\frac{A^2}{\sigma^2} = \frac{2E_b}{N_o} \tag{3.1.7}$$

Using a link budget, the link margin at the satellite's receiver is computed. This value will determine the range of SNRs to use when evaluating Eq. 3.1.6. Since the aircraft simulator of Section 2.2.1 will be used for testing, all link budget calculations are done with regard to this scenario.

Stellenbosch University http://scholar.sun.ac.za

Work done in [7] suggests that an aircraft's altitude be 3 km for this simulation. An aircraft pass is illustrated in Fig. 3.3. The ground station uses a stationary patch antenna that faces towards the sky. Its beam width allows for communication within a $\alpha = 120°$ field of view. Therefore, communication will begin at elevation angle $\phi = 30°$. At this particular ϕ, the distance between the ground station and aircraft is $d = 6$ km. Signal loss at this d is the highest during a pass, and therefore places a lower bound on the receiver's link margin. A summary of the parameters used by link budget Eq. 2.6.1 are given in Table 3.1.

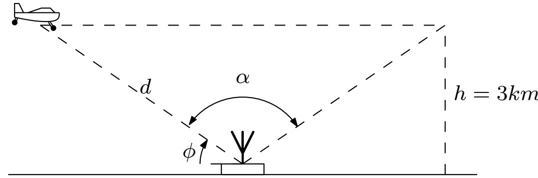


Figure 3.3: An aircraft passing over a ground station at altitude $h = 3$ km.

Parameter	Description	Value
G_{PA_TX}	Ground station transmit power.	30 dBm
L_{CABLE_TX}	Ground station cable losses.	0.5 dB
G_{ANT_TX}	Ground station antenna gain.	2.25 dB
L_{PATH}	Free space signal losses.	115 dB
G_{ANT_RX}	Payload antenna gain.	6 dB
L_{CABLE_RX}	Payload cable losses.	0.5 dB

Table 3.1: Link budget parameters for the uplink when using aircraft altitude $h = 3$ km and elevation angle $\theta = 30°$.

Transmit power and free space losses for this simulation have been determined in [35] and [7]. Cables connecting the amplifiers to antennas typically have low losses as indicated [5]. The SAA's gain is about $G_{ANT_RX} = 6$ dB when steering towards the ground station [36]. Sensitivity of the receiver at the payload has been designed to be $P_{SENSITIVITY}$ = -93 dBm [37]. By using Eq. 2.6.2 along with the aforementioned values leads to :

$$\begin{aligned} P_{LINK} &= (30 - 0.5 + 2.25 - 115 + 6 - 0.5) - (-93) \\ &= 15.25 \quad dB \end{aligned} \tag{3.1.8}$$

Knowing from Section 2.6.1 that P_{LINK} serves as a receiver's SNR, Value 3.1.8 forms the upper bound on the range of simulated SNRs for Eq. 3.1.6. A BCH's block length in bits is given as follows [3] :

Stellenbosch University http://scholar.sun.ac.za

$$n = 2^i - 1 \quad , \quad i \geq 3 \tag{3.1.9}$$

These blocks contain frames that are used by the software protocol layers. Frames are composed of both protocol overhead and file data to be sent over the link. Overhead for this implementation is to be calculated as $N_{ovh} = N_{ovh_TM} + N_{ovh_ARQ} = 256$ bits. Details regarding this calculation are discussed in Section 3.6. Since no file data can be fitted into a block having $n = N_{ovh}$, the minimum length, therefore, has to be $n > N_{ovh}$. Using this constraint along with Eq. 3.1.9, a minimum length of $n_{min} = 511$ is obtained.

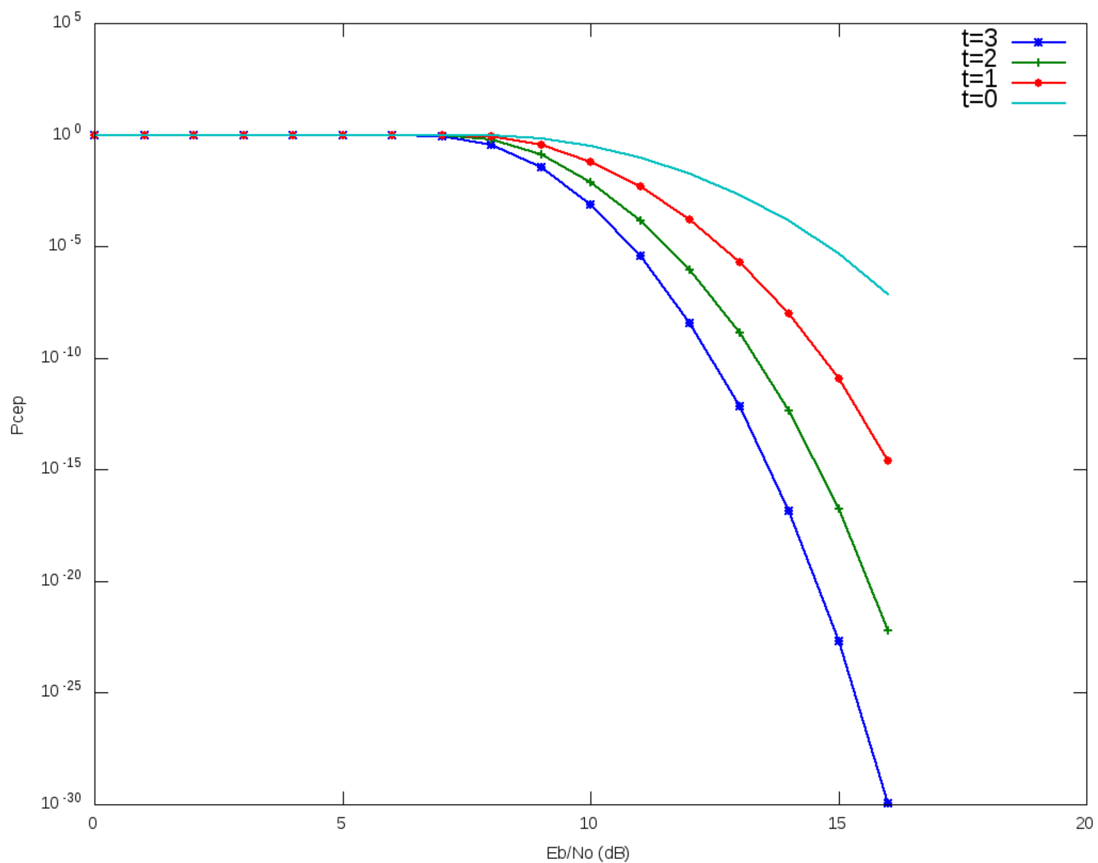


Figure 3.4: Codeword error probability vs. SNR for BCH when using block length $n = 511$ bits.

The results of evaluating Eq. 3.1.6 at different SNRs up to P_{LINK} is shown in Fig. 3.4. Different error correction capabilities ranging from $t = 0$ to $t = 3$ has been plotted for length $n = 511$. In general, a good error rate is $P_{blep} = 10^{-6}$. Clearly a BCH correcting up to $t = 3$ bit errors satisfies this property

Stellenbosch University http://scholar.sun.ac.za

between 11 dB and 15 dB SNR. Using the lookup table of standard BCH codes in [3], a $(n, k) = (511, 484)$ BCH code having $t = 3$ error correction capability is chosen.

3.2 Channel Coding Design

Work done in [1] suggests that these modules implement standards from the Consultative Committee for Space Data Systems (CCSDS). Specifically the channel coding standard of the Telemetry (TM) Space Data Link Protocol is implemented. Both ground station and satellite platforms, will contain the following modules :

- A 16-bit cyclic redundancy check (CRC).
- Forward error correction (FEC).
- A pseudo randomiser.
- An attached synchronisation marker (ASM).

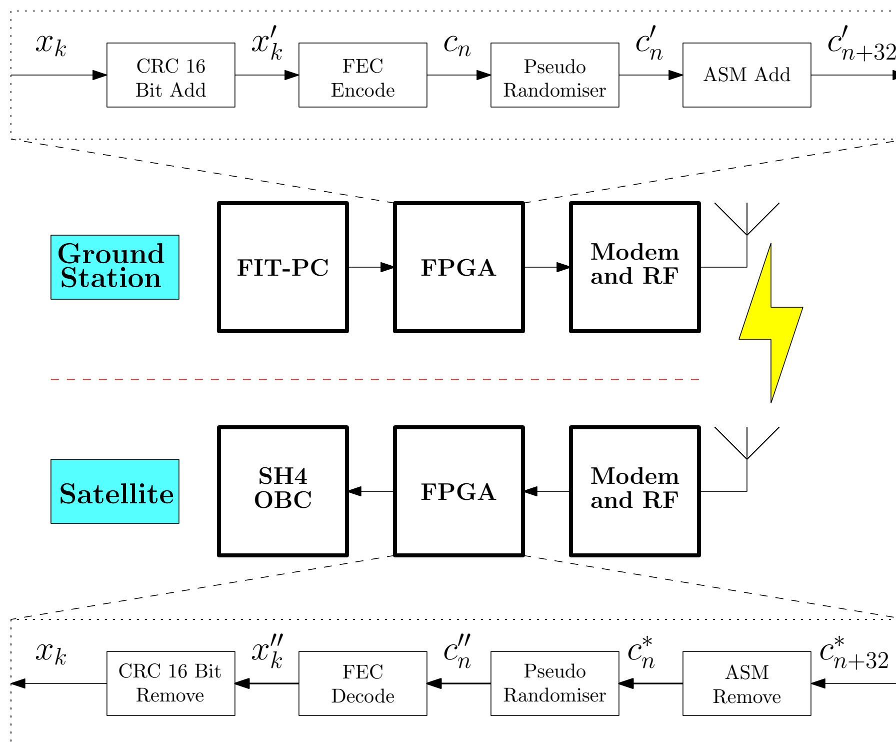


Figure 3.5: Interaction between channel coding modules on the FPGA for both ground station and satellite platforms.

Stellenbosch University http://scholar.sun.ac.za

Fig. 3.5 identifies data vectors entering or exiting a module by using different symbols. The subscripts of these symbols indicate how many bits they contain after being processed by a specific module. For example vector x_k contains k bits. These symbols are used in the following discussion of each module.

3.2.1 Cyclic Redundancy Check

A CRC code is used for error detection. It is excellent for detecting both random and burst errors present in a set of binary data [3]. In case FEC fails to correct all errors in a codeword, CRC will detect these errors and reject the faulty data. Similar to FEC, CRC adds information to a message at the transmitter and removes this information at the receiver.

CRC Encoding Step	Binary View	Polynomial View
i) Message and Generator Polynomials	$g = [g_1, g_0]$ $= [1, 1]$ $m = [m_2, m_1, m_0]$ $= [1, 1, 1]$	$g(X) = g_1X + g_0$ $= X + 1$ $m(X) = m_2X^2 + m_1X + m_0$ $= X^2 + X + 1$
ii) Zero Pad Message	$m' = [1, 1, 1, 0]$	$m'(X) = Xm(X)$ $= X^3 + X^2 + X + 0$
iii) Find Remainder After Division	1 0 1 1 1) 1 1 1 0 1 1 0 1 1 1 1	$X^2 + 1$ $X + 1$) $X^3 + X^2 + X + 0$ $X^3 + X^2$ 0 + X X + 1 1
iv) Add Remainder to Zero Padded Message	$m'' = m' + r$ $= [1, 1, 1, 1]$	$m''(X) = m'(X) + r(X)$ $= X^3 + X^2 + X + 1$

Figure 3.6: CRC encoding procedure.

Like BCH's encoding procedure, CRC views the message to be encoded as a polynomial $m(X)$. Firstly, a string of h zero bits are appended to $m(X)$ which is equivalent to $X^h m(X)$. The value of h is equal to the order of the CRC's generator polynomial $g(X)$. Using long division, polynomial $X^h m(X)$

Stellenbosch University http://scholar.sun.ac.za

is divided by $g(X)$ until remainder $r(X)$ is obtained. This remainder gets modulo-2 subtracted from $X^h m(X)$ such that $m'(X) = X^h m(X) - r(X)$ is divisible by $g(X)$ with zero remainder. Fig. 3.6 shows this encoding procedure from both binary and polynomial perspectives.

The ground station PC sends a k-bit frame, $\mathbf{x}_k$, to the channel coding Section in Fig. 3.5. This frame implements the TM protocol standard from ECSS and is discussed in Section 3.6.1. Due to this standard, $\mathbf{x}_k$ already contains the trailing zeros required for CRC. Therefore the CRC encoded message $\mathbf{x}'_k$ is also k bits long. After FEC decoding at the receiver, $\mathbf{x}''_k$ gets divided by $g(X)$. Message $\mathbf{x}''_k$ is error free when remainder $r(X)$ is zero, otherwise $\mathbf{x}''_k$ contains errors.

The CRC generator polynomial used in this design is given in Eq. 3.2.1. This polynomial has been specifically adopted by the CCSDS for its low probability of missing undetected errors [19]. A 16-bit CRC field available in message $\mathbf{x}_k$ will be set by this polynomial.

$$g(X) = X^{16} + X^{12} + X^5 + 1 \tag{3.2.1}$$

3.2.2 Forward Error Correction

Both BCH and LDPC will be implemented. The ground station adds $(n - k)$ redundant bits to $\mathbf{x}'_k$ to create n-bit codeword $\mathbf{c}_n$. The satellite's FEC decoder will attempt to correct all errors after which it will strip redundancy to produce error free message $\mathbf{x}''_k$. Details regarding BCH and LDPC are provided later in this chapter.

3.2.3 Pseudo Randomiser

Randomising the codeword allows it to have sufficient bit transitions. This ensures optimal utilisation of bandwidth for the given baudrate of the system [19]. It prevents all transmitted power being focused in a narrow bandwidth due to long sequences of either ones or zeros in $\mathbf{c}_n$. Secondly, the SDR modem uses a timing error detector to update how many symbol samples to wait for, before the next symbol starts. This timing error detector requires frequent symbol transitions to work properly.

A pseudo-random sequence is generated using a linear feedback shift register (LFSR). Its random output is generated by using polynomial $g(X)$, as determined by the CCSDS [19] :

$$g(X) = X^8 + X^7 + X^5 + X^3 + 1 \tag{3.2.2}$$

Fig. 3.7 shows the 8-bit LFSR's output being XORed with the FEC encoder's output to produce a randomised sequence of data. Since data only gets randomised, the length of $\mathbf{c}'_n$ is the same as $\mathbf{c}_n$ in Fig. 3.5.

Stellenbosch University http://scholar.sun.ac.za

The LFSR's output at x_1 is also fed back to its input at x_8. This output gets XORed with other LFSR bits along the way. Polynomial $g(X)$ determines which LFSR bits to XOR output x_1 with. This polynomial also determines how long the LFSR's random sequence is before repeating itself. This design's sequence repeats after 255 bits.

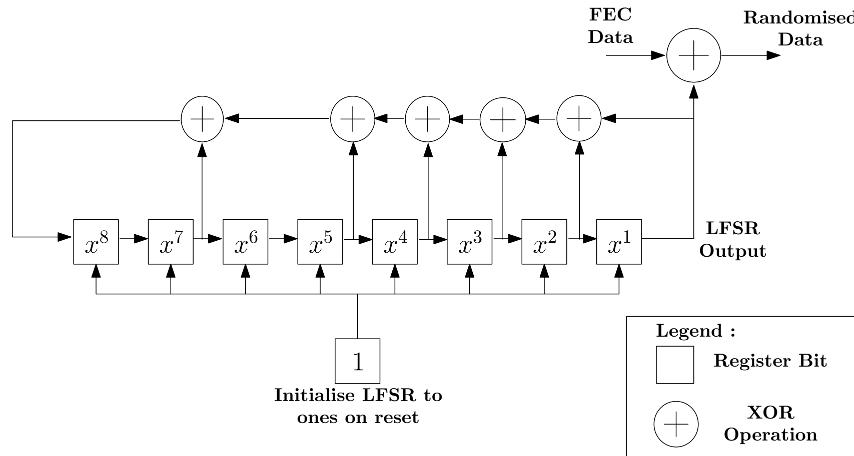


Figure 3.7: Pseudo random sequence LFSR.

At the receiver in Fig. 3.5, codeword $\mathbf{c}_n^*$ is XORed with the same pseudo-random sequence used at the transmitter. This removes the randomisation effect that has been applied to $\mathbf{c}_n$.

3.2.4 Attached Synchronisation Marker

The attached synchronisation marker (ASM) synchronises the randomiser, FEC and CRC at the receiver with the start of a new codeword. This 32-bit sequence precedes the randomised codeword $\mathbf{c}'_n$ to produce $\mathbf{c}'_{n+32}$ in Fig. 3.5. At the receiver, the ASM is removed before $\mathbf{c}_n^*$ enters the de-randomiser. The sequence used in this design is shown below in hexadecimal and binary format, where the right most bit represents the least significant bit (LSB) :

$$\underbrace{1}_{0001}\ \underbrace{A}_{1010}\ \underbrace{C}_{1100}\ \underbrace{F}_{1111}\ \underbrace{F}_{1111}\ \underbrace{C}_{1100}\ \underbrace{1}_{0001}\ \underbrace{D}_{1101}$$

This specific sequence contains sufficient bit transitions and, according to the CCSDS, can also be used by the receiver's SDR for symbol synchronisation purposes [38]. However, in this project the ASM will only be used to identify the start of a codeword.

Stellenbosch University http://scholar.sun.ac.za

3.3 BCH FEC Design

This section discusses the encoding and decoding techniques for the $(511, 484)$ BCH code of Section 3.1. Note that BCH makes extensive use of Galois field (GF) arithmetic. A Galois field is the group of a finite collection of unique numbers [3]. Addition or multiplication operations on any two or more numbers of this group, results in another number that exists in this group.

A Galois field of prime q having a group size of q^m elements is denoted GF(q^m). The prime q indicates which alphabet is being used. In the case of a binary alphabet, $q = 2$ and there exists 2^m m-bit binary numbers. The $(511, 484)$ BCH code uses GF(2^9).

Finally, there exists a m-bit number, α, in GF(2^m) such that subsequent powers of α will generate $2^m - 1$ unique numbers in the group. The group's 2^m unique numbers include $0, 1, \alpha, \alpha^2 \cdots \alpha^{2^m-2}$.

3.3.1 Encoder

A generator polynomial $g(X)$ is used to create a codeword as mentioned in Section 2.5.2. The polynomial for a $(511, 484)$ code obtained from lookup tables in either Matlab or [3]. A (511,484) code's $g(X)$ is shown below :

$$\begin{aligned} g(X) =& X^{27} + X^{26} + X^{24} + X^{22} + X^{21} + X^{16} + X^{13} + X^{11} \\ &+ X^9 + X^8 + X^6 + X^5 + X^4 + X^3 + 1 \end{aligned} \tag{3.3.1}$$

Encoding is performed similar to that of CRC in Fig. 3.6. The $(511, 484)$ code appends 27 zeros to 484-bit message $m(X)$ to form a 511-bit vector $m'(X)$. Using long division, $m'(X)$ is divided by $g(X)$ until a remainder $r(X)$ of order less than 27 is found. This $r(X)$ gets modulo-2 added to $m'(X)$ to form codeword $c(X)$, which is divisible by $g(X)$.

Since the (511,484) code can correct $t = 3$ bits, the abovementioned polynomial has a special property. It has GF elements α^i for $i = 1, 2, \cdots, 2t$ as its roots such that $g(\alpha^i) = 0$ [3]. Furthermore, polynomial $g(X)$ is the least common multiple (LCM) of at most $2t$ minimal polynomials $\phi_i(X)$ for $i = 1, 2, \cdots, 2t$. A minimal polynomial cannot be further factored and has α^i as its root such that $\phi_i(\alpha^i) = 0$ [3]. Polynomial $g(X)$ can therefore be written as :

$$g(X) = LCM\{\phi_1(X), \phi_2(X), \cdots, \phi_{2t}(X)\} \tag{3.3.2}$$

These minimal polynomials are necessary for calculating the syndrome in the decoder as discussed below.

Stellenbosch University http://scholar.sun.ac.za

3.3.2 Decoder

Section 2.5.1 mentioned the use of parity matrix **H** to compute syndrome **s**. In BCH codes, matrix **H** is defined as [3]:

$$\mathbf{H} = \begin{bmatrix} 1 & \alpha & \alpha^2 & \cdots & \alpha^{n-1} \\ 1 & (\alpha^2) & (\alpha^2)^2 & \cdots & (\alpha^2)^{n-1} \\ \vdots & & & & \vdots \\ 1 & (\alpha^{2t}) & (\alpha^{2t})^2 & \cdots & (\alpha^{2t})^{n-1} \end{bmatrix} \tag{3.3.3}$$

Note that **H** only has $2t$ rows, which is not equal to the number of parity bits present in a codeword. This is due to the m-bit GF elements present in **H**, which forces it to have less than $(n-k)$ rows.

The decoder receives codeword $\mathbf{c}^*$ which may contain errors. Eq. 3.3.4 writes this i.t.o the correct codeword **c** and an error vector **e**. A BCH decoder must find this **e** in order to correct errors in $\mathbf{c}^*$.

$$\mathbf{c}^* = \mathbf{c} + \mathbf{e} \tag{3.3.4}$$

Using Eq. 2.5.9 and assuming **e**=0, leads to :

$$\begin{aligned} \mathbf{s} &= \mathbf{c}^* \times \mathbf{H}^T \\ &= 0 \end{aligned} \tag{3.3.5}$$

Putting **s** in polynomial format, the above equation implies :

$$s_i = c^*(\alpha^i) \quad , \quad i = 1, 2, \cdots, 2t \tag{3.3.6}$$

As mentioned earlier, $g(\alpha^i) = 0$. Combining Eqs. 3.3.4, 3.3.6 and 2.5.12 gives :

$$\begin{aligned} s_i &= c^*(\alpha^i) \\ &= c(\alpha^i) + e(\alpha^i) \\ &= d(\alpha^i)g(\alpha^i) + e(\alpha^i) \\ &= e(\alpha^i) \end{aligned} \tag{3.3.7}$$

Eq. 3.3.7 does not imply that $e(X)$ is the remainder after dividing $c^*(X)$ by $g(X)$, since the order of $e(X)$ could be greater than $g(X)$. By using property 3.3.2 and Eqs. 3.3.4 and 2.5.12, the following is obtained :

$$c^*(X) = c(X) + e(X)$$

Stellenbosch University http://scholar.sun.ac.za

$$
\begin{aligned}
&= d(X)g(X) + e(X) \\
&= a_i(X)\phi_i(X) + b_i(X) \quad , \quad i = 1, 2, \cdots, 2t
\end{aligned}
\tag{3.3.8}
$$

where $b_i(X)$ is the remainder after dividing $c^*(X)$ by $\phi_i(X)$. Evaluating Eq. 3.3.8 at α^i also gives $s_i = c^*(\alpha^i) = b(\alpha^i)$, since $\phi(\alpha^i) = 0$. Now that the syndromes can be computed, error locations in $\mathbf{c}^*$ need to be identified. Since BCH corrects up to t random errors, vector $e(X)$ is written as [3]:

$$
e(X) = X^{j_1} + X^{j_2} + \cdots + X^{j_v} \tag{3.3.9}
$$

where j_v is a random bit position in codeword $\mathbf{c}^*$ for $v \leq t$. Using Eqs. 3.3.7 and 3.3.9 gives :

$$
\begin{aligned}
s_i &= (\alpha^i)^{j_1} + (\alpha^i)^{j_2} + \cdots + (\alpha^i)^{j_v} \\
&= (\alpha^{j_1})^i + (\alpha^{j_2})^i + \cdots + (\alpha^{j_v})^i \\
&= \beta_1^i + \beta_2^i + \cdots + \beta_v^i \quad ; \quad i = 1, 2, \cdots, 2t \quad ; \quad v \leq t
\end{aligned}
\tag{3.3.10}
$$

Lin and Costello [3] defines an error-location polynomial $\sigma(X)$ shown in Eq. 3.3.11. The inverse of its roots represent the error locations in codeword $\mathbf{c}^*$.

$$
\begin{aligned}
\sigma(X) &= (1 + \beta_1 X)(1 + \beta_2 X) \cdots (1 + \beta_v X) \\
&= \sigma_0 + \sigma_1 X + \cdots + \sigma_v X^v \quad , \quad v \leq t
\end{aligned}
\tag{3.3.11}
$$

It is also shown by [3] that the syndromes in 3.3.6 and the coefficients of 3.3.11 are related as follow :

$$
\begin{aligned}
s_1 + \sigma_1 &= 0 \\
s_2 + \sigma_1 s_1 + 2\sigma_2 &= 0 \\
s_3 + \sigma_1 s_2 + \sigma_2 s_1 + 3\sigma_3 &= 0 \\
&\cdots \\
s_v + \sigma_1 s_{v-1} + \cdots + \sigma_{v-1} s_1 + v\sigma_v &= 0 \\
s_{v+1} + \sigma_1 s_v + \cdots + \sigma_{v-1} s_2 + \sigma_v s_1 &= 0
\end{aligned}
\tag{3.3.12}
$$

Three techniques exist to solve the above equations. Peterson's algorithm puts these equations in matrix form and tries to solve all $\sigma(X)$ coefficients [15]. Although this is a simple approach, it involves computing determinants of matrices which increases in complexity for long codewords [15]. The Berlekamp-Massey algorithm (BMA) tries to find a LFSR having feedback coefficients equal to the coefficients of $\sigma(X)$ as shown in Fig. 3.8. This LFSR eventually produces all syndromes present in the last Eq. of 3.3.12. Similar to BMA,

Stellenbosch University http://scholar.sun.ac.za

the Euclidean algorithm (EA) also tries to find the coefficients of a LFSR that produces all syndromes [15]. However, the BMA works with polynomials having a degree less than $\sigma(X)$ when compared to EA [15]. This results in the BMA having a simpler hardware implementation than EA. Since the BMA is a fast decoding algorithm with low hardware complexity ([2], [15]), it will be used in this design.

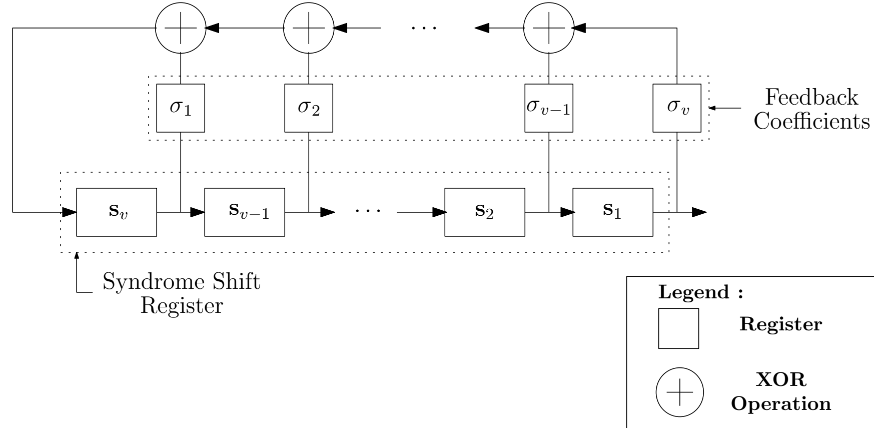


Figure 3.8: LFSR computed by the Berlekamp-Massey algorithm [2].

BMA iteratively finds the coefficients of $\sigma(X)$ by solving each Eq. in 3.3.12. Letting k depict the current iteration, $\sigma^{(k)}(X)$ is the error location polynomial at iteration k where $1 \leq k \leq 2t$:

$$\sigma^{(k)}(X) = 1 + \sigma_1^{(k)} X + \sigma_2^{(k)} X^2 + \cdots + \sigma_l^{(k)} X^l \quad , \quad l \leq v \tag{3.3.13}$$

The BMA is initialised with $\sigma^{(0)}(X) = \sigma_0 = 1$ [3]. At $k = 1$, the first Eq. of 3.3.12 is evaluated to obtain $\sigma^{(1)}(X)$. Since $\sigma_1 = 0$ at this stage, a correction term is added to $\sigma^{(0)}(X)$ such that $\sigma^{(1)}(X) = \sigma^{(0)}(X) + T_{corr}(X)$ satisfies the first Eq. of 3.3.12. Similarly, for $k = 2$ the second Eq. of 3.3.12 is evaluated using coefficients from $\sigma^{(1)}(X)$. Again, a correcting term would be added to $\sigma^{(1)}(X)$ should the second Eq. not be satisfied. However, using modulo-2 addition results in $2\sigma_2 = 0$ since σ_2 is multiplied with an even value. Therefore Eq. two of 3.3.12 is satisfied when using $\sigma^{(1)}$, hence $\sigma^{(2)}(X) = \sigma^{(1)}(X)$. In general $\sigma^{(k+1)}(X) = \sigma^{(k)}(X)$ when the $(k+1)$-th Eq. of 3.3.12 is satisfied by using coefficients from $\sigma^{(k)}(X)$. Any other case will result in a correction term being added such that $\sigma^{(k+1)}(X) = \sigma^{(k)}(X) + T_{corr}(X)$ satisfies the first $(k+1)$ Eqs. of 3.3.12 [3]. Note that correction term, $T_{corr}(X)$, is calculated such that $\sigma^{(k+1)}(X)$ is of minimal degree [3]. The BMA will run for at most $2t$ iterations before $\sigma(X)$ is found. Further details regarding the BMA are discussed in [3] and [15].

Stellenbosch University http://scholar.sun.ac.za

The final step before finding polynomial $e(X)$ in Eq. 3.3.9 is to locate the roots of $\sigma(X)$. This is done by substituting and testing whether $\sigma(\alpha^i) = 0$ for $i = 0, 1, 2, \cdots, 2^m - 2$. After finding all α^i for which $\sigma(\alpha^i) = 0$, the inverse of these roots are calculated as required by Eq. 3.3.11. Error locations in a codeword are then revealed by the exponents of these inverted roots. These exponents coincide with exponents j_v from error vector $e(X)$ in Eq. 3.3.9. Using Eq. 3.3.4, this $e(X)$ is XORed with $c^*(X)$ to obtain error free codeword $c(X)$. A GF element's inverse is defined as [3] :

$$(\alpha^i)^{-1} = \alpha^{2^m-1-i} \quad , \quad i = 0, 1, 2, \cdots, 2^m - 1 \tag{3.3.14}$$

Since evaluation of $\sigma(\alpha^i)$ starts at $i = 0$, the exponent of $(\alpha^i)^{-1}$ is $j = 2^m - 1 - i$. If codeword $c^*(X)$ is stored in the decoder as shown in Eq. 2.5.10, error correction on $c^*(X)$ would take place from right to left. A specialised circuit to perform error locating and correction has been developed by Chien [3] as shown in Fig. 3.9. Before starting, the registers represented by box A gets initialised to 1. Also the complete received codeword is present in a FIFO register before starting. Firstly, the contents of registers A gets multiplied by $\sigma(X)$ coefficients in box B. These outputs go to a summation circuit at C which evaluates $\sigma(\alpha^i)$ as mentioned earlier. Note that $\sigma_0 = 1$ is not included in this summation. This causes the summation output to be '1' when $\sigma(\alpha^i) = 0$. Bit $c^*_{2^m-1-i}$ is then read from the FIFO after which it gets XORed with either 1 or 0, depending whether $\sigma(\alpha^i) = 0$ or not.
Algorithm 1 displays a pseudo code summary of the BCH decoder. After receiving a complete codeword $\mathbf{c}^*$, the syndromes are calculated. This is followed by the BMA which calculates error-locator polynomial $\sigma(X)$. Using this $\sigma(X)$, the Chien circuit determines the error locations in $\mathbf{c}^*$ and corrects them.

Stellenbosch University http://scholar.sun.ac.za


```
Input  : Codeword c* (n-bit vector)
Output : Message x (k-bit vector)

** Syndrome Calculation **

for i ← 1 to 2t do
    Divide c*(X) by φ_i(X);
    Store remainder as r_i(X);
    s_i = r_i(α^i);
end

** Calculate σ(X) with BMA **

Initialise : σ^(0)(X) = σ_0
for i ← 1 to 2t do
    sum = [Σ_{j=0}^{i-1} (s_{j-i})(σ_j)] + (i)(σ_i)
    if sum not equals 0 then
        Calculate Correction Term T_cor;
        σ^(i) = σ^(i-1) + T_cor;
    else
        σ^(i) = σ^(i-1);
    end
end

** Chien Search **

for i ← 0 to n - 1 do
    sum = [Σ_{j=1}^{t} ((α^j)^i)(σ_j)];
    if sum equals 1 then
        Change bit (n - i) of codeword c*;
    end
end

x = Strip (n - k) redundant bits from c*;
```

Algorithm 1: BCH decoder algorithm.

Stellenbosch University http://scholar.sun.ac.za

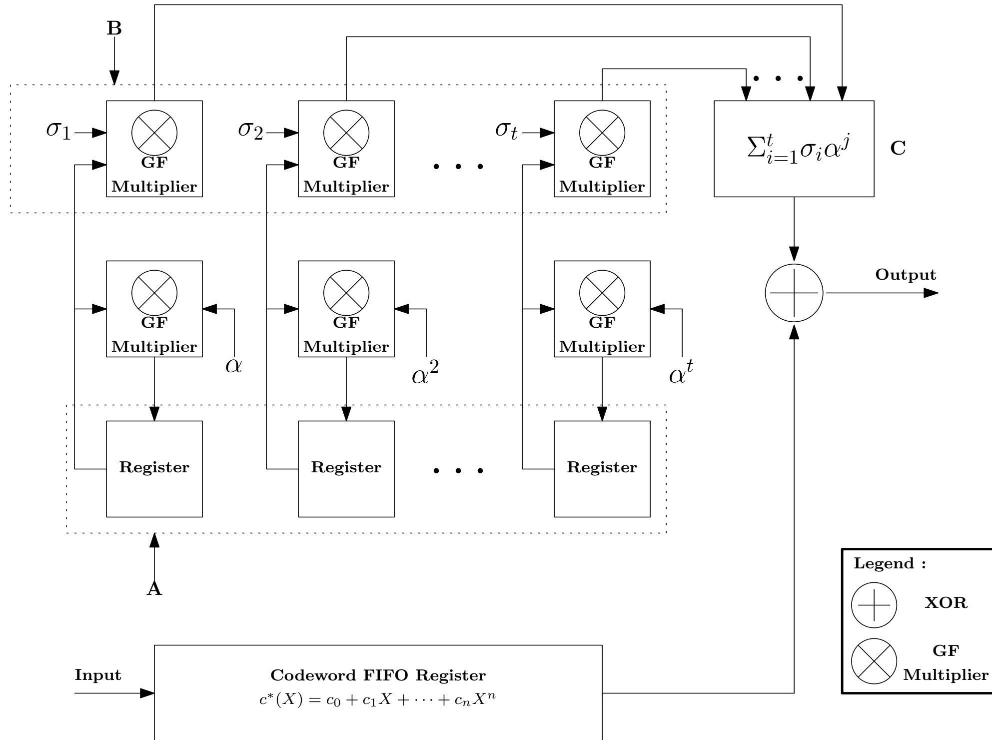


Figure 3.9: A Chien search circuit [3].

3.4 LDPC FEC Design

The (511,484) BCH is the main FEC used during the aircraft based test. However, it has been shown in [1] that LDPC is also worth considering over conventional linear block codes such as BCH. Parallel with BCH's development, LDPC FEC is implemented as an alternative to the (511,484) BCH.

Unlike BCH, LDPC doesn't have a designed error correction capability. Error correction performance depends on the construction technique of parity check matrix **H** as mentioned in Section 2.5.3. Random irregular constructions of **H** delivers the best performance [39] when using long block lengths of $n >$ 10000 [15]. However, structured codes can outperform random and irregular codes for short to medium block lengths where $n < 10000$ [39].

Work done in [1] indicates that half rate code LDPC delivers a good BLER performance at low SNRs of 4 dB. Using row weight ω_r and column weights ω_c, a regular LDPC code's rate is given by :

$$R = 1 - \frac{\omega_c}{\omega_r} \tag{3.4.1}$$

Among the half rate LDPC codes, (3,6)-regular codes have a low hardware complexity decoder [40]. Column weights of $\omega_c < 3$ tend to deliver poor BLER results [15] whereas $\omega_c > 3$ lowers the error floor for high SNRs ([41], [42]) at the expense of elevated decoder complexity [40]. Error floors refer to a

Stellenbosch University http://scholar.sun.ac.za

phenomenon that the slope of a code's BLER curve decreases when increasing the SNR [43]. Irregular LDPC codes also have an encoding complexity that is quadratic with block length n [44]. In general, short regular LDPC codes can be encoded in almost linear time when using the technique suggested by [30]. By using a regular QC-LDPC structured code, this encoding complexity is further reduced to linear time [4]. Due to QC-LDPC's structure, an encoder can use simple shift registers to perform encoding [4] . This avoids having to store large generator matrices in memory. This implementation will use a (3,6)-regular QC-LDPC code having block length $n = 512$.

A QC-LDPC code's $\mathbf{H}$ matrix is constructed in Fig. 3.10a. Sub-matrices $H_{i,j}$ are square $v \times v$ circulant matrices where $1 \leq i \leq a$, $1 \leq j \leq b$, $a = k/v$ and $b = n/v$. A circulant matrix has each row equal to the previous row being rotated one bit position to the right. Similarly, each column is equal to the previous being rotated one bit position downwards. Circulant matrices having a row weight of one are referred to as circulant permutation matrices as shown in Fig. 3.10b.

$$\mathbf{H} = \begin{bmatrix} H_{1,1} & \cdots & H_{1,b} \\ \vdots & \ddots & \vdots \\ H_{a,1} & \cdots & H_{a,b} \end{bmatrix}$$

(a) Circulant composition of a QC-LDPC parity check marix $\mathbf{H}$.

$$H_{i,j} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

(b) An example of a 5x5 circulant permutation matrix.

Figure 3.10: A QC-LDPC parity matrix structure.

Two classes of decoders exist, namely hard decision and soft decision. Similar to the Chien error correction circuit of BCH in Section 3.3.2, hard decision decoding directly changes a bit's value in the received codeword [15]. It provides the simplest decoding architecture [15] at the expense of BER performance as shown in [1] and [15]. Performance penalties up to 1.5 dB at the same BER can be expected when using hard decision instead of soft decision decoding [1]. Soft decision decoders gather bit-probability information from the modem before starting a decoding cycle. Bit-probability provides a confidence value in the correctness of each bit as received by the modem. These confidence values are then used to solve each parity check equation of $\mathbf{H}$ such that the most probable codeword $\mathbf{c}_{new}$ is found for which $\mathbf{c}_{new}\mathbf{H}^T = 0$ [1]. Since soft decision decoding delivers superior BER performance compared to hard decision decoding [15], it will be used in this design.

Stellenbosch University http://scholar.sun.ac.za

3.4.1 Soft-Decision Decoding

Section 2.5.1 mentioned each row of $\mathbf{H}$ representing a parity check equation. Also each column position of $\mathbf{H}$ coincides with a bit's position in the received codeword $\mathbf{c}^*$. Since this LDPC implementation has a column weight $\omega_c = 3$, each bit of $\mathbf{c}^*$ is involved in three different parity check equations. The set of parity check equations in which bit c_i^* participates is denoted M_i for $0 \leq i \leq n$. A set which excludes parity check equation p is written as $M_{i/p}$ where $p \in M_i$. Similarly, a row weight $\omega_r = 6$ indicates that six bits of $\mathbf{c}^*$ are used per parity check equation. Such a set of bits participating in parity check s_j is denoted N_j for $0 \leq j \leq k$. Set $N_{j/i}$ excludes parity bit c_i^*, where $c_i^* \in N_j$.

Soft decision decoding consists of two major steps; horizontal step updating and vertical step updating. During these steps, messages containing probabilities are passed between check nodes and variable nodes in the Tanner graph. Each step is listed below :

- **Horizontal Updating** : Each check node C_j receives a message from all its connected variable nodes V_i of set N_j. In return, a message is sent from C_j to each connected V_i, which contains the probability $Pr(s_j = 0|c_i^* = x)$ where x is either 1 or 0. This message indicates the probability of parity check equation j being satisfied given the value of bit c_i^*. Only messages from variables nodes of set $N_{j/i}$ are considered when sending V_i a message.

- **Vertical Updating** : During horizontal updating, each variable node V_i received messages from its connected check nodes C_j of set M_i. A message is then generated for each check node C_j, which contains the probability $Pr(c_i^* = x|s_j = 0)$ for $x = 1$ or $x = 0$. This message represents the probability of bit c_i^* being its its current value given that all parity check Eqs. s_j involving bit c_i^* are satisfied. Again, only messages from check nodes of set $M_{i/j}$ are considered when sending C_j a message.

A message sent from check node C_j to variable node V_i is denoted r_{ji}. Similarly, message q_{ij} is sent from V_i to C_j. These messages along with their directions are shown in Fig. 3.11. Note that $q_{ij} = q_{ij}(0) + q_{ij}(1) = 1$ where $q_{ij}(0)$ is the probability of bit $c_i^* = 0$ and $q_{ij}(1)$ the probability of $c_i^* = 1$ [15]. Both $q_{ij}(0)$ and $q_{ij}(1)$ are sent when sending q_{ij}.
When initialising the decoder, bit-probability information are supplied by the modem. These probabilities are denoted $p_i(0) = Pr(c_i^* = 0)$ and $p_i(1) = Pr(c_i^* = 1)$ and will serve as initial values for vertical update messages q_{ij} such that $q_{ij}(0) = p_i(0)$ and $q_{ij}(1) = p_i(1)$. By using this p_i along with all q_{ij}, a value q_i is calculated from which an initial codeword $\mathbf{c}^*$ is determined. Having $q_i = q_i(1) + q_i(0) = 1$, bit $c_i^* = 1$ when $q_i(1) > 0.5$, or $c_i^* = 0$ when $q_i(0) > 0.5$. This is followed by evaluation of syndrome $\mathbf{s} = \mathbf{c}^*\mathbf{H}$. Should $\mathbf{s} \neq 0$, a decoding cycle starts otherwise the codeword is error free when $\mathbf{s} = 0$.

Stellenbosch University http://scholar.sun.ac.za

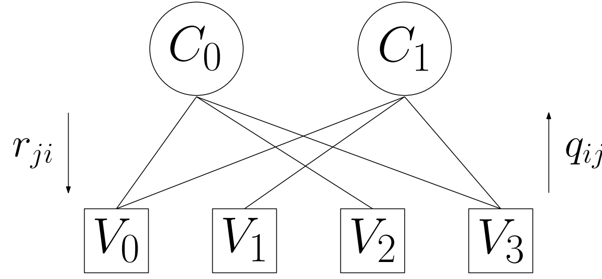


Figure 3.11: Message passing along the edges of a Tanner graph.

A decoding cycle may run many iterations before finding an error free codeword. The first step in an iteration performs horizontal updating for all check nodes. This is followed by vertical updating where new messages q_{ij} are generated. Calculating q_i as stated earlier, a new codeword $\mathbf{c}_{new}$ is determined before evaluating $\mathbf{s} = \mathbf{c}_{new}\mathbf{H}$. If $\mathbf{s} \neq 0$ a new iteration is started. Decoding is successful as soon as $\mathbf{s} = 0$. Failure is declared when $\mathbf{s} \neq 0$ and a predefined maximum number of iterations are exceeded. The decoding technique described in this Section is known as a belief propagation (BP) or sum-product (SP) decoder.

3.4.2 Min-Sum Decoder

The SP decoder's messages are all expressed in the probability domain. Calculating messages r_{ji} and q_{ij} requires many multiplication operations [15] which consumes lots of logic when implemented on a FPGA. Working in the log domain transforms many of these multiplication operations into simple addition operations [27]. Specifically bit-probability $Pr(c_i^* = x)$ is expressed as a log-likelihood ratio (LLR) :

$$LLR = log\left(\frac{Pr(c_i^* = 0)}{Pr(c_i^* = 1)}\right) = log\left(\frac{Pr(c_i^* = 0)}{1 - Pr(c_i^* = 0)}\right) \tag{3.4.2}$$

Message r_{ji} of the SP decoder is now replaced by a log domain message L_{ji} :

$$L_{ji} = -2tanh^{-1}\left(\prod_{j' \in M_{i/j}} tanh\left(\frac{Z_{ij'}}{2}\right)\right) \tag{3.4.3}$$

Similarly, message q_{ij} is replaced by :

$$Z_{ij} = (L_c)(c_i^*) + \sum_{i' \in N_{j/i}} L_{ji'} \tag{3.4.4}$$

Stellenbosch University http://scholar.sun.ac.za

Term L_c is known as the channel reliability and is given by :

$$L_c = 2\frac{\sqrt{(R)(E_b)}}{\sigma^2} \tag{3.4.5}$$

where R is the FEC code rate, E_b the energy per bit and σ^2 the noise power. Term q_i, from the previous Section, is also replaced by Y_i [15]:

$$Y_i = (L_c)(c_i^*) + \sum_{i \in N_j} L_{ji} \tag{3.4.6}$$

However, Eq. 3.4.3 still contains products and non-linear $tanh()$ functions that would consume lots of logic on a FPGA. Secondly, L_c of Eq. 3.4.5 requires noise power which is difficult to measure [27]. By using the minimum-sum (MS) algorithm, a LLR SP decoder's hardware complexity can be further reduced. Here, Eq. 3.4.3 can be estimated as :

$$L_{ji} \approx \left(\prod_{j' \in M_{i/j}} sign(Z_{ij'}) \right) . (min|Z_{ij'}|) \tag{3.4.7}$$

Value L_{ji} in Eq. 3.4.3 is primarily dominated by the smallest value of $Z_{ij'}$, hence the approximation in 3.4.7. Term $sign(Z_{ij'})$ refers to the $+1$ or -1 sign of $Z_{ij'}$. Also the $sign()$ function has a low complexity implementation in hardware [39]. In Eq. 3.4.5, term $(L_c)(c_i^*)$ represents the initial value of Z_{ij} obtained from the modem. However, in MS channel reliability is omitted and $(L_c)(c_i^*)$ replaced by the LLR of c_i^* [27], denoted $(c_i^*)_{LLR}$. An additional improvement to MS is the normalised MS algorithm. Since L_{ji} is approximated in the MS, its value tends to be slightly higher than expected [29]. Term L_{ji} is replaced with :

$$L_{ji} \rightarrow \alpha L_{ji} \tag{3.4.8}$$

where $\alpha < 1$ is a scaling value. The best α for a LDPC implementation is determined by simulation [1]. Typical values for α range between 0.7 and 0.9 [29]. By terminating scaling a few iterations before decoding finishes, provides about 0.25 dB improvement i.t.o BER performance [29]. It has been found that α shrinks the value of L_{ji} too much, when a maximum of 20 to 30 iterations per decoding cycle are used. This thesis will use a normalised MS decoder with early termination of α for a FPGA implementation of LDPC. Algorithm 2 explains a soft decision MS decoder.

Stellenbosch University http://scholar.sun.ac.za

Input : Codeword c^* (n-bit vector)
Output: Message x (k-bit vector)

*** Initialise ***

foreach *bit* c_i^* *in codeword* $\mathbf{c}^*$ **do**
    foreach *row* j *where* $H_{ij} = 1$ **do**
        $Z_{ij} \leftarrow (c_i^*)_{LLR}$;
    end
end
while *iteration* $<$ *MAX_ITERATIONS* ***and*** *done* $==$ *0* **do**
    *** Check node update ***
    foreach *row* i *in* H **do**
        find first and second minimum $|Z_{ij}|$ in row i;
        calculate $\prod$ *signs* of terms Z_{ij};
        foreach j *where* $H_{ij} = 1$ **do**
            $sign(L_{ji}) = (\prod signs) \times sign(Z_{ij})$;
            if *|first minimum|* $==$ $|Z_{ij}|$ **then**
                $|L_{ji}| = \alpha \times$ |second minimum|;
            else
                $|L_{ji}| = \alpha \times$ |first minimum|;
            end
        end
    end
    *** Variable node update ***
    foreach *column* i *in* $\boldsymbol{H}$ **do**
        sum=0;
        foreach j *where* $H_{ij} = 1$ **do**
            sum = sum + L_{ji};
        end
        foreach j *where* $H_{ij} = 1$ **do**
            $Z_{ij} = sum - L_{ji}$
        end
        $(c^*_{new_i})_{LLR} = (c_i^*)_{LLR}$ + sum;
    end
    *** Compute new codeword ***
    foreach *element* i *in* $(\mathbf{c}^*_{new})_{LLR}$ **do**
        if $(c_{new_i})_{LLR} > 0$ **then**
            $c_{new_i} = 0$;
        else
            $c_{new_i} = 1$;
        end
    end
    s = $\mathbf{c}_{new} \times H$;
    if s $==$ *0* **then** *done* = 1;**else** *done* = 0;
    iteration++;
end

Algorithm 2: Minimum-Sum LDPC decoder algorithm.

Stellenbosch University http://scholar.sun.ac.za

3.4.3 Parity Check Matrix Construction

Minimum Hamming distance and girth are the main parameters that influence a QC-LDPC code's BLER performance. A QC-LDPC code's minimum Hamming distance is bounded by the column weight of **H** [45]. Unlike a random construction of **H**, this distance does not increase linearly as block length n increases for QC-LDPC [45]. A good Hamming distance primarily lowers the error floor phenomenon mentioned before. High girth allows the decoder to converge quickly towards a correct codeword. The (3,6)-regular LDPC in [1] achieves a BER of 10^{-6} at low SNRs of 4 dB without encountering an error. Therefore only girth optimisations will be performed when constructing **H**. By using the encoding technique from [30], matrix **H** is designed to be approximately lower triangular :

$$\mathbf{H} = \begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix} \tag{3.4.9}$$

with T a lower triangular matrix having only zeros above its diagonal. Having this **H**, codeword **c** is given as :

$$\mathbf{c} = [x, p_1, p_2] \tag{3.4.10}$$

where x is a k-bit message. Parity vectors p_1 and p_2 have a combined length of $n-k$ bits and is given by :

$$p_1^T = -\phi^{-1}[-(E)(T^{-1})(A) + C]x^T \tag{3.4.11}$$

$$p_2^T = -T^{-1}[(A)(x^T) + (B)(p_1^T)] \tag{3.4.12}$$

with

$$\phi = -(E)(T^{-1})(B) + D \tag{3.4.13}$$

Dimensions of the sub-matrices in Eq. 3.4.9 appear in Fig. 3.12.
Dimension g is conveniently chosen such that $g = n - k - g$. Having $k = 256$ and $n = 512$, matrices B, T, D and E are all square of size $g \times g$ where $g = 128$. Circulant permutation matrices of weight $\omega = 1$ is chosen as the building blocks of **H**. This choice is important when performing girth optimisation as outlined in [45]. Permutation matrices cannot have dimension 128×128, otherwise the row weight of **H** will be $\omega_c = 2$. Dividing dimension $k = 256$ by three does not result in a integer. Choosing a 64×64 permutation matrix leads to $\omega_c = 4$

Stellenbosch University http://scholar.sun.ac.za

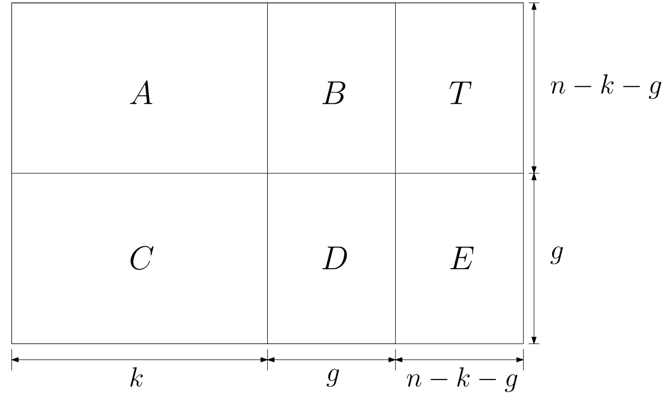


Figure 3.12: Dimensions of sub-matrices within **H**.

$$
\mathbf{H} = \left[\begin{array}{cccc|cc|cc}
H_{00} & H_{01} & H_{02} & H_{03} & H_{04} & H_{05} & H_{06} & H_{07} \\
H_{10} & H_{11} & H_{12} & H_{13} & H_{14} & H_{15} & H_{16} & H_{17} \\
\hline
H_{20} & H_{21} & H_{22} & H_{23} & H_{24} & H_{25} & H_{26} & H_{27} \\
H_{30} & H_{31} & H_{32} & H_{33} & H_{34} & H_{35} & H_{36} & H_{37}
\end{array}\right]
$$

(Columns 0–3 form A (top) / C (bottom), columns 4–5 form B / D, columns 6–7 form T / E.)

Figure 3.13: Layout of **H** using the template in Fig. 3.12.

and $\omega_r = 8$. Such a **H** is given in Fig. 3.13, with H_{ij} a 64×64 matrix for $0 \leq i \leq 3$ and $0 \leq j \leq 7$.

Permutation matrix H_{ij} is rewritten as $I_{ij}(v)$ for $0 \leq v < 64$, where v indicates the column position of the first row's 1. Letting some $I_{ij}(v) = 0$, a (3,6)-regular code can be obtained. Matrix ϕ in Eq. 3.4.13 is designed to be an identity matrix, which guarantees invertibility as required in Eq. 3.4.11 [46]. By using the technique from [46] to find this ϕ, leads to **H** in Eq. 3.4.14.

$$
\mathbf{H} = \left[\begin{array}{cccc|cc|cc}
I_{00}(b_0) & 0 & I_{02}(b_1) & I_{03}(b_2) & I_{04}(b_{12}) & I_{05}(b_{13}) & I_{06}(0) & 0 \\
I_{10}(b_3) & I_{11}(b_4) & 0 & I_{13}(b_5) & 0 & I_{15}(a_0) & I_{16}(b_{15}) & I_{17}(0) \\
\hline
I_{20}(b_6) & I_{21}(b_7) & I_{22}(b_8) & 0 & I_{24}(0) & 0 & I_{26}(a_1) & I_{27}(b_{17}) \\
0 & I_{31}(b_9) & I_{32}(b_{10}) & I_{33}(b_{11}) & I_{34}(a_3) & I_{35}(0) & 0 & I_{37}(a_2)
\end{array}\right] \tag{3.4.14}
$$

Matrices $I_{ij}(v)$ having $v = b_l$ are generated at random. Values $v = a_k$ can be solved in terms of b_l such that ϕ in Eq. 3.4.13 is an identity matrix. A Matlab script has been developed to generate this **H** after which girth optimisations are applied to. Using the girth optimisations explained in [45], cycles of up to length 10 has been removed from **H**, therefore ensuring a girth of 12. These optimisations involve comparing all b_l and a_k against certain criteria as set in

Stellenbosch University http://scholar.sun.ac.za

[45]. If some b_l have to be changed to satisfy this criteria, all a_k are regenerated. The script runs until all girth criteria are satisfied.

3.5 IPC Design

A message passing scheme is selected for IPC between software protocol layers. Since it uses a client-server relationship, it provides synchronisation between two communicating processes. Message passing is recommended in the QNX OS for its speed [11]. Among all QNX's IPC schemes, message passing is the most flexible. For example, an entire message or only part of it can be read by a receiving process. However, Linux Ubuntu 7.10 does not come with message passing as standard. Therefore a basic message passing library for Ubuntu will be designed by using other IPC schemes.

Linux supports two different IPC standards, namely System V and portable operating system interface for Unix (POSIX). Since System V is outdated, this design uses the latter for portability between modern POSIX operating systems. Shared memory is chosen, as it is the fastest IPC scheme in Linux. As mentioned in Section 2.4.1.1, shared memory has no synchronisation support for concurrent data access between two processes. Combining a semaphore with shared memory will ensure synchronisation.

A server will sleep until a client initiates a communications transaction with it. After a client sent its message and received a reply from the server, a transaction is complete. On startup, a server process creates a shared memory structure along with three binary semaphores, SEM_0, SEM_1 and SEM_2, as shown in Fig. 3.14. On creation, SEM_0, SEM_1 and SEM_2 are initialised with values 1,0 and 0 respectively. Afterwards the server calls $sem_wait(SEM_1)$ that requests a lock from SEM_1. Since there is no lock available, the server sleeps.

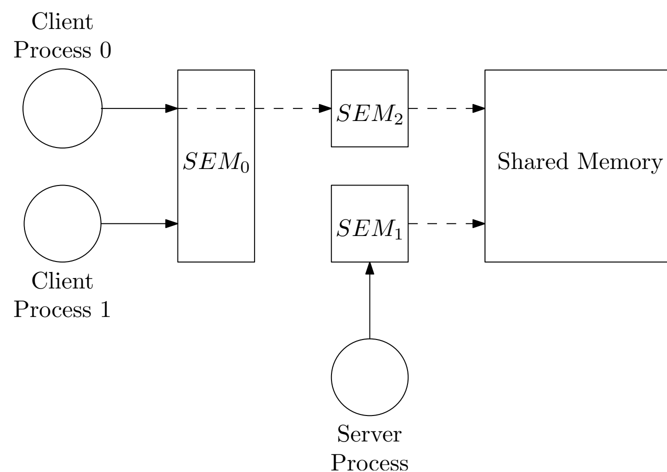


Figure 3.14: Message passing IPC using POSIX semaphores and shared memory.

Stellenbosch University http://scholar.sun.ac.za

Only one client may communicate with a server at any given time, hence multiple clients compete for SEM_0 at first. The client possessing this lock may initiate a communications transaction with the server. A client writes data to the shared memory before calling $sem_post(SEM_1)$ and $sem_wait(SEM_2)$. This causes the client to block and the server to wake up. After processing the client's data in shared memory, the server calls $sem_post(SEM_2)$ and $sem_wait(SEM_1)$, putting itself to sleep and waking the client. A communications transaction will consist of many such function calls from both client and server. A complete transaction is listed as follow :

1. Client : Write size of message in bytes to shared memory.
2. Server : Read this message size.
3. Client : Write message data.
4. Server : Read message data. Process data. Do not reply yet.
5. Server : Write size of reply message.
6. Client : Read size of reply message.
7. Server : Write reply message data.
8. Client : Read reply message data.

After completing the procedure above, the client releases SEM_0 while the server sleeps again.

3.6 Software Protocol Layers

The data link protocol sublayer in Fig. 2.4 implements standards from the TM Space Data Link Protocol of the CCSDS. This thesis will refer to this protocol sublayer as TM. The transport layer in Fig. 2.4 implements a mission specific stop-and-wait ARQ protocol. Other standardised transport layer protocols such as the TCP based Space Communications Protocol Specification Transport Protocol (SCPS-TP) and CCSDS File Delivery Protocol (CFDP) will add unnecessary complexity in this design. The CFDP has the ability to manage files remotely over a space link. However, this is already done by the SCSS running on the SH4. The IS-HS 2 satellite only communicates with one ground station at a time, hence the network-management features of SCPS-TP would be excessive. Fig. 3.15 depicts the interaction between the application layer, ARQ, TM and the channel coding layer. The dashed lines indicate data flow direction between two layers.

Data transfer between software layers happens via message passing IPC as discussed in Section 3.5. A process receiving data acts as a server, while the

Stellenbosch University http://scholar.sun.ac.za

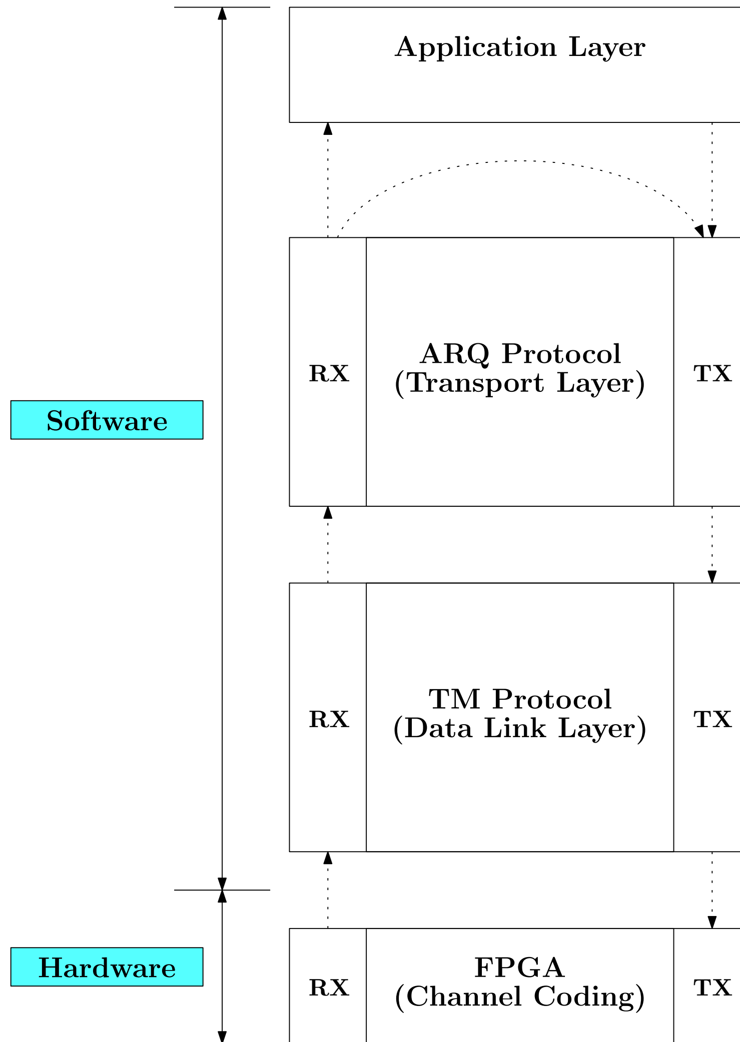


Figure 3.15: Interaction between OSI software layers.

sender is a client. The interface between TM and the channel coding layer differs for both SH4 and FIT-PC platforms. On the SH4 a QNX resource manager [35] is used to write and read from the FPGA whereas the FIT-PC uses USB-to-RS232 modules to communicate with the channel coding layer.

Each module contains both a receive (RX) and transmit (TX) thread. In general the TX thread creates and transmits data. Similarly, the RX thread processes and extracts data from the received packet or frame. Variable length packets are sent and received by ARQ. Fixed length frames are sent and received by TM. A detailed discussion of ARQ packet and TM frame structures are given later in this section.

In Fig. 3.15, the application layer sends file names to ARQ's TX thread for transmission over the link. At the receiver, ARQ's RX thread notifies the application layer of received files. Also note that ARQ's RX and TX threads

Stellenbosch University http://scholar.sun.ac.za

are connected to each other. Upon receiving an ARQ packet, the receiver has to acknowledge reception of this packet. A request for generating and transmitting such an acknowledge is sent from RX to TX via this path. Both the application layer and ARQ's RX thread communicates with ARQ's TX thread, hence the reason for semaphore SEM_0 in Fig. 3.14 of Section 3.5.

3.6.1 TM Protocol

The TM protocol manages data being sent and received over a space link. For example, it could control many devices' data streams, known as virtual channels (VC), simultaneously over a single space link. Data sent and received by TM are fixed length frames of k'-bits. These frames are directly sent to and received from the channel coding layer in the FPGA. Communication takes place between one ground station and the satellite at any given moment, hence VC support is not required for the IS-HS 2. The most important function of TM is to split and reassemble ARQ packets at the transmitter and receiver respectively. A TM frame shown in Fig. 3.16 consists of a header, packet length, data segment and a CRC control field. Assuming (511,484) BCH FEC, the TM frame length of k' bits is less than $k = 484$. Both the SH4 and FIT PC address data on byte level, hence $k - k' = 4$ bits are not used in the 511-bit block. Information necessary for TM frame processing is present in the 48-bit header. The packet length field indicates the length in bytes of a new ARQ packet. Specifically, this field is only present in a frame containing the start of an ARQ packet. The data section contains ARQ packet segments. Finally, the 16-bit CRC field is set by the channel coding layer as stated in Section 3.2.1. Total frame overhead in bytes for TM is :

$$\begin{aligned} N_{ovh_TM} &= N_{hdr_len} + N_{arq_len} + N_{crc_len} \\ &= 6 + 4 + 2 \\ &= 12 \quad bytes \end{aligned} \tag{3.6.1}$$

The 48-bit header is expanded in Fig. 3.17. Three fields are of interest here : the master channel frame count, virtual channel frame count and first header pointer (FHP). The remaining header fields are set as recommended by [1]. The master and virtual channel counts are the same as this design does not use virtual channels. These counters are incremented by one for each TM frame sent. A receiver uses them to detect a dropped frame in a sequence of received frames. A dropped frame causes TM to discard all received frames for the ARQ packet currently being reassembled.

The FHP indicates the offset after the frame header where a new ARQ packet starts. At this offset the 32-bit packet length field shown in Fig. 3.16 is present, followed by an ARQ packet. Setting FHP=0x7FF, a whole TM frame's data section is spanned by a packet. A FHP=0x7FE indicates idle or

Stellenbosch University http://scholar.sun.ac.za

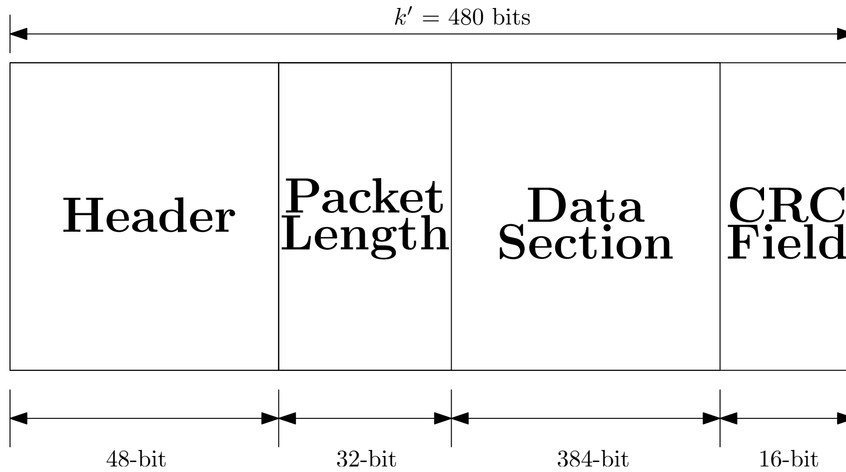


Figure 3.16: TM frame layout.

Master Channel Identifier		Virtual Channel Identifier	Operational Control Field Flag	Master Channel Frame Count	Virtual Channel Frame Count	Transfer Frame Data Field Status				
Transfer Frame Version Number	Spacecraft Identifier					Transfer Frame Secondary Header Flag	Synch. Flag	Packet Order Flag	Segment Length ID	First Header Pointer
2 bits	10 bits	3 bits	1 bit	8 bits	8 bits	1 bit	1 bit	1 bit	2 bits	11 bits

Figure 3.17: TM header layout [1].

unusable data in the whole frame. Such a frame is sent to synchronise the receiver and transmitter's master frame count. Having the FHP < 0x7FE, indicates an offset as mentioned earlier. The FHP works well with ARQ schemes that transmit multiple packets at a time. Due to the stop-and-wait ARQ's operation, only one packet needs to be sent. Therefore, in this implementation, the FHP points to idle data when the first packet ends.

Suppose a 110 byte ARQ packet has to be sent across the space link. The generated TM frames are shown Fig. 3.18. Note that the shaded area in the data section of frame 2 indicates idle data.
A detailed TM frame processing implementation is provided in the next chapter.

3.6.2 ARQ Protocol

Since TM cannot guarantee reliable data transfer, an ARQ strategy existing in the transport layer will ensure reliable data transfer across the space link. ARQ's transmitter receives a file from the application layer, after which it gets split into ARQ packets to be sent over the link. A receiving ARQ then assembles this file from the received packets.

Stellenbosch University http://scholar.sun.ac.za

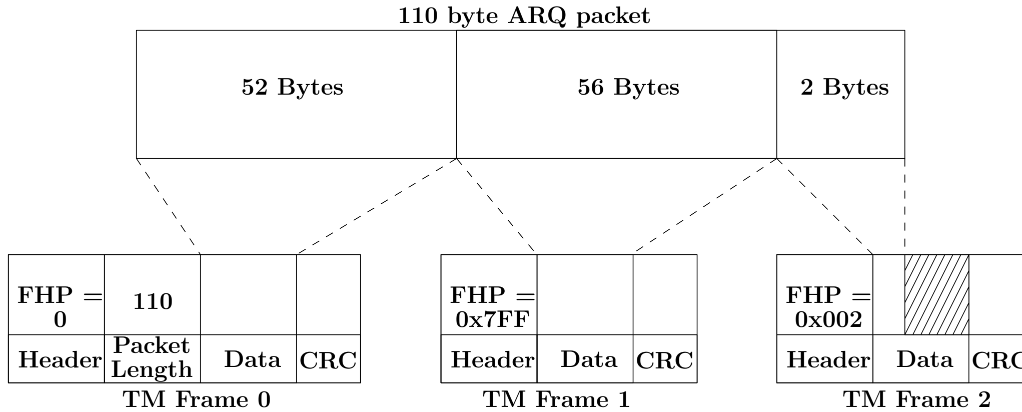


Figure 3.18: Setting FHP when ARQ packet spans multiple TM frames.

In an ARQ strategy, packets sent by the transmitter must be acknowledged by the receiver. Transmitted packets contain a sequence number in its header. Upon receiving this packet, the receiver puts this sequence number in an ARQ acknowledge packet that is destined for the transmitter. Various ARQ strategies exist for handling this send and acknowledge scheme [47] :

- **Stop-And-Wait ARQ** : Only one packet is sent at a time. The transmitter measures the duration between sending a packet and receiving its acknowledge. Should a threshold duration be exceeded, the transmitted packet or its acknowledge is assumed to be lost. The same packet is then retransmitted. Data transmission is aborted when a maximum retransmit count is exceeded for the current packet.

- **Go-Back-N ARQ** : This is an instance of sliding window ARQ. In this window a sequence of l packets are sent whilst having l outstanding acknowledges. Packets are sent without waiting for an acknowledge between packet transmissions. Acknowledges are received in the same order of packet transmission. However, should packet i for $0 \leq i \leq l$ contain errors, the receiver will reject all subsequent received packets. The receiver sends a reject message, indicating it expects packet i. All packets from i are then retransmitted. Should the reject message get lost, a time out similar to stop-and-wait ARQ causes all packets from i to be retransmitted.

- **Selective-Reject ARQ** : The transmitter continually sends packets and their respective acknowledges. A negative acknowledge message is received when a packet at the receiver contains errors. Unlike go-back-N ARQ, only the rejected packet is retransmitted. Acknowledges that get lost, cause a time out at the transmitter after which the particular packet is retransmitted.

Stellenbosch University http://scholar.sun.ac.za

Go-Back-N and selective reject schemes have the best channel utilisation of all ARQ types [47]. Delays between subsequent packet transmissions are minimal, hence the data throughput is high. This implementation is expected to work with file sizes of 10 kB and less, therefore a high throughput ARQ scheme will be unnecessary. Due to the simplicity of a stop-and-wait ARQ [47], it is chosen for this design. It is shown in the next chapter, that Stop-And-Wait ARQ has sufficient data throughput for transferring 10 kB files during an aircraft pass.

3.6.2.1 Packet Structure

In Fig. 3.19 an ARQ packet consists of both a header and data section. Each ground station and the satellite have a unique 32-bit ID assigned to it. A transmitter includes both the receiver's and its own ID in the destination and source address fields respectively. The satellite has knowledge of each ground station's ID. Since the satellite initiates communication, the ground station will get the satellite's ID from the initial ARQ packets. Ground stations may be closely located to each other and could all receive the same ARQ packet. By checking the packet's destination ID, a ground station can decide if this packet is intended for it or not.

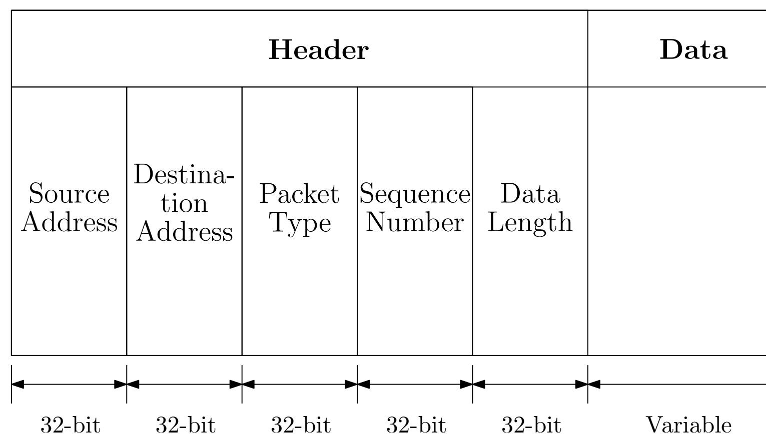


Figure 3.19: An ARQ packet structure.

The packet type field distinguishes between a normal or acknowledge type. Normal types are further divided into four subtypes : *first*, *message*, *last* and *single*. Type *first* signals the first packet of a new file. A packet not being either the first or last is classified as *message*. Type *single* is used for special cases when a file is only one packet long. Finally, type *last* is the final packet of the current file.

The sequence number field is used by the send-acknowledge scheme. After sending a packet and receiving its acknowledge, the transmitter increments this sequence number for the next packet. When acknowledging a sequence number, the receiving ARQ knows the next sequence number to expect. Should

Stellenbosch University http://scholar.sun.ac.za

the next packet's number not match this expected number, a retransmitted packet has been received. Retransmitted packets only get acknowledged as their contents are discarded by the receiver. Lastly, the packet length field indicates the length in bytes of the data section. The total packet overhead in bytes is :

$$\begin{aligned} N_{ovh_ARQ} &= N_{src_adr} + N_{des_adr} + N_{pck_typ} + N_{seq_num} + N_{pck_len} \\ &= 4 + 4 + 4 + 4 + 4 \\ &= 20 \quad bytes \end{aligned} \tag{3.6.2}$$

A detailed packet processing flow diagram is provided in the next chapter.

3.6.2.2 Round Trip Time Calculation

A round trip time (RTT) refers to the delay between sending an ARQ packet and receiving its acknowledge [3]. This parameter accounts for delays introduced by data processing in the various OSI layers, and signal propagation delays through the physical link. The RTT will be the lower bound on ARQ's time out value. Both satellite and ground station platforms consist of many hardware devices and software layers, making it difficult to calculate an accurate RTT. This parameter is measured when both ground station and satellite platforms are implemented. A measurement setup is shown in Fig. 3.20. The current time t_A is taken when an ARQ packet leaves A. This packet travels through the downlink until it reaches ARQ at B on the ground station. An acknowledge travels from B over the uplink before reaching C where the current time t_C is taken. Time stamps t_A and t_C are accurate to the millisecond when using Unix's *clock_gettime*() function. Finally, RTT $= t_C - t_A$.

3.7 FEC Block Error Rate Simulation

A receiver's BLER at different E_b/N_o SNRs is simulated when transmitting data over a noisy AWGN channel. Simulation is performed in Matlab and on FPGA for both BCH and LDPC FEC schemes. Each simulation platform consists of the following components :

- Message Generator : Creates a random k-bit message $\mathbf{x}$.
- FEC Encoder : Encodes message $\mathbf{x}$ into n-bit codeword $\mathbf{c}$ for either LDPC or BCH.
- QPSK Modulator : Modulates codeword $\mathbf{c}$.
- AWGN Channel : Adds Gaussian distributed amplitude noise to both **I** and **Q** channels.

Stellenbosch University http://scholar.sun.ac.za

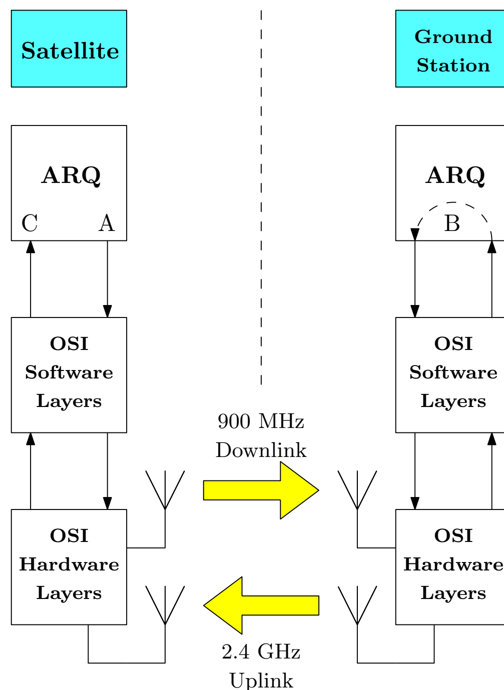


Figure 3.20: Packet round trip time measurement.

- QPSK Demodulator : Demodulates the noisy QPSK signal.
- FEC Decoder : Decodes the received codeword $\mathbf{c}^*$ using either LDPC or BCH.

A detailed flow diagram of the simulation is given in Fig. 3.21. At first, the range of simulated SNRs along with the maximum number of iterations per simulated SNR are created. This is followed by the simulation's main loop where a message is created, encoded, sent over an AWGN channel and decoded for each iteration.

Beginning at the main loop, a random message $\mathbf{x}$ having k bits is generated. This message gets encoded for either a BCH or LDPC scheme. The resulting codeword $\mathbf{c}$ is then translated to QPSK symbols by using Fig. 3.1a as reference. Symbol amplitude A_{qpsk} is derived from the simulated SNR by using Eq. A.2.9 :

$$\frac{A_{qpsk}^2}{\sigma^2} = \frac{2E_b}{N_o} \tag{3.7.1}$$

where noise variance $\sigma^2 = 1$. After adding noise to the QPSK symbols, demodulation is performed. BCH uses the constellation diagram in Fig. A.1a to translate the received symbols into codeword $\mathbf{c}^*$. Section 3.4.2 explained that MS LDPC decoders require input LLRs for each bit in $\mathbf{c}^*$. These LLRs are

Stellenbosch University http://scholar.sun.ac.za

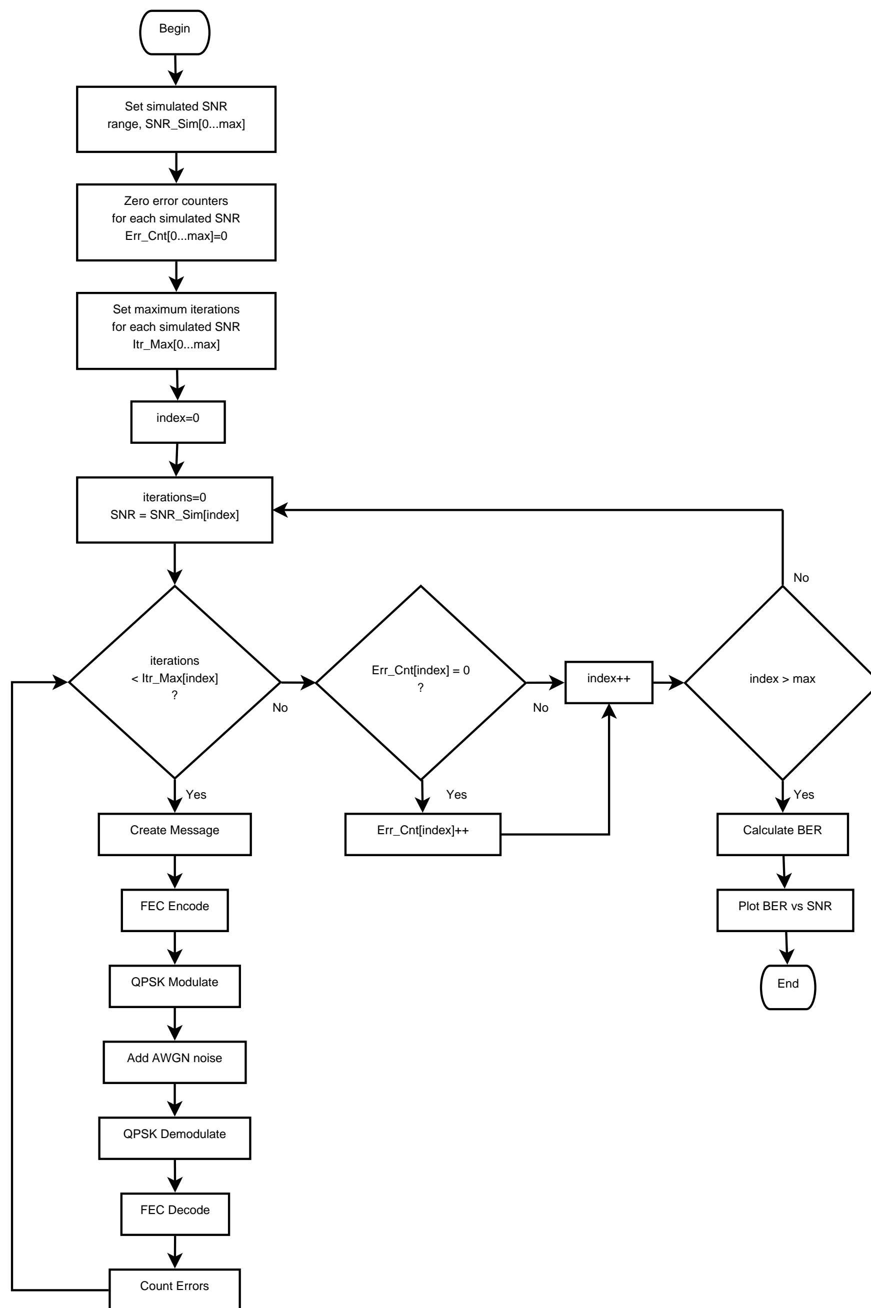


Figure 3.21: General operation of simulator.

calculated inside the decoder by only using $Pr(c_i^* = 0)$ as measured by the demodulator.

An example of this measurement is now explained. Fig. 3.22 shows the position of a received symbol, S_R, on the QPSK constellation diagram. As S_R falls in S_2's quadrant, S_R is interpreted as S_2. Having $S_2 = [1, 0] = [b_0, b_1]$,

Stellenbosch University http://scholar.sun.ac.za

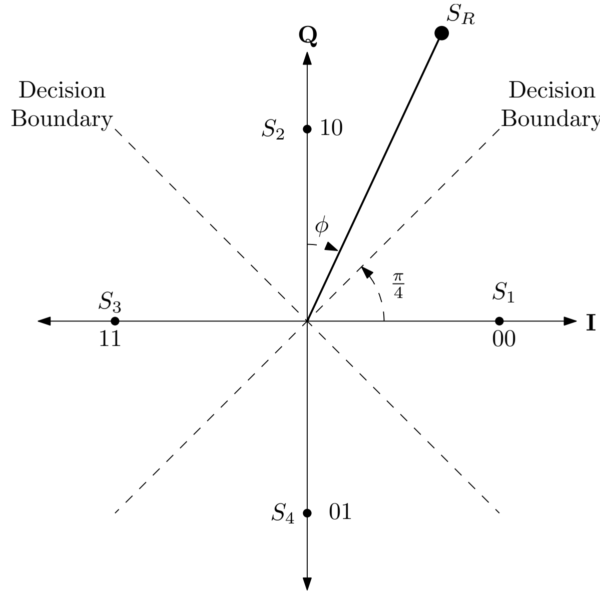


Figure 3.22: Bit probability decision making.

probability values are assigned to bits b_0 and b_1. Since S_R lies between S_1 and S_2, the demodulator only decides between $b_0 = 1$ or $b_0 = 0$ because $b_1 = 0$ for both S_1 and S_2. Using angle ϕ in Fig. 3.22 to decide on b_0, the following probabilities for b_0 and b_1 are obtained :

$$Pr(b_0 = 0) = 1 - Pr(b_0 = 1) = 1 - \left(\frac{\frac{\pi}{2} - \phi}{\frac{\pi}{2}}\right) \tag{3.7.2}$$

$$Pr(b_1 = 0) = 1 \tag{3.7.3}$$

A BCH decoder receives $\mathbf{c}^*$ and either declares failure or success after decoding. In case of a failure, $Err_Cnt[index]$ gets incremented by one. The LDPC decoder receives a vector containing $Pr(c_i^* = 0)$ for $0 \leq i \leq n$. After decoding completes, $\mathbf{c}$ is compared with the returned $\mathbf{c}'$. Counter $Err_Cnt[index]$ gets incremented when $\mathbf{c} \neq \mathbf{c}'$.

After completing $Itr_Max[index]$ iterations for the current SNR, the next SNR is selected. Prior to this step, $Err_Cnt[index]$ gets incremented when $Err_Cnt[index] = 0$. This prevents the BLER graph not displaying Err_Cnt correctly on a log scale. When all SNRs have been simulated, the BLER for each SNR is calculated :

$$BLER = \frac{Err_Cnt[index]}{Itr_Max[index]} \tag{3.7.4}$$

where ($Itr_Max[index]$) represents the total FEC blocks received by the decoder. The BLER values of Eq. 3.7.4 are then plotted against all SNR values.

Stellenbosch University http://scholar.sun.ac.za

3.7.1 Matlab Simulation

The software implementation of Fig. 3.21 is done in Matlab. Message $\mathbf{x}$ is a $1 \times k$ row vector containing ones and zeros. These binary values are randomly generated using Matlab's $rand()$ function.

Matlab also comes with a BCH toolkit, hence encoding is done with function $\mathbf{c} = bchenc(\mathbf{x}, n)$. Functions have been written for LDPC encoding which performs the encoding as per Section 3.4.3.

Since QPSK has two bits per symbol, the modulated codeword $\mathbf{w}$ is a $(1 \times n/2)$ row vector. This vector has complex numbers whose real and imaginary parts represent the $\mathbf{I}$ and $\mathbf{Q}$ channels respectively. Using Matlab's $randn()$, Gaussian noise having variance $\sigma^2 = 1$ is added to both $\mathbf{I}$ and $\mathbf{Q}$ in $\mathbf{w}$.

Using the BCH toolkit, $\mathbf{x} = bchdec(\mathbf{c}^*, k, n)$ is used for BCH decoding. A function implementing the MS decoder has been written for LDPC. As the iterative message passing decoder requires the parity matrix's structure, function $\mathbf{c'} = MinSum(\mathbf{c}^*_{prob}, \mathbf{H})$ is called with $\mathbf{H}$ as argument. Argument $\mathbf{c}^*_{prob}$ is a $(1 \times n)$ vector of $Pr(c_i^* = 0)$.

3.7.2 FPGA Simulation

This hardware simulator verifies Matlab's BLER simulation results by using both BCH and LDPC's implementation on FPGA. Fig. 3.23 illustrates the simulator layout. A PC running a C application houses some components of the simulator in Fig. 3.21. An Altera Cyclone III EP3C120F780 FPGA, as part of a Cyclone III DSP development board, contains both the encoder and decoder for either BCH or LDPC. Bidirectional communication between the PC and FPGA occurs via a high speed 1.5 Mbps FTDI FT-232BL USB-To-UART module.

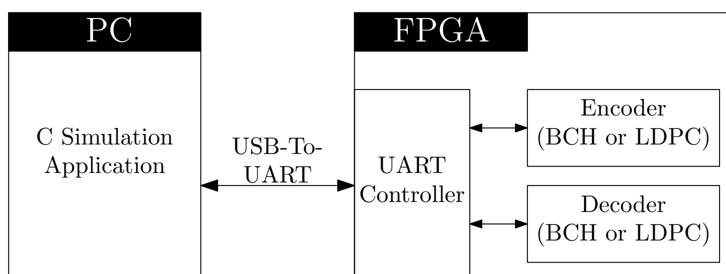


Figure 3.23: Hardware based BER simulation.

Sending data from the PC to the FPGA is preceded by a control byte b_{cntl}. This byte's value signals the FPGA to route incoming data to either the encoder or decoder. Values of b_{cntl} are listed in Table 3.2. After receiving all data for either the encoder or decoder, the FPGA acknowledges its reception by sending byte 0x00 to the PC.

Stellenbosch University http://scholar.sun.ac.za

Value of b_{cntl}	Description
0	Send data to encoder.
2	Send data to decoder.

Table 3.2: PC to FPGA control byte values.

During simulation, message $\mathbf{x}$ is created on the PC as an array of bytes having a total of k bits. These are created at random using the standard $rand()$ function. After sending $\mathbf{x}$ to the FPGA for encoding, the PC waits for the returned codeword $\mathbf{c}$. By calling POSIX $read()$ on the UART, the PC sleeps until it receives data from the encoder.

Similar to Matlab's simulation, $\mathbf{c}$ gets modulated after which Gaussian noise is added. Since the C language doesn't come with Gaussian random number generators, the GNU scientific library for C is used to generate such numbers. After demodulation, $\mathbf{c}^*$ is sent for decoding. The hardware decoder module for either LDPC or BCH does not declare a decoding failure. It simply discards a codeword $\mathbf{c}^*$ without notifying external components. Therefore the PC sets an interrupt timer, before calling $read()$ to wait for the decoder. This timer allows the C application to wake up and continue simulation instead of waiting indefinitely due to a decoding failure.

After simulation, the BLER data is saved to a file. Using Matlab, this data is plotted for comparison to the Matlab simulation results. These simulation results are shown in Sections 5.2 and 5.3 of Chapter 5.

3.8 Summary

A FEC block length of $n = 511$ bits have been determined in this chapter. Implementing a (511,484) BCH, capable of correcting $t = 3$ errors, would be sufficient for the estimated link quality. Detailed descriptions of the channel coding modules for both satellite and ground station platforms, have also been provided. Pseudo code segments also described the behaviour of both BCH and LDPC decoders. A LDPC code implementing a soft decision MS decoder will provide superior performance compared to a hard decision based decoder. Applying the two optimisation strategies mentioned, will further enhance its performance. A message passing IPC scheme that uses shared memory and semaphores have been designed for Linux Ubuntu. This will provide a fast and synchronised communications medium between ARQ and TM. Frame and packet structures for TM and ARQ have also been presented. Finally, a Matlab and FPGA simulation strategy for FEC implementations have been described.

Chapter 4 will provide hardware architectures for all channel coding modules. The message passing API for IPC is also described. Finally, detailed frame and packet processing routines for TM and ARQ are also presented.

Stellenbosch University http://scholar.sun.ac.za

Chapter 4

Implementation

This chapter starts off with a description of existing hardware configurations for both satellite and ground station platforms. An implementation for channel coding on a FPGA is then provided. Finally, the packet processing routines for both TM and ARQ are provided.

4.1 Existing Hardware Layout

Section 2.2 showed which hardware modules are connected to each other for both satellite and ground station platforms. This section explains some of these components' interfaces in detail which are necessary for implementation of channel coding. Before starting, the term $de - assert$ refers to a logic high state or a '1' whereas $assert$ indicates a logic low state or '0'.

4.1.1 Satellite Platform

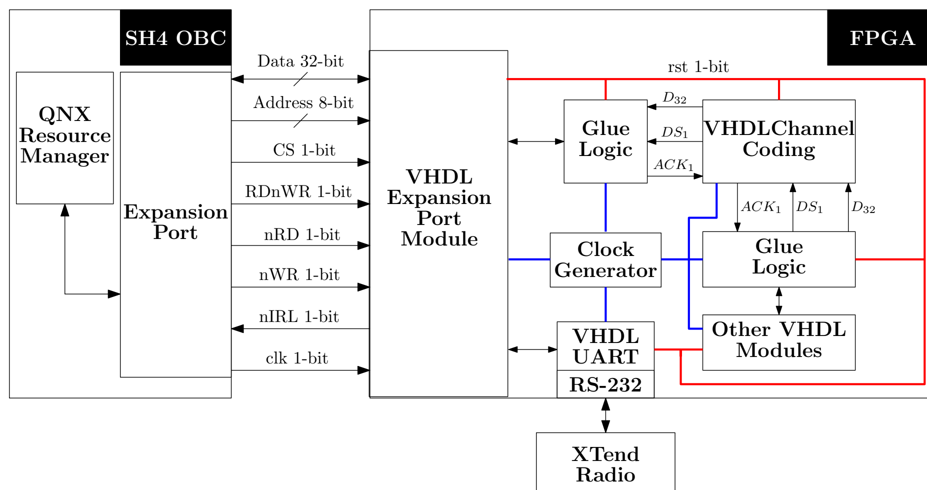


Figure 4.1: Channel coding on the satellite platform.

Stellenbosch University http://scholar.sun.ac.za

The Sun Space SH4 OBC interfaces with a Xilinx ML501 Virtex 5 FPGA development board via its expansion port as per Fig. 4.1. A Digi International 9XTend-PKG-R radio used by the downlink is also connected via RS-232 to the FPGA. Black connector lines between components indicate data direction.

Expansion port lines entering the FPGA are described and listed in Table 4.1. This interface provides bidirectional communication between the SH4 and FPGA. A VHDL expansion port module written by [35] allows the SH4 to access a command register, status register and data FIFOs on the FPGA. The command register is 32 bits wide and is used to reset all modules on the FPGA. Modules are reset by de-asserting the red rst line. A transmit data FIFO, buffers data between the SH4 and XTend radio whereas a receive FIFO stores data from the channel coding module. The status register indicates whether these FIFOs are full or empty. By asserting $nIRL$, the FPGA interrupts the SH4 when the receive FIFO is full. Only when the SH4 reads the status register, this interrupt gets acknowledged after which $nIRL$ is de-asserted.

Expansion Port Line	**Description**
Data	CPU's 32-bit data bus.
Address	An 8-bit segment of the CPU's address bus.
CS	Chip Select line. Asserted when SH4 access FPGA .
RDnWR	De-asserted when SH4 reads from FPGA at value on address bus. Asserted when SH4 writes to FPGA at value on address bus.
nRD	Asserted when reading value from data bus.
nWR	Asserted when writing value on data bus.
nIRL	Interrupts SH4 when asserted.
clk	Clock from SH4.

Table 4.1: SH4 to FPGA expansion port lines and their description.

Channel coding interfaces with this expansion port module and other FPGA modules via glue logic. Glue logic represents a module that connects two modules, having different interfaces, with each other. Channel coding's interface consists of three entities : a 32-bit data bus D_{32}, data strobe D_1 and acknowledge line ACK_1. Note that these symbols' subscripts indicate the number of bits represented by each line. The expansion port's FIFO contains a 32-bit data input, hence channel coding also uses a 32-bit data bus.

4.1.2 Ground Station Platform

The FIT-PC connects to a Xilinx Spartan 3E starter kit FPGA board and a 9XTend-PKG-R radio via USB-To-RS232 modules in Fig. 4.2. Black connector lines indicate the direction of data flow between components.

Stellenbosch University http://scholar.sun.ac.za

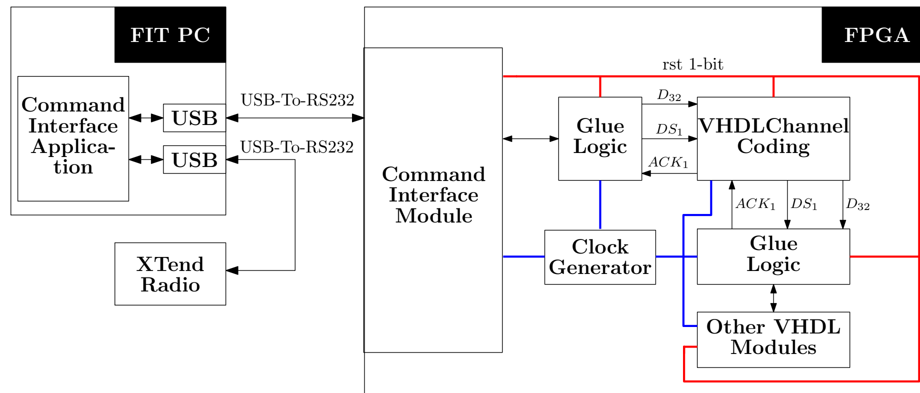


Figure 4.2: Channel coding on the ground station platform.

A command interface application on the FIT-PC transfers TM frames to the FPGA. The FPGA command interface module, that were previously developed, receives and acknowledges these TM frames. This module may also receive a reset command from the FIT-PC's command interface application. All FPGA modules are then reset by asserting the red *rst* line for a short period.

Channel coding also interfaces via glue logic to the command interface module and other FPGA modules. In order to keep the design procedure simple, the same interface for channel coding's modules is used as mentioned in Section 4.1.1.

4.2 Channel Coding Implementation

4.2.1 Module Interface

Fig. 3.5 from Section 3.2 showed the order in which channel coding's modules are connected to each other. All modules connect with the same interface which allows modules to be easily added, removed or replaced. No additional glue logic is required when connecting any two modules with each other. This design is very convenient when changing from BCH to LDPC and vice versa.

An interface for a general channel coding module is shown in Fig. 4.3. The width in bits of each input and output is provided below the arrowed line. All inputs and outputs are synchronised to the rising edge of the clock. Inputs for clock, *clk*, and asynchronous reset, *rst*, are obtained from the blue and red lines in Fig. 4.2. Data strobe *ds_in* is de-asserted when valid data is available on *dat_in*. Input *ds_in* is rising-edge sensitive, meaning that the line must be asserted for at least 1 clock cycle before de-asserting. After latching *dat_in*, the module de-asserts acknowledge *ack_out* to inform the previous module that *dat_in* has been received. Line *ack_out* is de-asserted for 1 clock cycle. When outputting values on *dat_out*, data strobe *ds_out* is de-asserted until

Stellenbosch University http://scholar.sun.ac.za

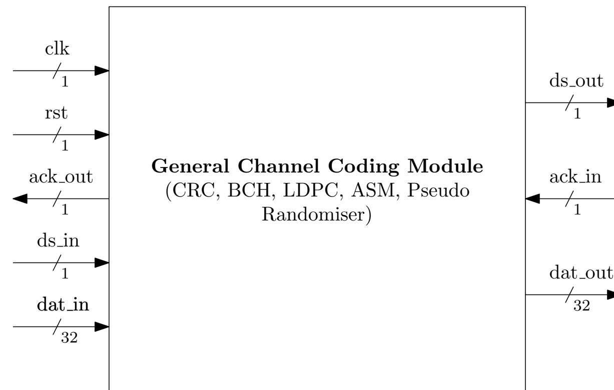
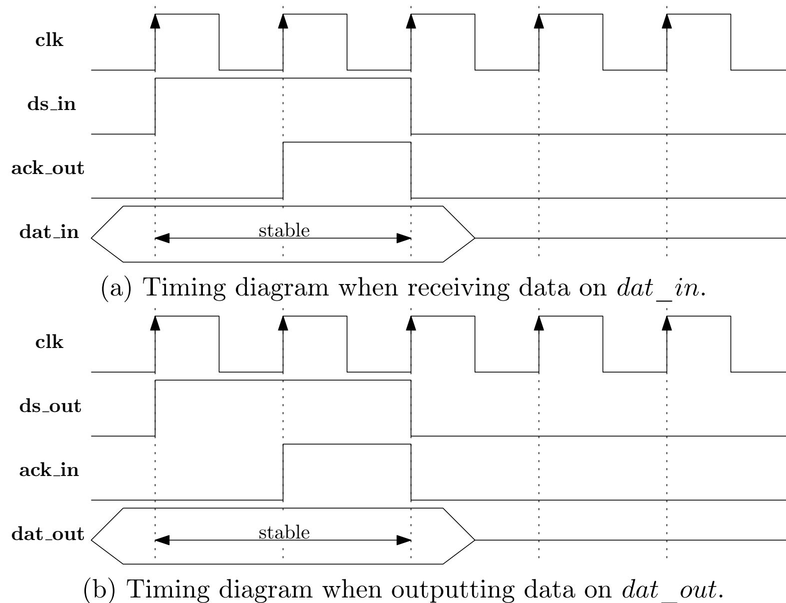


Figure 4.3: Interface of a general channel coding module.

acknowledge *ack_in* gets de-asserted. This is followed by assertion of *ds_out*. Timing diagrams for receiving data on *dat_in* and output data on *dat_out* are provided in Figs. 4.4a and 4.4b respectively. The vertical arrows on *clk* indicate that all other lines are toggled on the rising edge of *clk*.



(a) Timing diagram when receiving data on *dat_in*.

(b) Timing diagram when outputting data on *dat_out*.

Figure 4.4: Timing diagram of the channel coding module from Fig. 4.3.

4.2.2 Module Internals

Each module implements a finite state machine (FSM), which controls when the module inputs, outputs and processes data. A single VHDL process is used for the FSM as recommended by Xilinx [48]. The CRC, pseudo randomiser and ASM modules, processes data as it arrives on *dat_in*. In general this is

Stellenbosch University http://scholar.sun.ac.za

true for all channel coding modules except for FEC decoding, which requires 512 bits of data before processing starts. A general state diagram of a module is provided in Fig. 4.5.

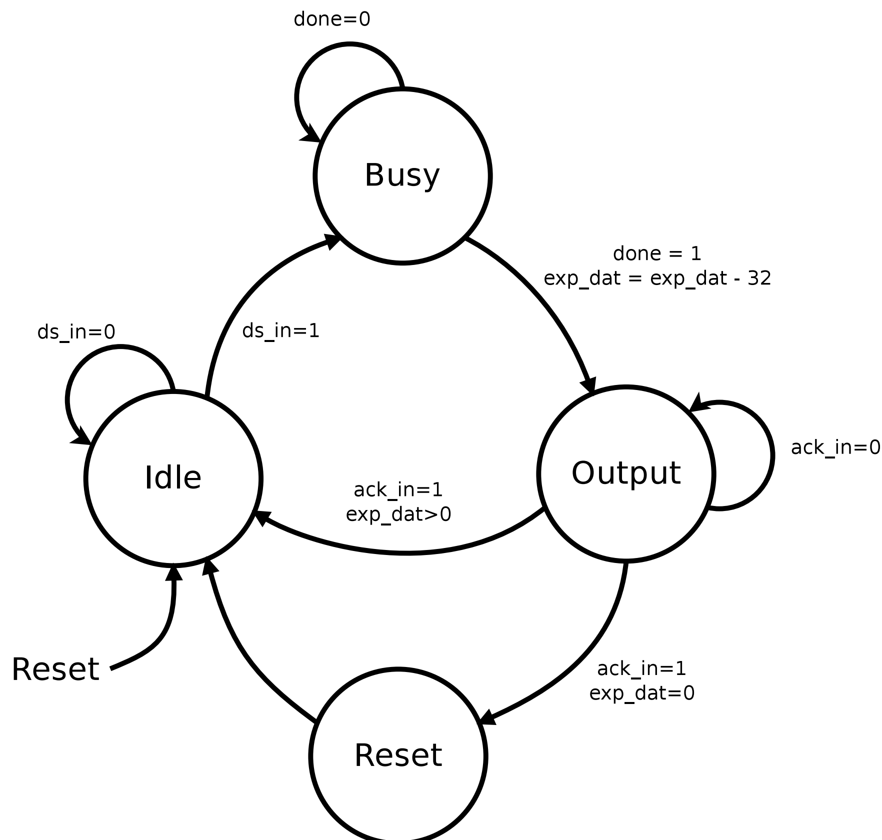


Figure 4.5: State machine diagram of the module in Fig. 4.3.

Four states exist namely *Idle*, *Busy*, *Output* and *Reset*. Upon de-asserting rst in Fig. 4.3, the *Idle* state is entered. During reset, an expected data counter, exp_dat, is set to the number of data bits the module expects. Since dat_in is 32 bits wide, exp_dat gets decremented by 32, after data has been latched.

During *Idle* the module waits for valid data on dat_in. Upon receiving data, the *Busy* state is entered where data processing occurs. To keep this state diagram simple, the *Busy* state represents a collection of many sub-states during data processing. After data processing finishes, the *Output* state puts data on dat_out for the next module. Variable exp_dat is evaluated before returning to *Idle*. Should $exp_dat = 0$, no more data is expected by this module. A *Reset* state is then entered where all internal registers and exp_dat are restored as done for an asynchronous reset.

Stellenbosch University http://scholar.sun.ac.za

4.2.3 Parallel Data Processing

Both CRC and pseudo randomiser modules use shift registers when processing their input data. As an example, Fig. 3.7 shows a serial implementation of the randomiser's register. Since the module accepts 32 bit data, it would require 32 clock cycles to process data with this register. To minimise this latency, a parallel processing scheme is adopted. A technique proposed by [49] transforms a serial shift register into a parallel shift register. Continuing the randomiser example, a single shift and feedback operation is characterised by Eq. 4.2.1.

$$\mathbf{x'} = \mathbf{T}\mathbf{x} \tag{4.2.1}$$

$$\begin{bmatrix} x'_8 \\ x'_7 \\ x'_6 \\ x'_5 \\ x'_4 \\ x'_3 \\ x'_2 \\ x'_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_8 \\ x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \end{bmatrix}$$

Vector $\mathbf{x}$ is the serial register's state before a shift-feedback operation. Matrix $\mathbf{T}$ applies a single shift and feedback operation to $\mathbf{x}$ in Fig. 3.7. Therefore, when multiplying any given state of $\mathbf{x}$ with $\mathbf{T}$, the next state $\mathbf{x'}$ is obtained. The i-th state of $\mathbf{x}$, denoted $\mathbf{x}^{(i)}$, is equivalent to :

$$\mathbf{x}^{(i)} = \mathbf{T}^i\mathbf{x} \tag{4.2.2}$$

Matrix $\mathbf{T}^{(i)}$ is pre-computed in Matlab using modulo-2 arithmetic. Each row j in $\mathbf{T}^{(i)}$ indicates which bits of $\mathbf{x}$ is XORed to obtain the value of row j in $\mathbf{x}^{(i)}$. An example will now be given. Using Eq. 4.2.2 and assuming $\mathbf{T}$ from Eq. 4.2.1, element $x_8^{(2)}$ from $\mathbf{x}^{(2)} = [x_8^{(2)}, x_7^{(2)}, x_6^{(2)}, x_5^{(2)}, x_4^{(2)}, x_3^{(2)}, x_2^{(2)}, x_1^{(2)}]$ is computed as follow :

$$x_8^{(2)} = x_8 + x_6 + x_5 + x_4 + x_3 + x_2 \tag{4.2.3}$$

where '+' indicates XOR. Similarly the other elements from $\mathbf{x}^{(2)}$ are calculated. This technique allows $\mathbf{x}^{(2)}$ to be computed in less than a clock cycle. Since the randomiser register is only 8-bits wide, it can be configured to randomise a 32-bit word in less than a clock cycle. A 32-bit register, $\mathbf{y}_{32}$, is initialised with values $\mathbf{y}_{32} = [\mathbf{x}^{(24)}, \mathbf{x}^{(16)}, \mathbf{x}^{(8)}, \mathbf{x}^{(0)}]$, which have been pre-computed in Matlab. Register $\mathbf{y}_{32}$ now contains the first 32 bits as output by a serial LFSR. Input data from *dat_in* is bit-wise XORed with register $\mathbf{y}_{32}$, before outputting the

Stellenbosch University http://scholar.sun.ac.za

result on *dat_out*. Finally, $\mathbf{y}_{32}$ is updated by letting $\mathbf{y}_{32} \leftarrow T^{32} \times \mathbf{y}_{32}$ such that $\mathbf{y}_{32} = [\mathbf{x}^{(24+32)}, \mathbf{x}^{(16+32)}, \mathbf{x}^{(8+32)}, \mathbf{x}^{(0+32)}]$.

The polynomial division circuit for CRC also uses a LFSR. By determining $\mathbf{T}$, it could parallelised in a similar way.

4.3 BCH Implementation

4.3.1 Polynomial Division Register

An example circuit in Fig. 4.6 divides polynomial $c(X)$ by $g(X)$ until $r(X)$ remains in the register. Letting $g(X) = 1 + X + X^2$, results in $r(X)$ having an order of at most 1. Polynomial $g(X) = g_0 + g_1X + g_2X^2$ is depicted as $\mathbf{g} = [g_0, g_1, g_2] = [1, 1, 1]$ in Fig. 4.6. Similarly the coefficients of $c(X)$ are denoted $\mathbf{c} = [c_0, c_1, \cdots, c_{n-1}]$. Register r is zeroed before division starts. Starting at c_{n-1}, register $\mathbf{c}$ is shifted into r, one bit per clock cycle. As soon as r_1 is a 1, the next shift will result in $r(X) = r_0 + r_1X + X^2$. In this case $g(X)$ gets subtracted from $r(X)$ such that $r(X)$ is of order 1 at most. This circuit essentially performs long division, as illustrated by the binary long division example from Fig. 3.6 in Section 3.2.1.

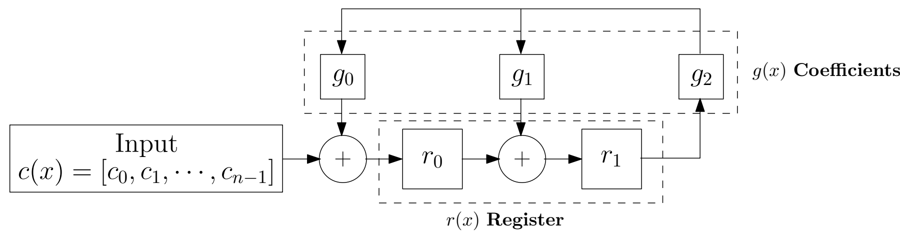


Figure 4.6: Serial implementation of a polynomial division LFSR.

4.3.2 Encoder

This module's state machine behave exactly as depicted in Fig. 4.5. Upon receiving a value on *dat_in*, it gets processed before being output on *dat_out*. Since BCH uses polynomial division for encoding, it uses a LFSR to implement this division operation. After processing a complete TM frame, this 27-bit LFSR contains the remainder after division. Exactly how this value is handled is explained later. By applying the parallelisation technique mentioned in Section 4.2.3, this module also processes a 32-bit word in one clock cycle.

Fig. 4.7 shows a codeword as output by the encoder module. Since the module uses a 32-bit data bus, the 511-bit codeword is modified to be 32-bit aligned. Before doing so, it is noted that a polynomial view of codeword c(X) has its highest order term on the left in Eq. 2.5.10. By adding a 0

Stellenbosch University http://scholar.sun.ac.za

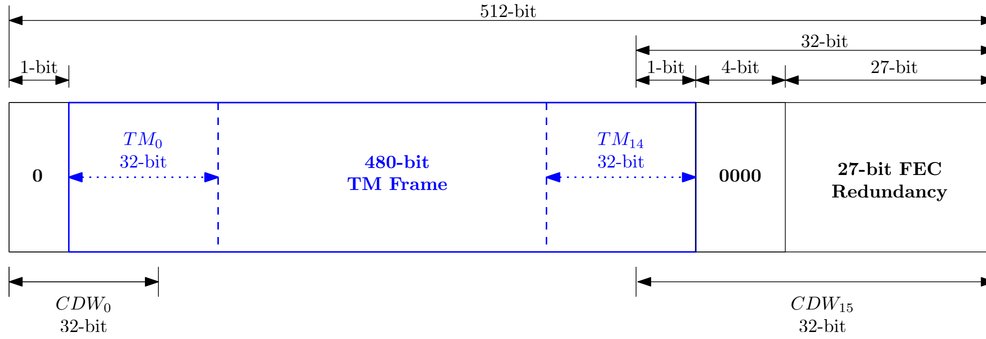


Figure 4.7: Codeword as constructed by BCH encoder.

left of this term aligns the codeword while leaving c(X) unchanged. The first TM frame segment to be received and processed, is TM_0 in Fig. 4.7. After processing TM_0, the first codeword segment, CDW_0, is output. Processing continues until the final segment, CDW_{15}, is constructed. Here, the all zero 4-bit segment in CDW_{15} is unused space, which has been explained in Section 3.6.1. The 27-bit FEC redundancy is obtained from the polynomial division LFSR, as mentioned earlier.

4.3.3 Decoder

4.3.3.1 State Diagram

Fig. 4.8 shows the decoder's state diagram. On asynchronous reset, it enters the *Idle* state. Received 32-bit data is stored in a FIFO, while at the same time being passed to the syndrome calculation module. Here, the received codeword $c^*(X)$ is simultaneously divided by each primitive polynomial $\phi_i(X)$, as mentioned in Section 3.3.2. The decoder returns to *Idle* if it expects more data for the current $c^*(X)$.

After receiving all 512 bits of $c^*(X)$, the calculated $2t = 6$ syndromes are moved to the Berlekamp-Massey module which finds $\sigma(X)$. Coefficients of $\sigma(X)$ are then moved to the Chien search module. Before starting, $counter = 32$, since 32-bit segments of $c^*(X)$ are processed at a time. Polynomial evaluation $\sigma(\alpha^i)$ is performed as depicted in Fig. 3.9 from Section 3.3.2. After evaluating $\sigma(\alpha^i)$ for an α^i, *counter* gets decremented, until 32 such evaluations have been done. This leads to 32-bit error vector **e**, which is XORed with a 32-bit segment from the $c^*(X)$ FIFO. The corrected segment is now output on *dat_out* for the next module. Chien search continues until all data in the FIFO has been processed and output. Finally, a *Reset* state clears all registers and restores counters before entering the *Idle* state again.

Stellenbosch University http://scholar.sun.ac.za

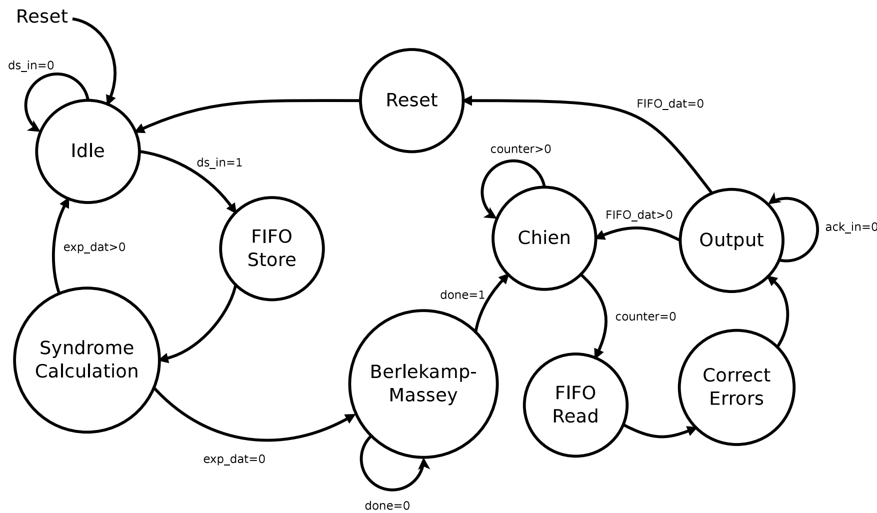


Figure 4.8: State machine diagram of a BCH decoder.

4.3.3.2 Hardware Layout

A hardware layout is presented in Fig. 4.9. The FSM component implements the state machine in Fig. 4.8. Bold blue lines represent a collection of wires that allows the FSM to control a component and read its status bits. These status bits indicate whether a component is done with its current operation, therefore allowing the FSM to make state change decisions if necessary. Control lines are used for example to read from the FIFO during Chien search.

Syndrome coefficients are transferred sequentially to the BM module via the indicated interface. A value on data bus D_9 is transferred when data strobe DS_1 is de-asserted. The BM module acknowledges data reception by de-asserting ACK_1. Coefficients are 9 bit GF values as explained in Section 3.3, hence D_9 is also 9 bits wide. Similarly data is transferred between the BM and Chien modules. After error vector e is ready, the FSM issues a read command to the FIFO. Value $FIFO_Data$ gets XORed with e before output on dat_out.

Stellenbosch University http://scholar.sun.ac.za

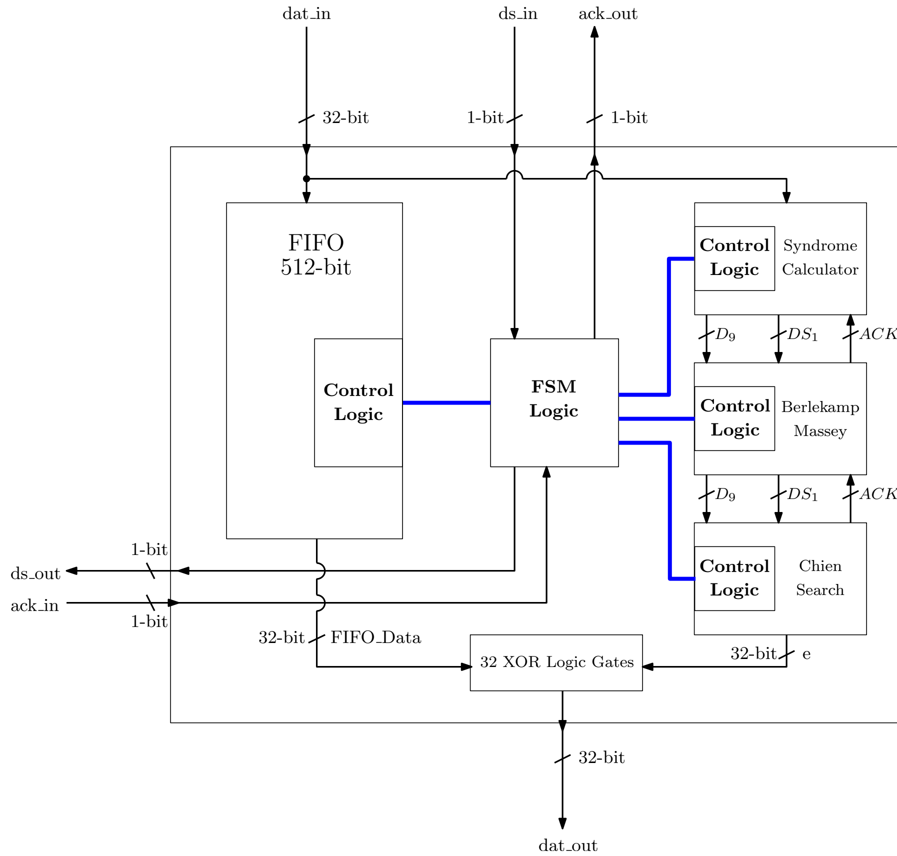


Figure 4.9: A BCH decoder's hardware layout.

4.4 LDPC Implementation

4.4.1 Matrix Multiplication Circuit

A circuit for multiplying a column vector with a cyclic matrix, is provided in this section. Suppose a (1×4) row vector $\mathbf{x}$ has to be multiplied with (4×4) cyclic matrix $\mathbf{T}$. Example values are provided below :

$$\begin{aligned} \mathbf{z} &= \mathbf{x} \times \mathbf{T} \\ &= \begin{bmatrix} x_0 & x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{4.4.1}$$

Assuming modulo-2 arithmetic, the multiplier architecture discussed in [4] is applied to Eq. 4.4.1 in Fig. 4.10. Register $z_{reg} = [z_0, z_1, z_2, z_3]$ contains the answer after multiplication.

Initially $z_{reg} = 0$, $x_{reg} = [x_0, x_1, x_2, x_3]$ and $T_{reg} = [1, 1, 0, 0]$, which is the first row of $\mathbf{T}$. At first, the AND gates multiply x_0 with T_{reg}. This answer

Stellenbosch University http://scholar.sun.ac.za

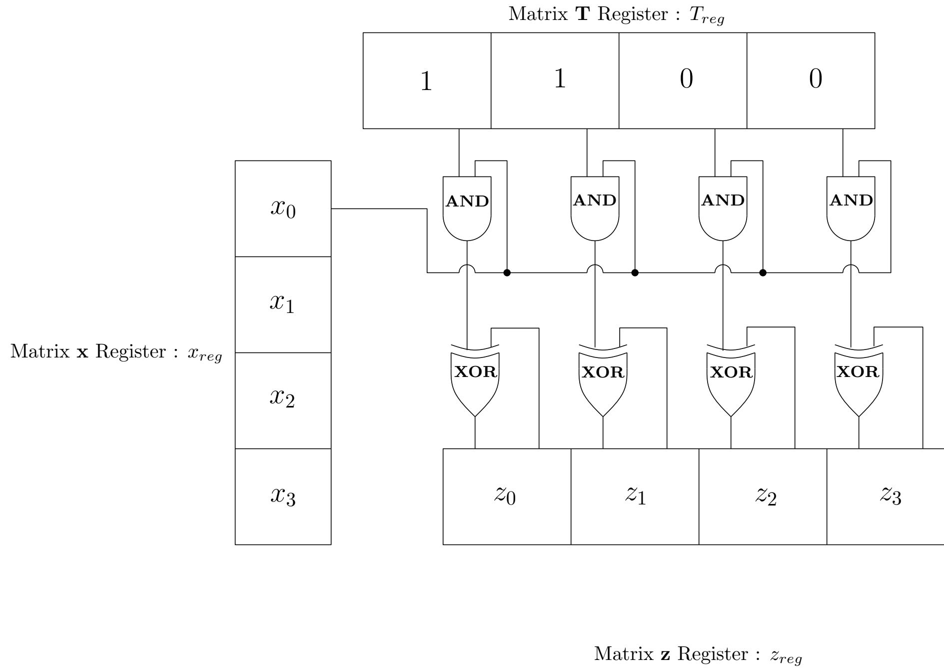


Figure 4.10: Cyclic matrix multiplier architecture from [4].

gets added by the XOR gates to z_{reg}. Now T_{reg} is rotated one bit to the right such that it becomes the second row of $\mathbf{T}$. Register x_{reg} is also rotated one bit position upwards, such that $x_{reg} \leftarrow [x_1, x_2, x_3, x_0]$. Value x_1 is then multiplied with T_{reg} before adding the result to z_{reg}. Continuing this multiply-add procedure for each element in x_{reg}, eventually leads to $z_{reg} = (\mathbf{x} \times \mathbf{T})$.

The multiplier in Fig. 4.10 allows any (4×4) matrix $\mathbf{T}$ to be multiplied by any (1×4) vector $\mathbf{x}$. By removing T_{reg}, the logic usage is reduced. However, this optimisation causes the multiplier to implement a specific $\mathbf{T}$ and cannot be used for any $\mathbf{T}$, as before. An optimised circuit is shown in Fig. 4.11. Note that all AND gates and three XOR gates have been removed.

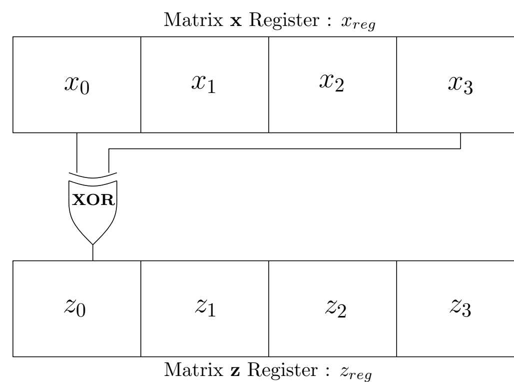


Figure 4.11: A reduced complexity cyclic matrix multiplier architecture.

Stellenbosch University http://scholar.sun.ac.za

The XOR gate's two inputs represent $\mathbf{x}$ being multiplied by the left most (first) column of $\mathbf{T}$. This result is stored in z_0 after which both x_{reg} and z_{reg} are rotated such that $x_{reg} \leftarrow [x_1, x_2, x_3, x_0]$ and $z_{reg} \leftarrow [z_1, z_2, z_3, z_0]$. Now $\mathbf{x}$ is multiplied with the second column in $\mathbf{T}$ where z_1 stores the result. This rotate-add operation is done for each element in z_{reg}.

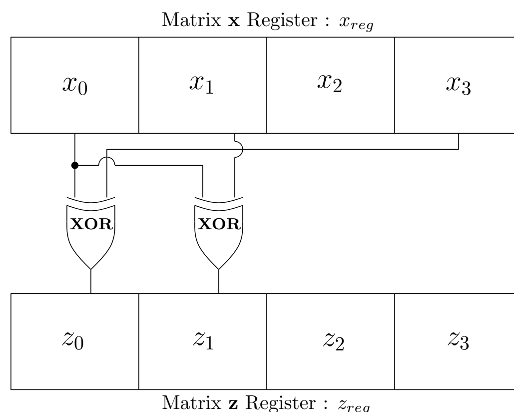


Figure 4.12: A parallelised implemenation of Fig. 4.11.

Fig. 4.12 shows a parallelised implementation of the design in Fig. 4.11. This implementation performs multiplication twice as fast. Instead of multiplying $\mathbf{x}$ with one column in $\mathbf{T}$, it is now multiplied with two subsequent columns in $\mathbf{T}$. Both z_{reg} and x_{reg} are rotated by two bit positions instead of one. Parallelisation can be applied until $z_{reg} = (\mathbf{x} \times \mathbf{T})$ is computed in one clock cycle.

4.4.2 Encoder

4.4.2.1 Frame Structure

A 480-bit TM frame would require a $(n, k) = (480, 960)$ half rate code LDPC. However, all channel coding modules have been designed around the 512 bit block length of BCH. In order to fit this LDPC module into the current hardware configuration, with BCH as FEC, a TM frame is split into two 240-bit sections. These two sections are individually encoded using a (256,512) LDPC code. Splitting a 480-bit TM frame is illustrated in Fig. 4.13. A 16-bit frame ID precedes a 240-bit TM section in each codeword. The decoder expects codeword 0 to arrive first, followed by codeword 1. Should the decoder not receive two successive codewords in this order, it discards all received TM sections.

Stellenbosch University http://scholar.sun.ac.za

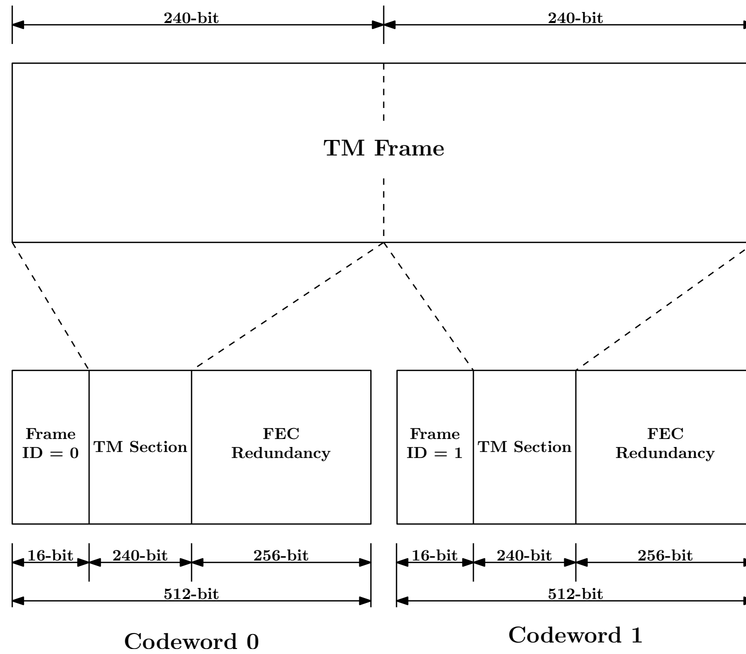


Figure 4.13: A modified TM frame structure for half code rate LDPC.

4.4.2.2 State Diagram

On reset, the *Idle* state is entered with variable $exp_dat = 512$, as shown in Fig. 4.14. Data received in 32-bit sections, from the CRC module, are directly stored in a 256-bit register called x_{reg}. Its first 16-bits represent the frame ID depicted in Fig. 4.13. After receiving the first 256 bits from CRC, variable $exp_dat = 224$. Only 16 bits of the last received 32-bit section are stored in x_{reg}. The other 16-bit section is buffered in a register, tmp_16. Register x_{reg} is now used to create codeword register $c_{reg} = [x_{reg}, p_{reg}]$, where register p_{reg} represents the parity section.

Vector $p_{reg} = [p_1, p_2]$, where p_1 is calculated from Eq. 3.4.11 by using the following multiplication sequence [30] :

1. $a_0 = (A \times x_{reg}^T)$ and $a_1 = (C \times x_{reg}^T)$
2. $a_2 = (T^{-1} \times a_1) = a_1$
3. $a_3 = (E \times a_2)$
4. $a_4 = (\phi^{-1} \times a_3) = a_3$

Finally, $p_1^T = (a_1 \text{ XOR } a_4)$. Note that T^{-1} and ϕ^{-1} are both identity matrices, hence multiplication in steps 2 and 4 are omitted. Next, vector p_2 is calculated by using Eq. 3.4.12 :

1. $b_0 = (A \times x_{reg}^T)$ and $b_1 = (B \times p_1^T)$

Stellenbosch University http://scholar.sun.ac.za

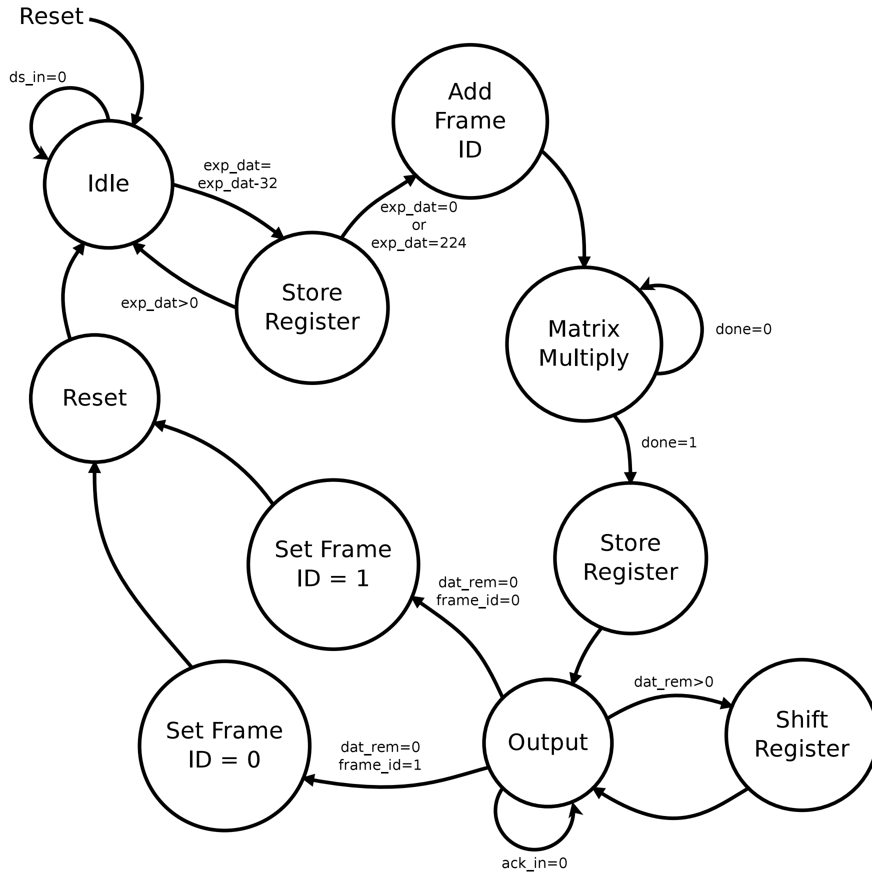


Figure 4.14: State machine diagram of a LDPC encoder.

2. $b_2 = (T^{-1} \times (b_0 \quad XOR \quad b_1)) = (b_0 \quad XOR \quad b_1)$

This leads to $p_2 = (b_0 \text{ XOR } b_1)$. Similar to p_1, multiplication with T^{-1} is omitted when evaluating p_2. The *StoreRegister* state now stores x_{reg} and p_{reg} in c_{reg}, as explained earlier. In vector $c_{reg} = [c_{n-1}, c_{n-2}, \cdots, c_0]$, bit c_{n-1} represents the most significant bit (MSB). The most significant 32-bits of c_{reg} are output during the *Output* state. Upon de-asserting ack_in, register c_{reg} is shifted 32-bits to the left such that $c_{reg} \leftarrow [c_{n-33}, c_{n-34}, \cdots, c_0, \mathbf{0}]$ where the bold $\mathbf{0}$ indicates 32 zeros. Variable dat_rem tracks the remaining bits in c_{reg}. After codeword 0 has been sent, variable $frame_id = 1$ is selected for codeword 1. Similarly, when codeword 1 has been sent, $frame_id = 0$ is selected. The *Reset* state clears all multiplier registers and variables except for $frame_id$ and tmp_16, which are cleared on asynchronous reset. Note that exp_dat is only restored after codeword 1 has been sent. Register x_{reg} is now updated to contain $frame_id$ and tmp_16. The remaining exp_dat bits from CRC are gathered and stored in x_{reg}. Codeword 1 is then created by following the same procedure as with codeword 0.

Stellenbosch University http://scholar.sun.ac.za

4.4.2.3 Hardware Layout

The most important components of an encoder is shown in Fig. 4.15. A FSM module implements the states in Fig. 4.14 and controls data flow between all components. Similar to BCH's decoder in Fig. 4.9, the bold blue lines represent data and control lines between a component and the FSM.

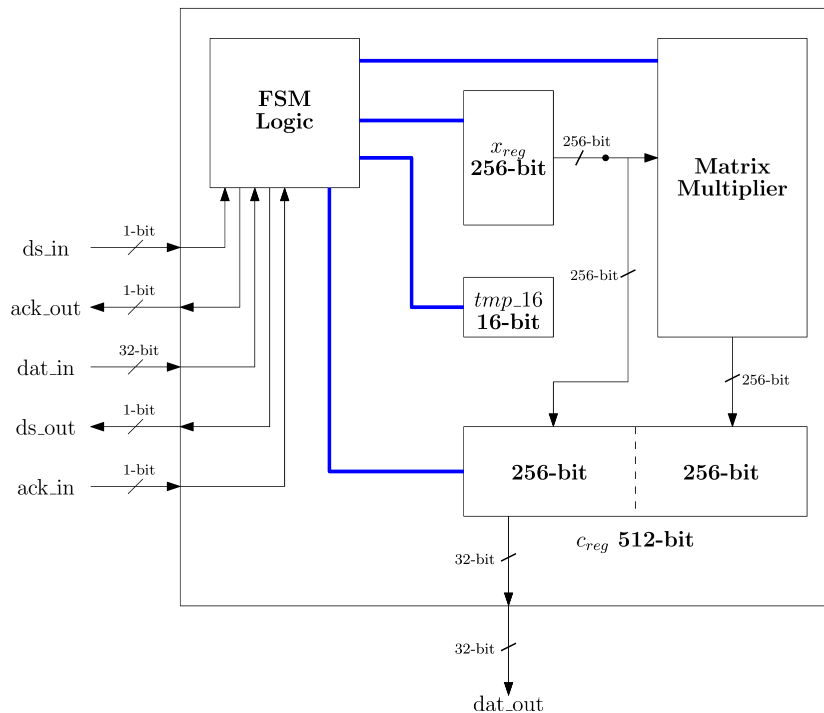


Figure 4.15: LDPC encoder hardware layout.

Data received on *dat_in* are routed by the FSM to the correct position in x_{reg} during the *StoreRegister* state. Register x_{reg} is connected to the matrix multiplier component via a 256-bit bus. The matrix multiplier component contains 4 individual matrix multipliers. In the previous section, only values a_0, a_1, a_3 and b_1 need to be computed since $a_0 = b_0$. Each multiplier implements the logic efficient architecture discussed in Section 4.4.1. The parallelisation technique from that section is also applied. A symbol, F_m, representing the degree of parallelisation, is now introduced. It indicates how much faster the circuit performs with parallelisation than without it. A $F_m = 32$ is chosen such that the multipliers perform at least an order of magnitude faster. Should this encoder, along with the other channel coding components, not fit onto the target FPGA, F_m could always be lowered to reduce logic usage. After multiplication, both x_{reg} and p_{reg} are moved to c_{reg} via 256-bit data busses. Register c_{reg} is then finally output, 32-bits at a time as shown.

Stellenbosch University http://scholar.sun.ac.za

4.4.3 Decoder

During development of the two LDPC decoders, both satellite and ground station platforms had to be prepared for demonstration in Belgium. Development on the Xilinx Virtex 5 and Spartan 3e FPGAs were suspended at that time. Therefore LDPC development continued on the Altera FPGA referred to in Section 3.7.2. The Altera FPGA has much more RAM and logic resources than the Xilinx Virtex 5. A design fitting on the Altera FPGA may therefore not necessarily fit on the Xilinx FPGA. However, by reducing the degree of parallelism in the design, a decoder's logic resources can be lowered to fit on the Xilinx FPGA.

4.4.3.1 State Diagram

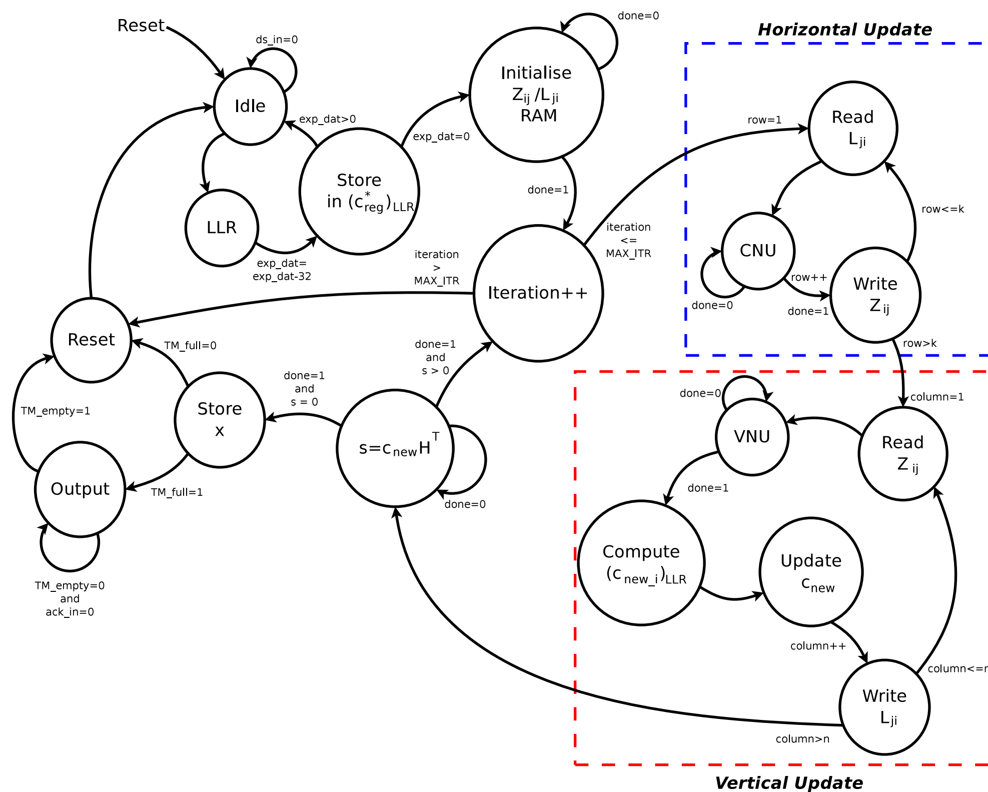


Figure 4.16: State machine diagram of a LDPC decoder.

Upon reset the *Idle* state is entered in Fig. 4.16. Probability values received from the modem have been quantised. Work done in [1] showed that 3-bit quantisation only performs 0.4 dB worse compared to 32-bit floating point values. In order to fit an even amount of probabilities inside a 32-bit data

Stellenbosch University http://scholar.sun.ac.za

bus value, N_q = 4-bit quantisation has been chosen. The total number of data bits expected by the decoder is $exp_dat = (N_q \times n) = (4 \times 512) = 2048$. After receiving a 32-bit word, the LLR state runs each of the $Pr(c_i^* = 0)$ values through a lookup table (LUT), which computes the LLR. These LLRs are then stored inside a register called c_{reg}, before returning to the $Idle$ state. Values stored in c_{reg} represent $(c_i^*)_{LLR}$ from Section 3.4.2. This process continues until $exp_dat = 0$.

A RAM-bank that buffers messages Z_{ij} and L_{ji} between horizontal and vertical updating, is initialised with values from c_{reg}. Messages are arranged in RAM as depicted by the **H** matrix. Accessing a row in **H** is synonymous to accessing a row in RAM. Similarly, a column access in **H** equals a column access in RAM. The rest of this section will refer to row and column access in **H**, for ease of explanation. The blue dashed box in Fig. 4.16 encapsulates the states associated with horizontal updating. The red box encapsulates all the vertical update states.

After incrementing the iteration count, horizontal updating is performed, where **H** is accessed row-wise. At first, $\omega_r = 6$ elements is read from row j in **H**, where $0 \leq j < 256$. A check node update (CNU) is then performed as depicted by Eq. 3.4.7. The result from this CNU is written back to row j. This process continues until all 256 rows have been processed. Vertical step updating is now performed, where **H** is now accessed column-wise. The first state reads $\omega_c = 3$ elements from column i in **H**, where $0 \leq i < 512$. Vertical node updating (VNU) is then performed by using the MS version of Eq. 3.4.4 :

$$Z_{ij} = (c_i^*)_{LLR} + \sum_{i' \in N_{j/i}} L_{ji'} \tag{4.4.2}$$

During a VNU for column i in **H**, the value of $(c_{new_i})_{LLR}$ is also computed with the MS version of Eq. 3.4.6 :

$$(c_{new_i})_{LLR} = (c_i^*)_{LLR} + \sum_{i' \in N_{j/i}} L_{ji'} \tag{4.4.3}$$

Message Z_{ij} are now written back to column i in **H**. Value $(c_{new_i})_{LLR}$ is also converted to a bit-value, c_{new_i}, before stored at position i in register $\mathbf{c}_{new}$. According to Eq. 3.4.2, a positive LLR has $Pr(c_i^* = 0) > Pr(c^* = 1)$, which results in $c_i^* = 0$. Similarly a negative LLRs has $c_i^* = 1$. Using this criteria, value c_{new_i} is determined from $(c_{new_i})_{LLR}$.

After vertical updating, the syndrome is evaluated as $\mathbf{s} = \mathbf{c}_{new}\mathbf{H}^T$. A matrix multiplier from Section 4.4.1, having parallelisation $F_p = 32$, is used. In case $\mathbf{s} \neq 0$, a new iteration is started. A maximum of 20 iterations is performed before declaring a decoding failure. A soft decision decoder requires

Stellenbosch University http://scholar.sun.ac.za

at most 20 iterations to be efficient [1]. Stopping at 100 iterations would only provide about 0.05 dB improvement.

Should $\mathbf{s} = 0$, the TM section from $\mathbf{c}_{new}$ is extracted and passed to x_{reg}. This TM section, along with the current value in x_{reg}, is discarded if the frame ID is not expected by the decoder. After resetting *exp_dat* in the *Reset* state, the decoder waits for the next TM section. Variables $TM_full = 1$ and $TM_empty = 0$ are set when x_{reg} contains a complete TM frame. Register x_{reg} is then output on *dat_out*, in a similar way as with c_{reg} on the LDPC encoder.

4.4.3.2 Hardware Layout

Layout of the decoder is presented in Fig. 4.17. Blue and black lines represent FSM control lines and data busses respectively.

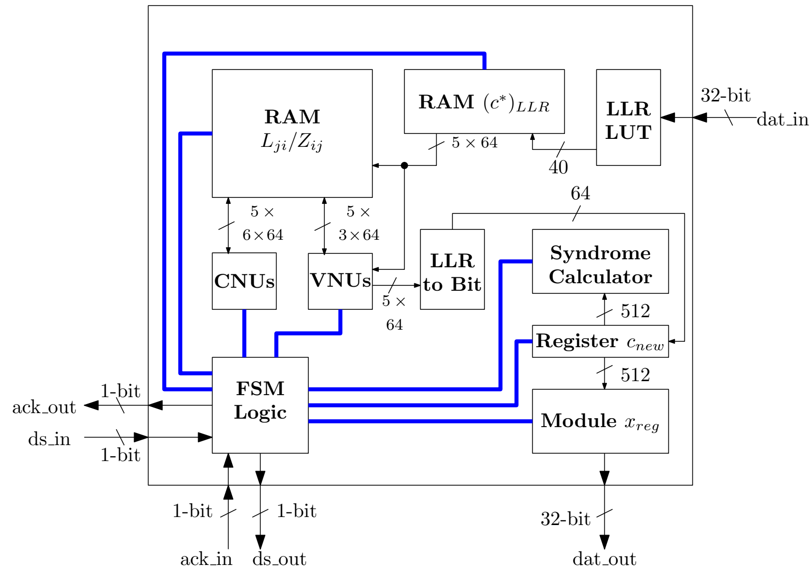


Figure 4.17: General hardware layout of a LDPC decoder.

The LLR LUT component consists of 8 individual LUTs, each having a $N_q = 4$-bit input. Since $N_q = 4$, only $2^4 = 16$ output values need to be stored inside a LUT. Each LUT outputs a $N_{LLR} = 5$-bit value, consisting of both a 4-bit magnitude and 1-bit sign. An example is given in Fig. 4.18. Setting the sign bit to 1 indicates a negative value. These LLRs are not in 2's complement format.

Multiple columns and rows of $\mathbf{H}$ can be processed simultaneously by adding more VNUs and CNUs. This requires the Z_{ij}/L_{ji} RAM-component to simultaneously access many rows or columns. Similar to the encoder, a parallelism factor $F_m = 64$ is chosen to be an order magnitude higher than a non-parallel implementation. This translates to F_m rows or columns in $\mathbf{H}$ being accessed

Stellenbosch University http://scholar.sun.ac.za

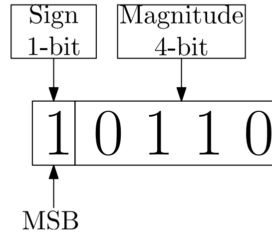


Figure 4.18: A 5-bit LLR value.

simultaneously. Since each row in **H** contains $\omega_r = 6$ LLR values, the CNUs and Z_{ij}/L_{ji} RAM are joined by a $(64 \times 6 \times 5)$-bit bus in Fig. 4.18. Similarly the VNUs and Z_{ij}/L_{ji} RAM are joined by a $(64 \times 3 \times 5)$-bit bus.

During VNU processing, F_m new values for c_{new} are calculated. Value $(c_{new_i})_{LLR}$ is converted to bit-value c_{new_i} by the $LLR-to-Bit$ combination logic circuit. This circuit outputs $c_{new_i} = 0$ when the LLR's sign bit is 0, otherwise it outputs $c_{new_i} = 1$. A 512-bit bus transfers c_{new} to both the syndrome calculator and x_{reg} module. After passing the syndrome check, the x_{reg} module is notified to process c_{new}. Here, the frame ID is parsed after which the module stores the TM frame section. A complete TM frame is output from x_{reg} via a 32-bit bus.

A detailed description of the CNU, VNU and RAM Z_{ij}/L_{ji} modules will now be given.

4.4.3.3 CNU Module

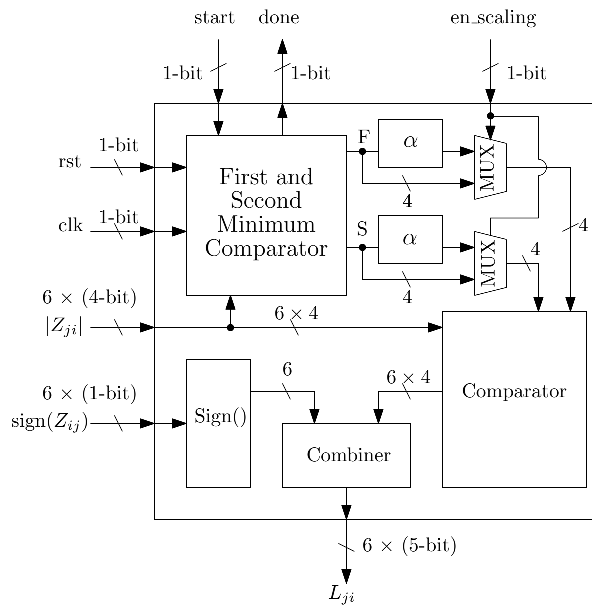


Figure 4.19: A CNU's hardware layout.

Stellenbosch University http://scholar.sun.ac.za

A CNU module is shown in Fig. 4.19. Asynchronous reset, rst, and clock, clk, are both connected to the decoder's matching inputs. This module expects $\omega_r = 6$ Z_{ij} messages from the RAM Z_{ij}/L_{ji} component. Both magnitude and signs of Z_{ij} are input on the $|Z_{ij}|$ and $sign(Z_{ij})$ lines respectively. De-asserting $start$ tells the module to begin processing its input data. By de-asserting $done$, output L_{ji} is assumed to be valid.

Firstly, the comparator finds the first and second minimum magnitudes among the 6 $|Z_{ij}|$ values. These first and second minimum values are output on lines F and S respectively. A clock cycle later, line $done$ is de-asserted . This gives the other combination logic circuits a clock cycle's time to processes F and S. Both F and S are multiplied by scaling term α from Section 3.4.2. The early termination scheme, also mentioned in that section, asserts $en_scaling$ to bypass the multiplier after a certain number of decoding iterations. Both multiplexer (MUX) outputs go to a final comparator where each original input $|Z_{ij}|$ is replaced by the first minimum value. However, should $|Z_{ij}|$ equal the first minimum, it is replaced by the second minimum. Using Eq. 3.4.7, a sign for each input Z_{ij} is computed in the $Sign()$ module. An output sign is computed by XORing the appropriate input signs. Finally, the corresponding signs and LLR magnitudes are joined before output on L_{ji}.

4.4.3.4 VNU Module

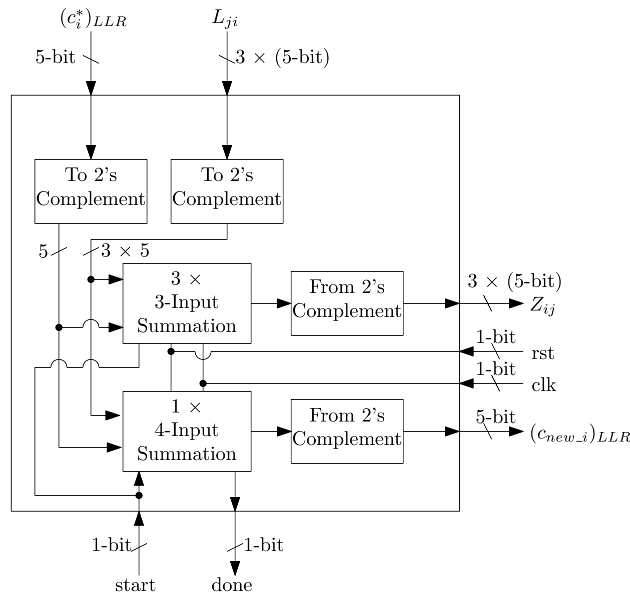


Figure 4.20: A VNU's hardware layout.

A VNU module is shown in Fig. 4.20. During vertical updating, it receives $\omega_c = 3$ L_{ji} messages from column i in RAM Z_{ij}/L_{ji}. This VNU also receives

Stellenbosch University http://scholar.sun.ac.za

$(c_i^*)_{LLR}$ from c_{reg}. Before starting, combination logic converts all VNU module data inputs to 2's complement format.

Upon de-asserting *start*, both adder modules sum their inputs. The 3-input adder module consists of 3 individual 2's complement adders. Each of these adders calculates Z_{ij} according to Eq. 4.4.2. The 4-input adder calculates $(c_{new_i}^*)_{LLR}$ by using Eq. 4.4.3. Since 4 terms are summed by this adder, it takes one clock cycle longer than the 3-input adders to finish. Therefore, only it de-asserts *done* when finished. Combination logic converts the outputs of all adders back to sign and magnitude format, before finally leaving the module.

4.4.3.5 RAM Z_{ij}/L_{ji} Module

Looking at Eq. 3.4.14, it is clear that each '1', of a row's ω_r ones, is in a separate (64×64) matrix. The same can be seen for each column's ω_c elements. Fig. 4.21 shows how this **H** is mapped to an array of (1×256) 9-kbit RAM blocks, on the FPGA.

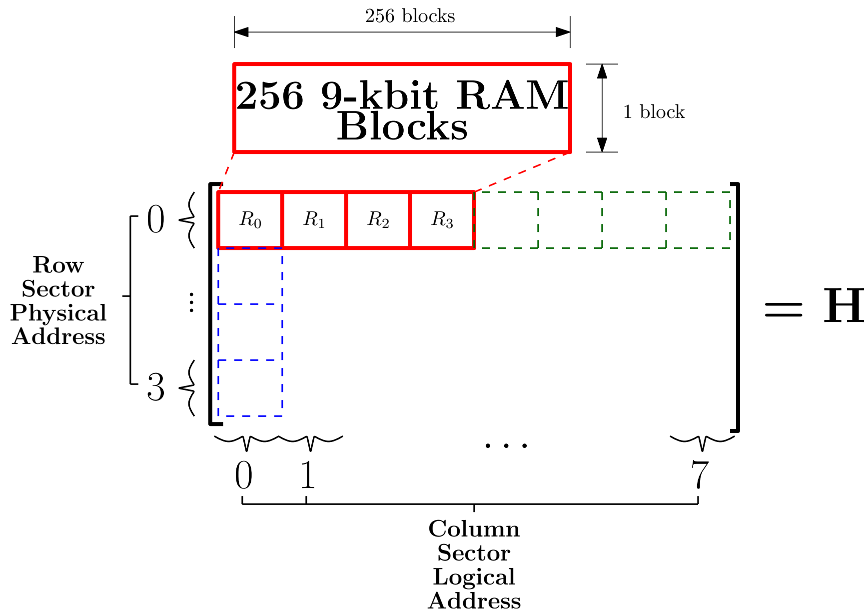


Figure 4.21: RAM block configuration for **H**.

This (1×256) RAM array is divided into four (1×64) segments, R_0 to R_3. Each (64×64) permutation matrix in **H** can be stored in such a (1×64) RAM segment. For example, a (4×4) permutation matrix is reduced to a (1×4) array in Fig. 4.22. This example has replaced the ones with symbols, x_i, for illustration purposes.

A 9-kbit RAM block's address space is partitioned in Fig. 4.23. Bits a_1 to a_0 controls which row sector is accessed in the blue dashed box of Fig. 4.21.

Stellenbosch University http://scholar.sun.ac.za

$$\begin{bmatrix} 0 & 0 & x_0 & 0 \\ 0 & 0 & 0 & x_1 \\ x_2 & 0 & 0 & 0 \\ 0 & x_3 & 0 & 0 \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} x_2 & x_3 & x_0 & x_3 \end{bmatrix}$$

Figure 4.22: A square permutation matrix stored as a row vector.

Bit a_2 allows the RAM blocks to access data in the green dashed box from Fig. 4.21.

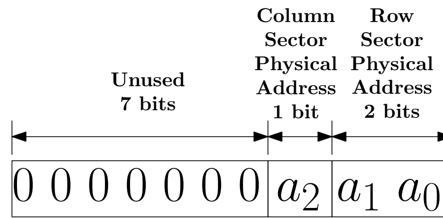


Figure 4.23: Address partitioning of a 9-kbit RAM block.

Note that column sector addresses in Fig. 4.21 are 3-bit logical addresses. These are given to the RAM array's column controller, which uses it to set bit a_2 from Fig. 4.23. Logical address, $[b_2, b_1, b_0]$, is translated to a physical address in Fig. 4.24. Bits b_1 and b_0 are used to select a particular (1×64) RAM segment among R_0 to R_3 in Fig. 4.21.

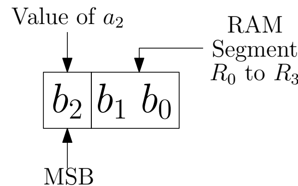


Figure 4.24: Logical to physical address translation.

A single row and column controller is used in this design. During horizontal updating, the row controller receives a row sector address. By knowing the positions of the all zero matrices in Eq. 3.4.14, the controller only reads the correct RAM segments among R_0 to R_3. This controller also knows how each permutation matrix's ones are ordered. Using this information, it groups each of the 64 row's ω_r messages before outputting it to the CNUs. Applying the reverse of this procedure, data from the CNUs are written back to memory. A column controller also knows where all the zero matrices are located. It receives

Stellenbosch University http://scholar.sun.ac.za

a particular logical column address, after which ω_c elements are sequentially accessed for each column in a (1×64) RAM section.

4.5 IPC Implementation

Function	Description
$*ipc_info =$ name_attach($*server_name$)	Creates an IPC server having name $server_name$. Returns ipc_info that points to the created IPC resources.
$*ipc_info =$ name_open($*server_name$)	Connects a client to an existing server of name $server_name$. Returns ipc_info which points to the server's IPC resources.
$success =$ name_close($*ipc_info$)	Disconnects from a server when called by the client. Releases IPC resources when called by a server. Returns the operation's success.
$success =$ msg_send($*ipc_info$, $*msg$, msg_len, $*rep$, rep_len)	Client sends the server a message, msg, of length msg_len bytes. Client blocks until reply, rep, of length rep_len bytes has been received.
$success =$ msg_receive($*ipc_info$, $*msg$, msg_len, $*msg_inf$)	Server waits for message msg. Should $msg = NULL$, the received message length is given by msg_inf. Server then allocates memory for message before calling $msg_read()$.
$success =$ msg_reply($*ipc_info$, $*msg$, msg_len)	Server sends the client a reply message, msg, of length msg_len bytes. Client now unblocked.
$success =$ msg_read($*ipc_info$, $*msg$, msg_len, $offset$)	Allows a server to retrieve a client's message while client is blocked. Retrieved message stored in msg. Variable $offset$ allows a certain segment of client's message to be read.

Table 4.2: Message passing IPC functions for Linux Ubuntu 7.10.

Table 4.2 provides a list of functions present in the Linux message passing scheme. These function names are the same as used in QNX, because TM and ARQ in this chapter will not distinguish between QNX and Linux IPC

Stellenbosch University http://scholar.sun.ac.za

function calls. Note that the input and return types of each function have been omitted. An interested reader is referred to the written C code.

During $msg_receive()$, a client's message is normally copied from the shared memory segment into buffer msg. However, a server doesn't need to allocated memory for msg, before calling $msg_receive()$. This is saves memory on the SH4 and FIT-PC when communication is not active. After receiving the length of a client's message via pointer msg_info, the server calls $msg = malloc()$ to allocate memory. Function $msg_read()$ is then called to read the message from the IPC shared memory segment.

4.6 TM Implementation

A configuration file is parsed by TM during startup in Fig. 4.25. This file contains IPC server and client names which connect TM to ARQ in Fig. 3.15. Both RX and TX threads of TM communicate with the channel coding layer via the two specified device names. Devices in QNX and Linux are listed as file entries under directory $/dev/$. On the SH4 both device names are $/dev/exp$, which is the QNX resource manager written by [35]. Linux's devices are two USB-To-RS232modules named $/dev/ttyUSB0$ and $/dev/ttyUSB1$. The downlink radio is connected to $ttyUSB0$ while the FPGA runs on $ttyUSB1$.

Functions from section A in Fig. 4.25, are contained in separate source files for both OS platforms. File $NEUTRINO_IPC.c$ is compiled for QNX while $LINUX_IPC.c$ is compiled for Ubuntu. Next, a set of callback functions are created. A callback function is a pointer to another function. These pointers reside in TM's source files and point to functions existing in the aforementioned IPC source files. The idea is that TM calls the callback which in turn calls the appropriate IPC routine in the IPC source files. These IPC routines consist of functions $msg_send()$, $msg_receive()$ and $msg_reply()$ from Table 4.2. This configuration allows TM's frame processing routines to be separated from platform dependent IPC. It also allows for easy switching between different OS platforms. Finally, both TX and RX threads of TM are created by using POSIX's $pthread_create()$ function.

4.6.1 Receive Thread

A TM frame is retrieved from channel coding by using callback function $tm_from_lower_layer()$ in Fig. 4.26. Firstly, the frame's master frame count is compared to TM RX's expected frame count. Should these two not be equal, any partial ARQ packet contained in buffer $ARQPacket$ is discarded. A frame having FHP=0x7FE, is used for synchronisation purposes as explained in Section 3.6.1. Frames having any other $FHP > 0$ are automatically dropped until a $FHP = 0$ is found, which contains the start of an ARQ packet. The

Stellenbosch University http://scholar.sun.ac.za

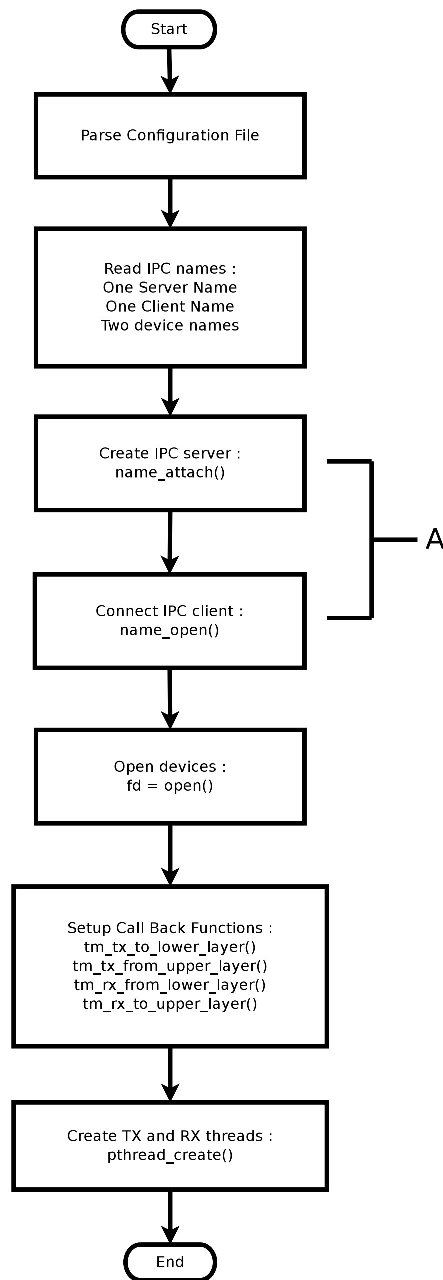


Figure 4.25: Startup of a TM module.

expected frame count is then set to that of the current frame before processing starts.

Firstly, it extracts an ARQ packet's length, as showed in Fig. 3.16, and allocates memory for buffer $ARQPacket$. All remaining data from this frame is then extracted. Should more data be expected, function $tm_from_lower_layer()$ is called. Finally, after receiving all ARQ packet data, $tm_to_upper_layer()$

Stellenbosch University http://scholar.sun.ac.za

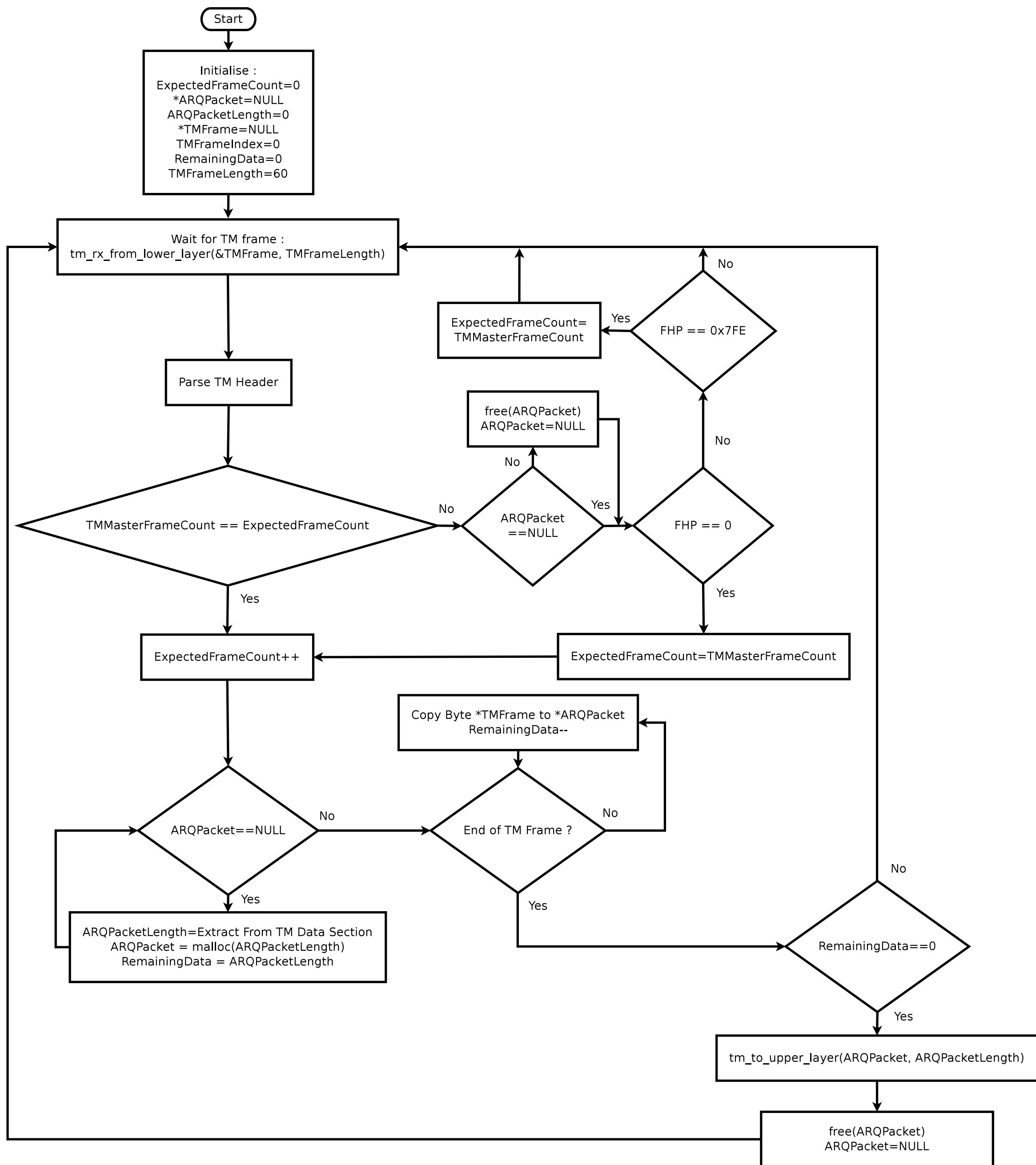


Figure 4.26: The receive thread of a TM module.

is called, which sends the assembled packet to ARQ.

4.6.2 Transmit Thread

By calling function $tm_from_upper_layer()$, the TX thread waits for an ARQ packet. Upon receiving $ARQPacket$ and its length, $ARQPacketLength$,

Stellenbosch University http://scholar.sun.ac.za

memory is allocated for buffer, $TMFrame$, which will store a TM frame. A frame header is then added, followed by the packet's length showed in Fig. 3.16. Next, ARQ data is copied to the frame's data section until it is full. The FHP is then set as explained in Section 3.6.1. By calling $tm_to_lower_layer()$, a frame is sent to the channel coding layer. The $MasterFrameCount$ variable is then incremented, before updating the frame counter in $TMFrame$. Should more ARQ data remain, the data section of $TMFrame$ is filled again. This process repeats until all packet data has been sent.

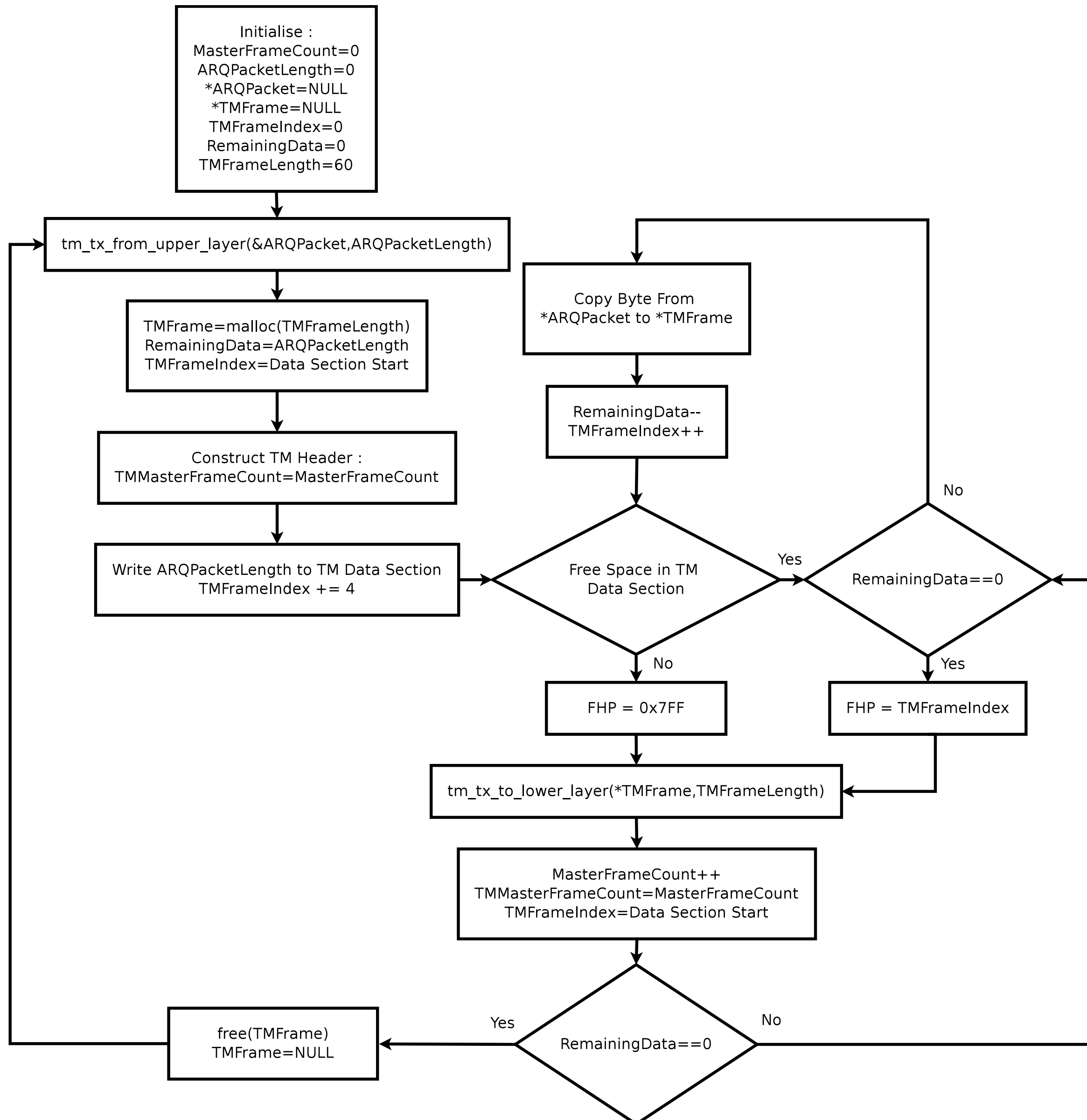


Figure 4.27: The transmit thread of a TM module.

Stellenbosch University http://scholar.sun.ac.za

4.7 ARQ Implementation

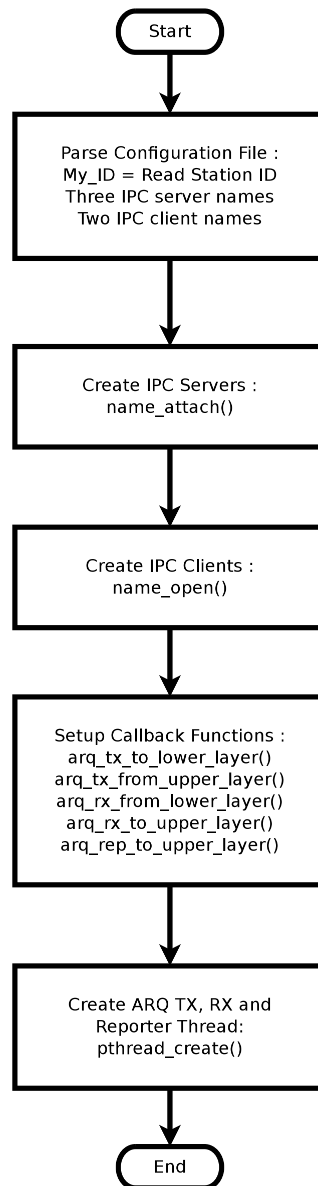


Figure 4.28: Startup of an ARQ module.

Similar to TM, a configuration file is parsed on startup in Fig. 4.28. The station ID from Section 3.6.2.1 is stored as My_ID. Three IPC servers are present in Fig. 3.15 and are listed as follow :

- **ARQ TX Server** : Receives data from both application layer and ARQ's RX thread.

Stellenbosch University http://scholar.sun.ac.za

- **ARQ RX Server** : Receives ARQ packets from TM.
- **ARQ Reporter** : Resides on the RX side of ARQ. Receives a request from the application layer for new files, received by ARQ. The reporter replies as soon as new files have been received.

Next, IPC resources are allocated. Note that separate IPC source files for both QNX and Linux exist, as in the case of TM. Callback functions are then also created to separate IPC functionality from ARQ packet processing routines. Finally, TX, RX and reporter threads are created by using *pthread_create*().

4.7.1 Packet RTT

An ARQ packet's RTT is measured in this section. Using the setup depicted in Fig. 3.20 from Section 3.6.2.2, both ground station and satellite platforms are placed 1 metre apart. Signal propagation delay is negligible at this distance. The packet retransmission feature of ARQ has been disabled for this test. The *RTT* of every tenth ARQ packet that has been sent was measured. Fig. 4.29 shows a section of this measurement.

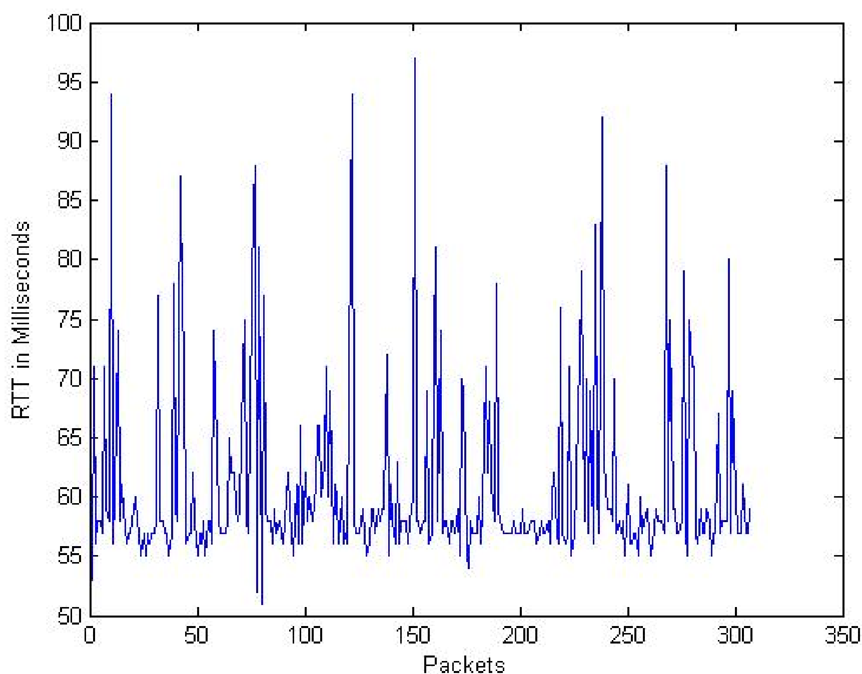


Figure 4.29: Round trip time measurements.

Averaging the graph in Fig. 4.29 leads to $RTT_{avg} = 60.9ms$. Values deviating from RTT_{avg} is suspected to be caused by high data traffic on the SH4.

Stellenbosch University http://scholar.sun.ac.za

Apart from ARQ and TM requesting CPU time, the SAA program on the SH4 performs extensive file I/O and communication over the CAN bus. As drivers have higher scheduling priority over user level processes, TM and ARQ might get unnecessarily delayed. At distance $d = 6$ km in Fig. 3.3 from Section 3.1, the signal propagation delay is calculated as :

$$t_{del} = \frac{d}{c} = 20 \quad \mu s \tag{4.7.1}$$

where $c = 3 \times 10^8$ m/s, is the speed of light [5]. Since t_{del} is an order of magnitude smaller than 1 ms, it is omitted from the final RTT. By using the results from Fig. 4.29, ARQ's time out value is chosen to be 100 ms.

4.7.2 Receive Thread

This thread calls $arq_rx_from_lower_layer()$ to wait for an ARQ packet from TM in Fig. 4.30. The first step in packet parsing compares its ID with My_ID. A mismatch causes the packet to be discarded, since it is not intended for this station. Should this packet be an acknowledge type, its sequence number gets checked first. After passing this check, global variable $ack_received$ is set before using POSIX's $pthread_signal()$ to notify the TX thread of its reception. Other packets intended for file transfer proceeds to the next processing step.

Firstly, its sequence number is compared to the expected sequence number. A mismatch occurs when an acknowledge got lost or when the transmitter rebooted during a file transfer. Should the packet be of type $First$ or $Single$, a new transfer starts and hence the expected sequence number is updated to that of the packet's. In any other case, packets get acknowledged without processing its data contents.

After sequence number verification, the expected sequence number is incremented before acknowledging packet reception. The acknowledge procedure is illustrated in Fig. 4.31. An RX thread of ARQ cannot transmit a packet, therefore an acknowledge request type is sent to TX by calling $arq_tx_to_upper_layer()$. This request type contains all the information required by TX to construct an acknowledge ARQ packet.

Upon receiving a type $First$ or $Single$ packet, a temporary file is created, or cleared if already opened by a previous transfer attempt. The packet's data section is then written to file. This file is closed if the current packet has been of type $Last$ or $Single$. By using $pthread_signal()$, the reporter thread is notified of this new file. Should file transfer not yet be completed, function $arq_rx_from_lower_layer()$ is called to receive the next packet.

Stellenbosch University http://scholar.sun.ac.za

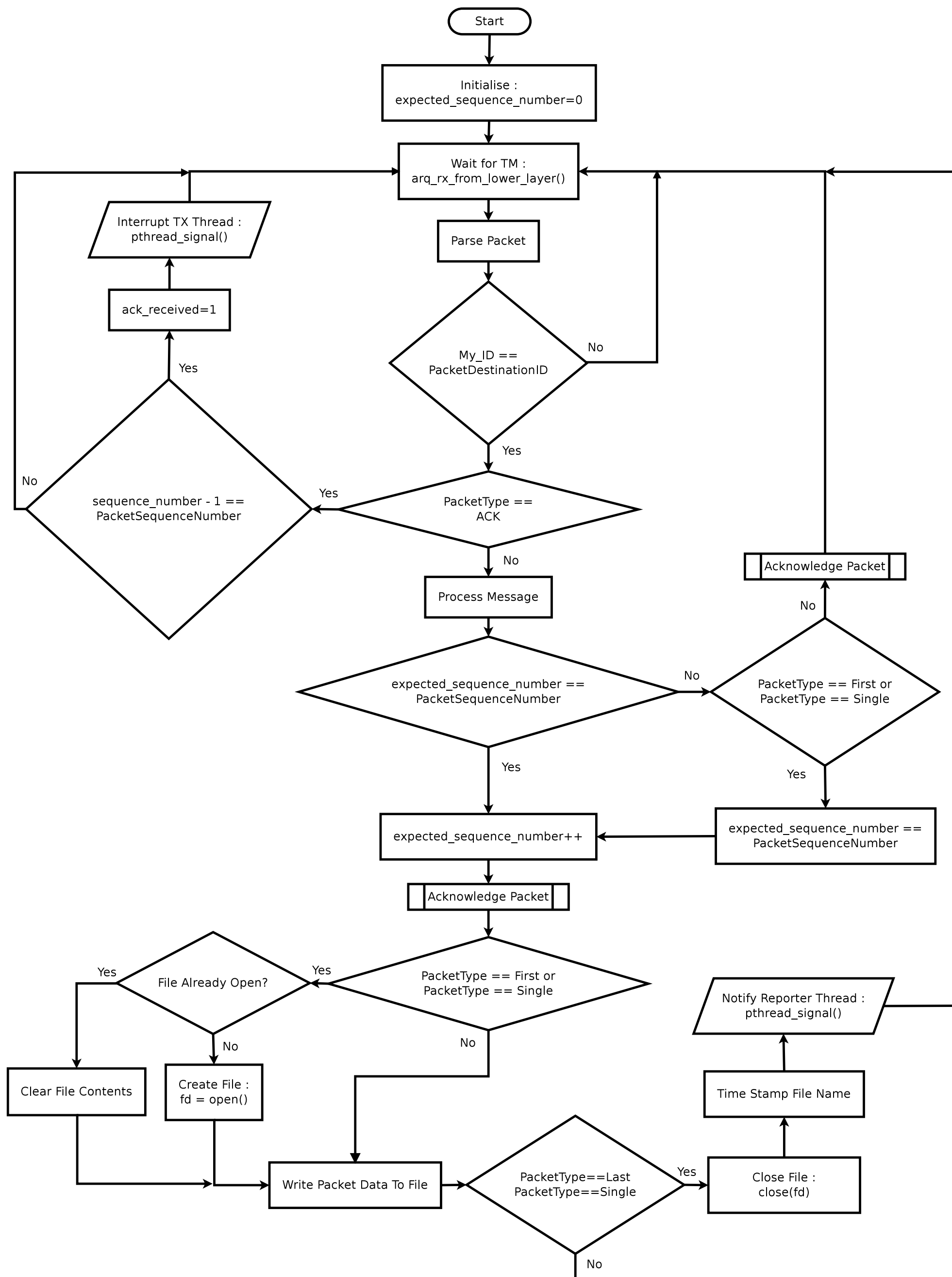


Figure 4.30: The receive thread of an ARQ module.

Stellenbosch University http://scholar.sun.ac.za

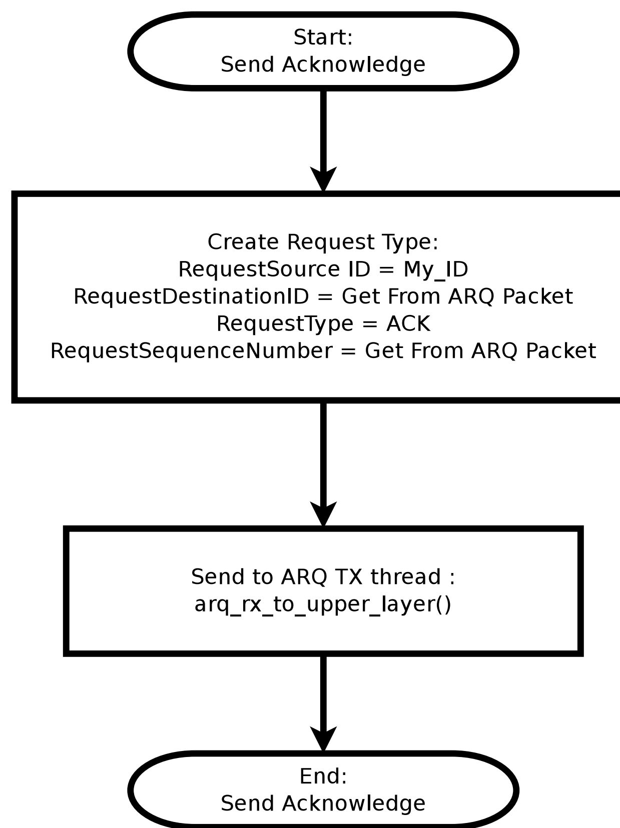


Figure 4.31: The acknowledge procedure from Fig. 4.30.

4.7.3 Transmit Thread

A request type is received when calling $arq_tx_from_upper_layer()$ in Fig. 4.32. This request is received from either ARQ's RX thread or the application layer. Note that IPC doesn't immediately reply when receiving a request via $arq_tx_from_upper_layer()$. By calling this function again after data transmission, it can reply transfer success or failure to the particular client.

A file request contains a file name, file directory and the destination station's ID. Depending on how many bytes have been read from this file, memory is allocated for packet buffer $ARQPacket$. Its header is constructed during the next phase. Here, the sequence number is set according to variable $sequence_number$. Since this variable gets incremented before packet transmission, ARQ RX must compare the acknowledge packet's sequence number to $(sequence_number - 1)$ in Fig. 4.30.

Before transmitting, variable $TransmitRetryCount$ is cleared. Calling $arq_tx_to_lower_layer()$ sends the packet to TM. The TX thread then waits 100 ms before retrying transmission. Should an acknowledge be received before this time runs out, TX is notified via a $pthread_signal()$ call

Stellenbosch University http://scholar.sun.ac.za

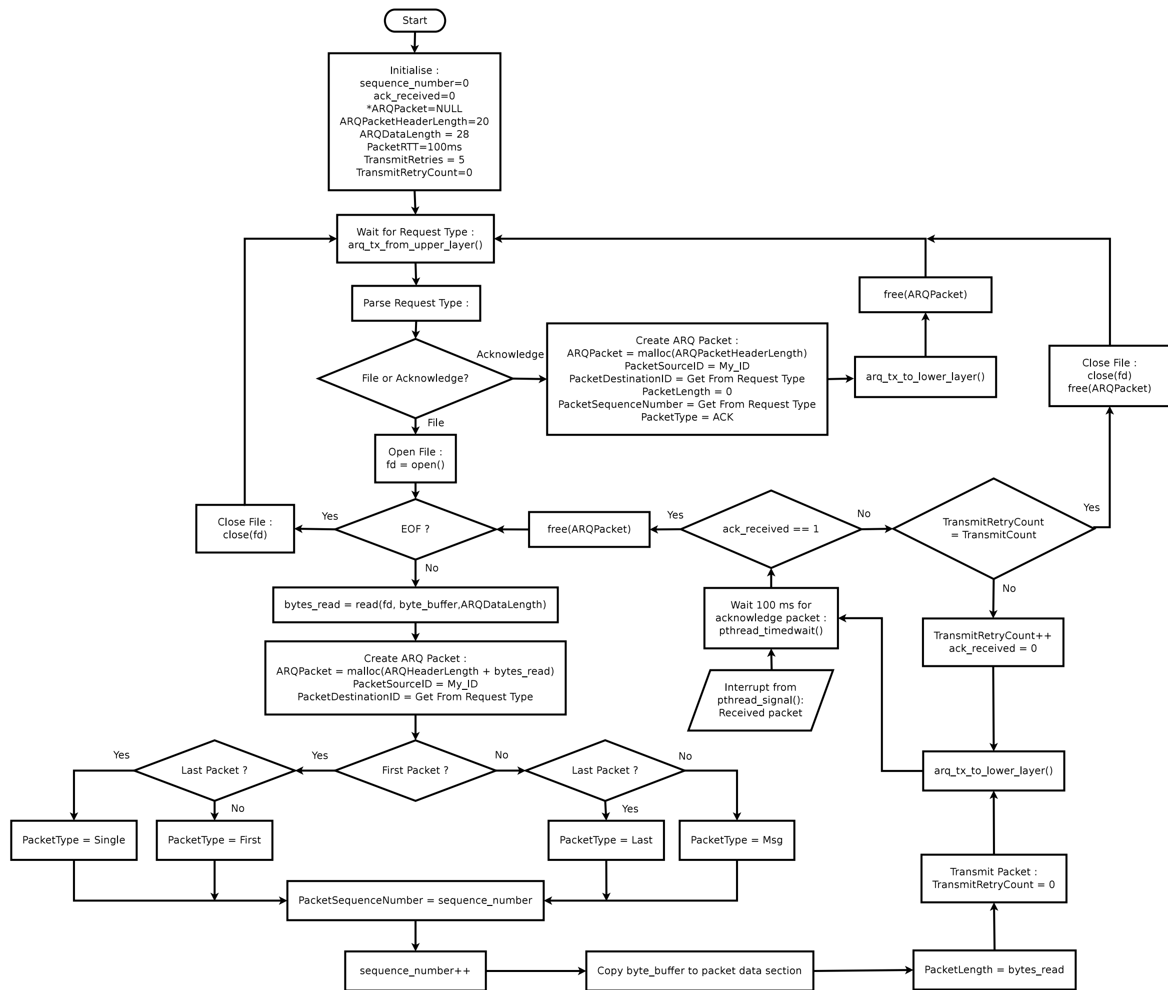


Figure 4.32: The transmit thread of an ARQ module.

from RX. A maximum of 5 packet re-transmissions are allowed before aborting file transmission. After a file has been successfully transferred, function $arq_tx_from_upper_layer()$ is called to notify the application layer of successful transfer.

4.7.4 Stop-And-Wait ARQ Data Throughput

Utilising the test aircraft's flight duration and an ARQ packet's RTT, the amount of file data transferable during a satellite pass can be determined. A worst case scenario where 5 packets have to be retransmitted is assumed. The time to transmit a single packet is therefore :

Stellenbosch University http://scholar.sun.ac.za

$$t_{pck} = (4 \times t_{ARQ_Time_Out}) + RTT_{avg} = (4 \times 0.1) + 0.0609 = 460ms \quad (4.7.2)$$

Using Eq. 4.7.2 and expecting the aircraft pass to last 15 minutes, the total number of transmitted ARQ packets are :

$$N_{Pck} = \frac{t_{flight}}{t_{pck}} = \frac{15 \times 60}{0.46} = 1956 \quad packets \quad (4.7.3)$$

Since 28 bytes of a packet are file data, the total number bytes transferable during a pass are :

$$N_{Bytes} = N_{ARQ_Pck} \times 28 = 54.78 \quad kB \quad (4.7.4)$$

Section 3.6.2 mentioned the expected file sizes to be 10 kB. Having $N_{Bytes} >$ 10 kB, the data throughput of stop-and-wait ARQ is more than adequate for this project.

4.8 Summary

It has been shown how channel coding's modules will interact with existing FPGA firmware's interfaces for both satellite and ground station platforms. These modules implement an interface that allows new modules to be easily added or existing ones to be easily removed. This is particularly useful when switching between BCH and LDPC implementations. Implementation of all channel coding modules have also been described by state machine - and hardware layout diagrams. Parallelisation techniques to reduce the data processing latency on each module, has also been provided.

A message passing API for IPC has also been presented. Various functions have been implemented in C to hide unnecessary IPC details from the user. Finally, detailed packet and frame processing routines for both ARQ and TM have been provided by flow diagrams. Data throughput for the stop-and-wait ARQ protocol have been shown to be sufficient for this project.

The next chapter evaluates these implementation results for both satellite and ground station platforms. Results from the project's demonstration at KUL are also highlighted.

Stellenbosch University http://scholar.sun.ac.za

Chapter 5

Testing, Results and Discussion

In this chapter, the results following the implementation of the different system components, as discussed in Chapter 4, are presented. Unit test procedures carried out for each developed component, are also set out. The channel coding subsection reveals the FPGA logic usage and data throughput in bps for each of its modules. Forward error correction BLER and BER plots are then presented for both BCH and LDPC. Finally, TM and ARQ's test procedures and performance results are provided.

5.1 Channel Coding

5.1.1 Testing

Unit testing for each VHDL module has been performed by writing a VHDL test bench. This test bench simulates a module's operation according to the given inputs on its interface as per Fig. 4.3. The module's output is then compared to a known and correct result. For example, the CRC polynomial division has been performed in Matlab by using $r = deconv(m, g)$, where vectors r,m and g contain binary coefficients from the example in Section 3.2.1. Inputting m to the VHDL module, the same output r is expected as in Matlab.

Integration testing comprised the loop back test as per Fig. 5.1. This test checks whether all modules communicate correctly with each other. It also tests whether each module resets its registers and variables correctly after processing a complete TM frame. The *Channel Coding Encode* module contains all channel coding modules from the ground station of Fig. 3.5. Similarly, the satellite platform's modules are encapsulated by the *Channel Coding Decode* module. A test application on the SH4 generates and sends TM frames from A, denoted TM_A. These are encoded and decoded on the FPGA before receiving them at B, denoted TM_B. After the application verified $TM_A = TM_B$, a new frame is generated and sent. This test was repeated for an hour and regarded

Stellenbosch University http://scholar.sun.ac.za

as successful if no failure occurred. This was indeed the case.

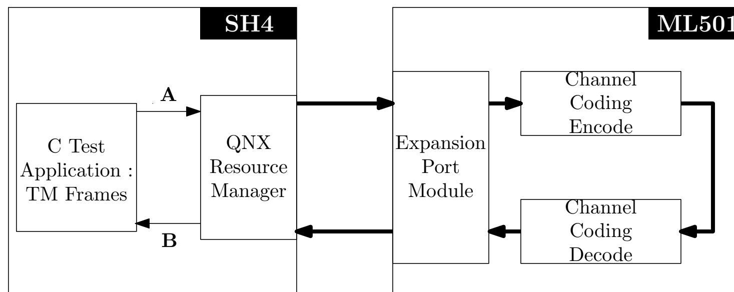


Figure 5.1: Hardware loopback test in FPGA for channel coding modules.

5.1.2 Results

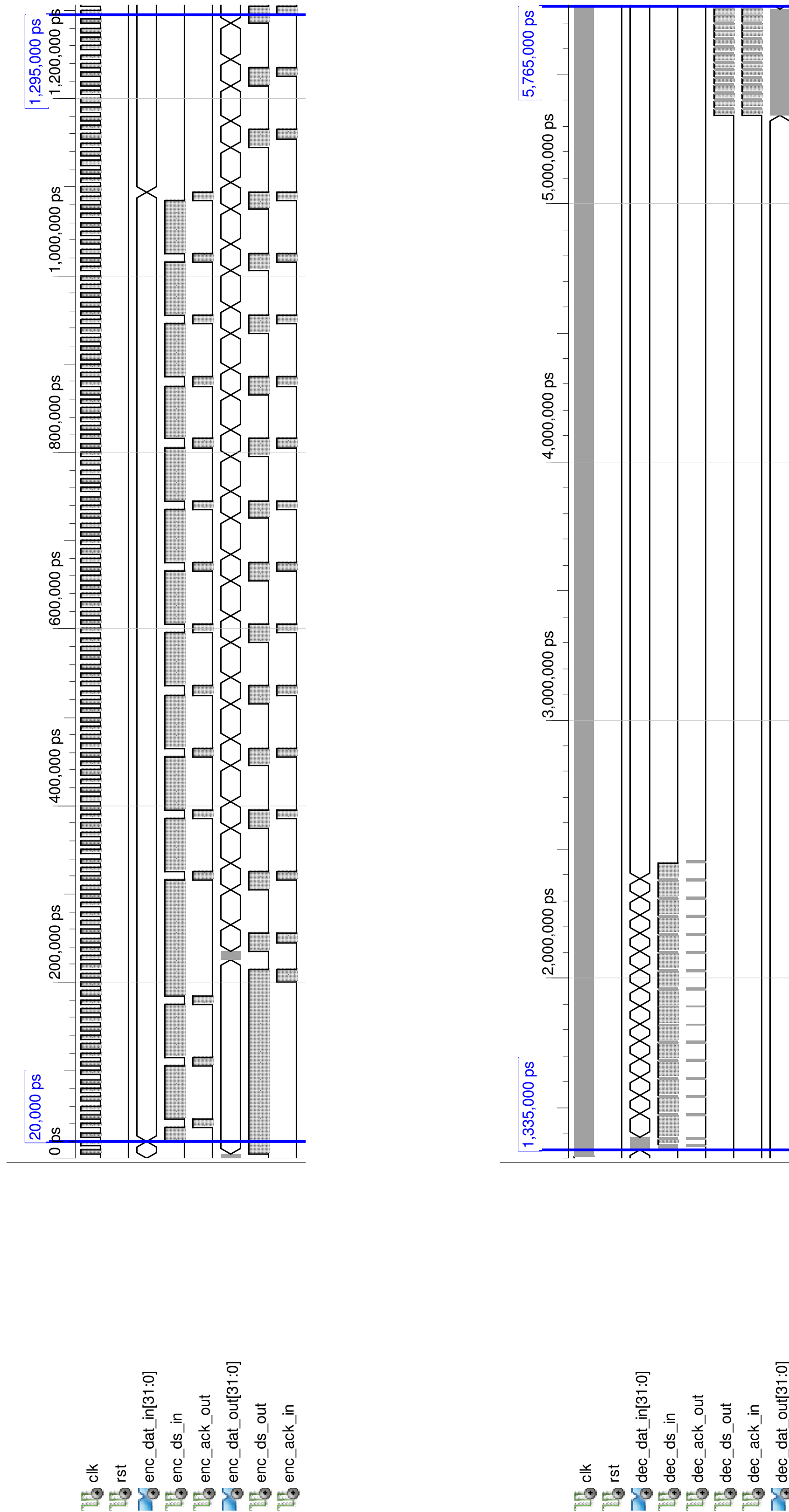
Table 5.1 shows the implementation results of both encoder and decoder components from Fig. 5.1. Logic usage on both FPGAs are expressed as a percentage of the available resources.

Component	FPGA	Logic Elements Used	Processing Delay at 100 MHz Clock
Channel Coding Encode	Xilinx Spartan3E XC3S500E	390 of 4656 Flip Flops; 818 of 9312 LUTs	1.275 μs
Channel Coding Decode	Xilinx Virtex5 XC5VLX50	2470 of 28800 Flip Flops; 3664 of 28800 LUTs	4.43 μs

Table 5.1: Channel coding FPGA module implementation details.

The processing delay refers to the time taken to successfully encode or decode a TM frame. Fig. 5.2a shows a timing diagram for the channel coding encode component. This diagram has been obtained from a VHDL test bench simulation in Xilinx's ISim simulator. Blue markers indicate the start and finish times for encoding. The first 32-bit section of a TM frame enters the module at 20 ns. The last 32-bit section leaves at 1.295 μs. This leads to an encoding delay of 1.275 μs. Similarly, the decoding delay has been determined from Fig. 5.2. Note that three random bits have been corrupted in this codeword.

Stellenbosch University http://scholar.sun.ac.za



(a) Encoding delay measurement.

(b) Decoding delay measurement.

Figure 5.2: Measurement of channel coding's processing delay.

Stellenbosch University http://scholar.sun.ac.za

5.1.3 Discussion

Logic usage on both FPGAs are quite low when implementing either the encoding or decoding chain of modules. This leaves plenty of space for other FPGA modules to be added during final systems integration. More than enough space will also be available when upgrading to half code rate BCH or LDPC in the future. The time to process a TM frame is also very low. Decoding delay is short, even when the maximum amount of 3 random bits have to be corrected by the (511,484) BCH. Finally, both encoder and decoder components add a delay that is more than an order of magnitude smaller than the measured RTT_{avg} of 60 ms in Section 4.7.1. Therefore, the channel coding process is very fast compared to the rest of the communications channel.

5.2 BCH

Results of the $(511, 484)$ code and a $(511, 259)$ code are discussed in this section. The latter code of rate $R = (259/511) \approx 0.5$ has been implemented to compare BCH's BLER performance with $R = 0.5$ LDPC.

5.2.1 Testing

Unit and integration testing for the $(511, 484)$ code has been part of testing in Section 5.1.1. Sample codewords have been simulated in a VHDL test bench. These codewords contained three random bit errors for the decoder to correct.

5.2.2 Results

Component	FPGA	Logic Elements Used	Processing Delay at 100 MHz Clock
(511,484) Encoder	Xilinx Spartan3E XC3S500E	102 of 4656 Flip Flops; 322 of 9312 LUTs	1.11 μs
(511,259) Encoder	Altera Cyclone III EP3C120F780C7N	1680 of 120000 LEs	2.515 μs
(511,484) Decoder	Xilinx Virtex5 XC5VLX50	1595 of 28800 Flip Flops; 2334 of 28800 LUTs	3.83 μs
(511,259) Decoder	Altera Cyclone III EP3C120F780C7N	9960 of 120000 LEs	45.47 μs

Table 5.2: BCH FPGA module implementation details.

Stellenbosch University http://scholar.sun.ac.za

Table 5.2 shows FPGA implementation details for both encoder and decoder modules. Altera's synthesis report expresses FPGA logic usage i.t.o logic elements (LEs) and not as flip flops or LUTs. Similar to channel coding, the processing delay measures how long it takes to encode or decode a complete TM frame. Here, the (511,484) decoder had to correct 3 bit errors. Since the (511,259) code is half rate, it implements the TM frame splitting technique discussed in Section 4.4.2.1. The (511,259) code corrected a maximum of 30 errors per sub-frame.

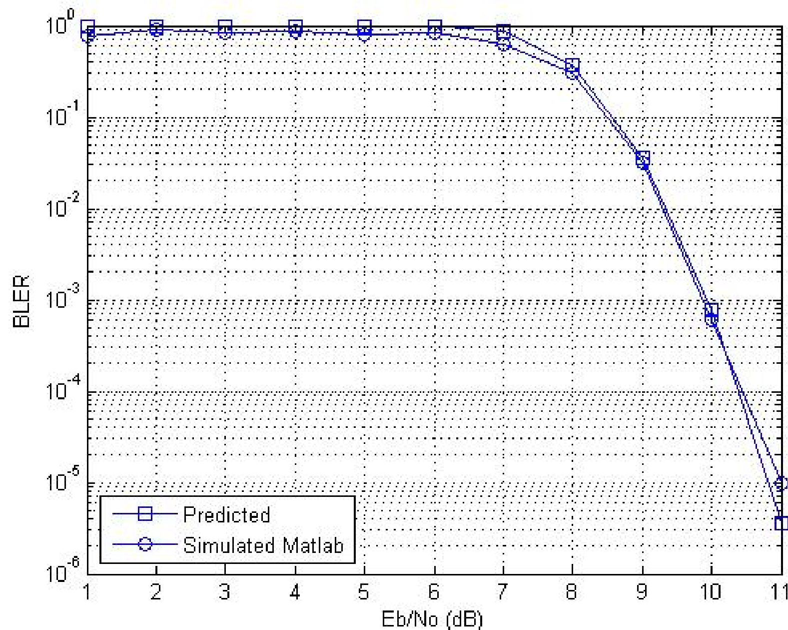


Figure 5.3: BLER plot of a (511,484) BCH code.

Fig. 5.3 shows the BLER performance of a $(511, 484)$ code. The *Predicted* values have been obtained by using Eq. 3.1.6 in Section 3.1. Note that *Predicted* matches with the BLER simulation in Matlab except at $E_b/N_o = 11$ dB.

The $(511, 259)$ code's BLER is shown in Fig. 5.4. By using Eq. 3.1.6, the *Predicted* values have been determined. Both FPGA and Matlab simulation results match each other for this code. Note that the Matlab results differ from the others by exactly an order of magnitude at $E_b/N_o = 6$ dB.

5.2.3 Discussion

Looking at the logic usage for (511,484) BCH, it is clear that it occupies the most logic from channel coding's usage in Table 5.1. At first, the logic usage of (511,259) may seem low. Since the Altera FPGA contains 120000 LEs, the encoder would use almost an equivalent of 36% resources on the Spartan3E

Stellenbosch University http://scholar.sun.ac.za

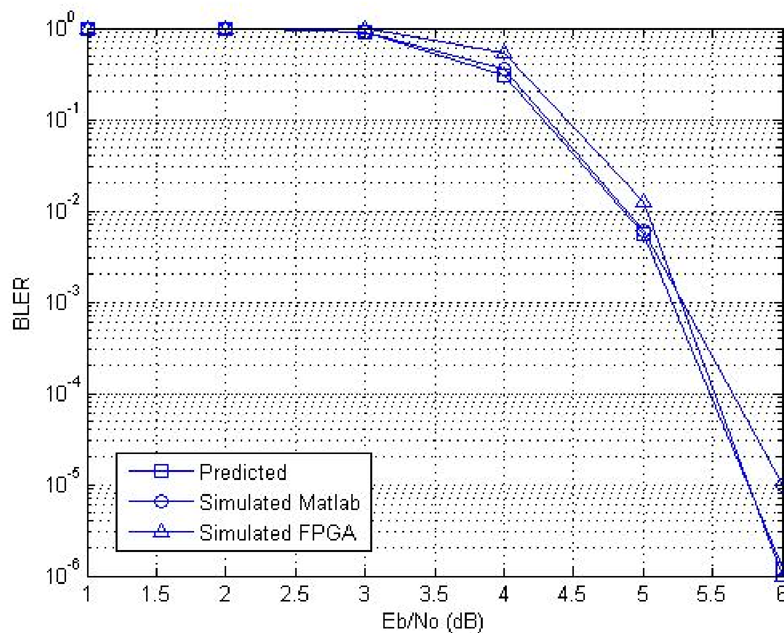


Figure 5.4: BLER plot of a (511,259) BCH code.

FPGA. Similarly, the decoder would take up 34% resources on the Virtex 5 FPGA. Even though this code requires more logic, it would still fit on both Xilinx platforms.

The block error probability predictions for the (511,484) BCH coincides with Matlab's BLER simulation in Fig. 5.3. Given the time constraints of this project, there was not enough time to simulate this code on the FPGA as well. However, this module did successfully pass the VHDL test bench simulations, as well as systems integration testing from Section 5.1.1. Furthermore, given that both Matlab and FPGA platforms implement the same (511,484) decoder, and the success of simulation in Matlab in Fig. 5.3, it can be deduced that the FPGA decoder is functioning correctly.

Clearly, the (511,259) module functions correctly since all three sets of results coincide with each other, as can be seen in Fig. 5.4. Given that this decoder implements the same syndrome computation, BM and Chien algorithms used by the (511,484) code, it serves as further proof that the (511,484) code is functioning correctly. An order of magnitude difference in the BLER at $E_b/N_o = 6$ dB, is due to not enough codewords being simulated in Matlab. In order to obtain a BLER of 10^{-6}, it is required to simulate at least a million codewords. Only 100 000 codewords have been simulated, as a million would have taken a few days to run. This resulted in a BLER of only 10^{-5} at $E_b/N_o = 6$ dB for the Matlab simulation. Since the hardware processing delay is short, the FPGA easily simulated a million codewords within 3 hours.

Stellenbosch University http://scholar.sun.ac.za

5.3 LDPC

5.3.1 Testing

Similar to the (511,259) BCH modules, LDPC has not been part of the loop back testing in Fig. 5.1. During development of the greater project, there has not been enough time to elsewhere implement bit confidence measurement on the QPSK modem. Only a BLER performance comparison between the FPGA and Matlab implementations, has been done.

5.3.2 Results

Implementation details for the FPGA modules are given in Table 5.3. This decoder assembled a TM frame from two subsequent codewords as discussed in section 4.4.2.1. After processing the first codeword, the next one has been input immediately. A maximum of 20 iterations per codeword have been performed while measuring the time to process a complete TM frame.

Component	FPGA	Logic Elements Used	Processing Delay at 100 MHz Clock
Encoder	Altera Cyclone III EP3C120F780C7N	3000 of 120000 LEs	2.63 μs
Decoder	Altera Cyclone III EP3C120F780C7N	38880 of 120000 LEs	151.91 μs

Table 5.3: LDPC FPGA module implementation details.

The α scaling value from Eq. 3.4.8 in Section 3.4.2 has been determined by Matlab simulation in Fig. 5.5. Note that a bit error rate (BER) instead of a block error rate (BLER) is used. Fig. 5.6 compares the optimal value of $\alpha = 0.9$ against no scaling, or $\alpha = 1$.

Scaling by α is terminated before reaching 20 decoding iterations, as discussed in Section 3.4.2. The iteration count at which to terminate scaling, denoted early termination (ET), has been simulated Matlab. Fig. 5.7 shows these results when using $\alpha = 0.9$. Clearly $ET = 15$ delivers the best BER performance. Fig. 5.8 compares this result to the absence of an ET scheme. Note the improvement of about 0.5 dB at BER=10^{-7}.

Using an $ET = 15$ and $\alpha = 0.9$, Fig. 5.9 compares the Matlab and FPGA results with each other. Both graphs follow each other except at $E_b/N_o = 6$ dB. Finally, Fig. 5.10 compares the BER performance of both (511,259) BCH and (512,256) LDPC implementations from Matlab.

Stellenbosch University http://scholar.sun.ac.za

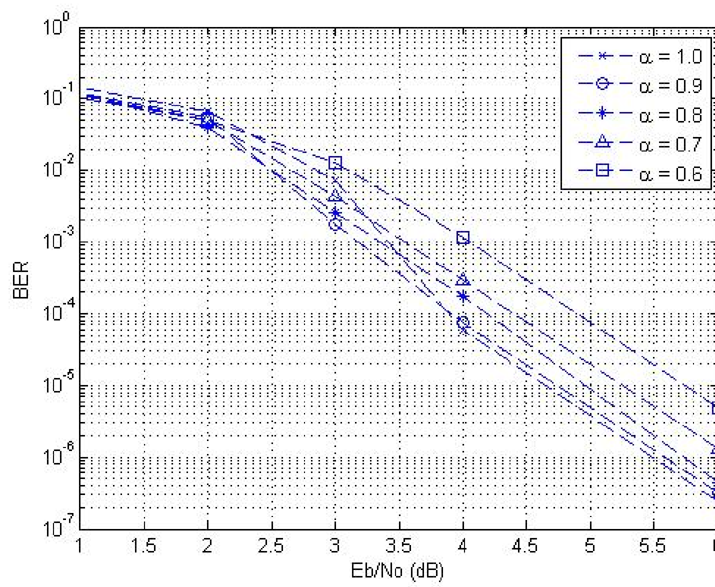


Figure 5.5: Optimal α search for the $(512, 256)$ code.

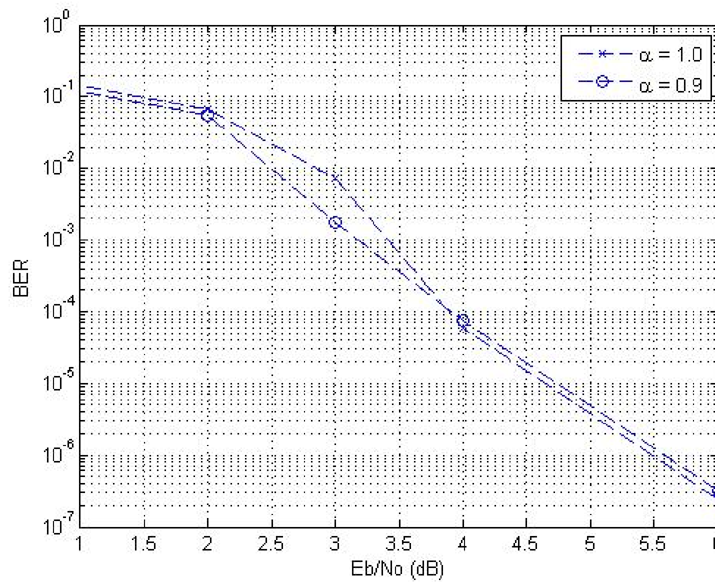


Figure 5.6: Optimal $\alpha = 0.9$ compared against $\alpha = 1$.

Stellenbosch University http://scholar.sun.ac.za

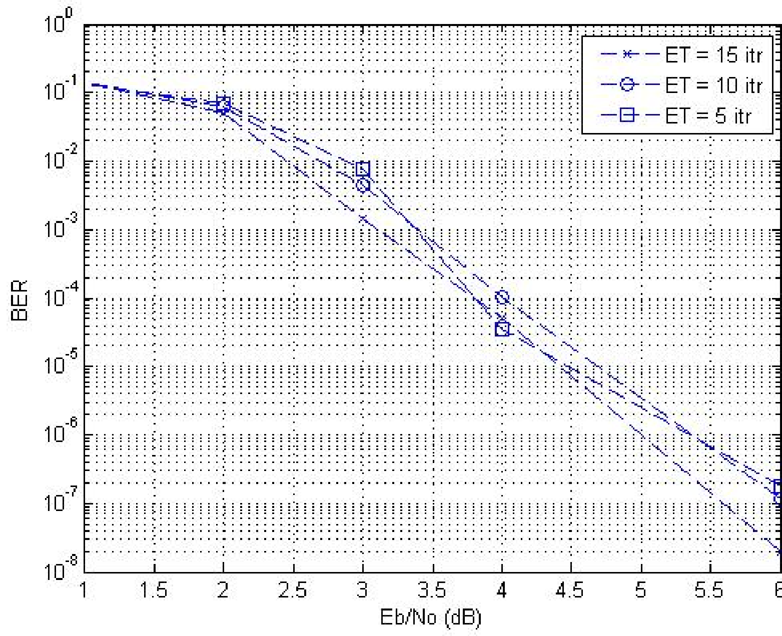


Figure 5.7: Termination of $\alpha = 0.9$ scaling at different iteration counts.

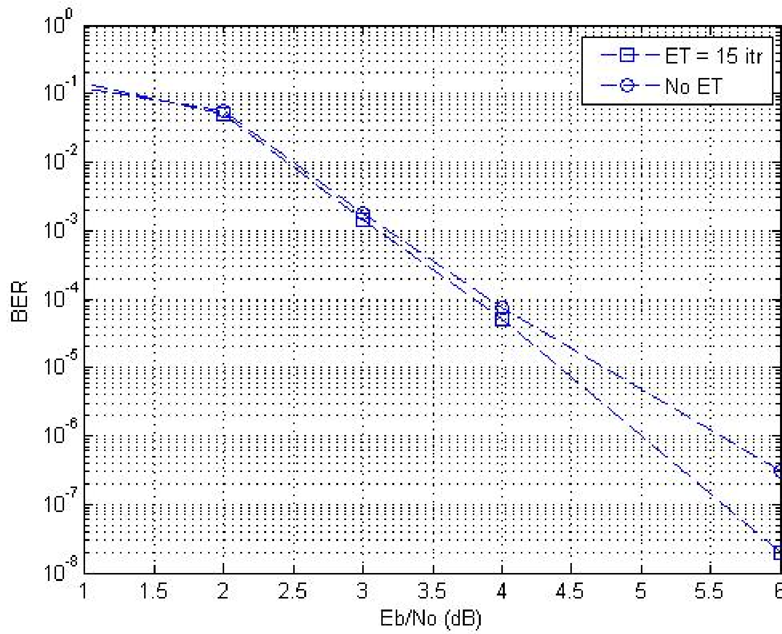


Figure 5.8: Comparison between $ET = 15$ iterations and no ET when using $\alpha = 0.9$.

Stellenbosch University http://scholar.sun.ac.za

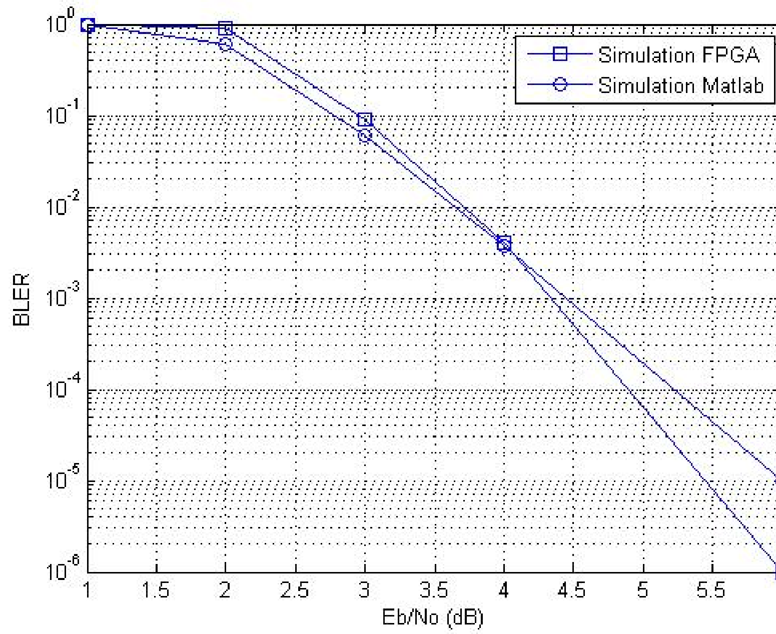


Figure 5.9: Comparison between FPGA and Matlab simulations for $\alpha = 0.9$ and $ET = 15$ iterations.

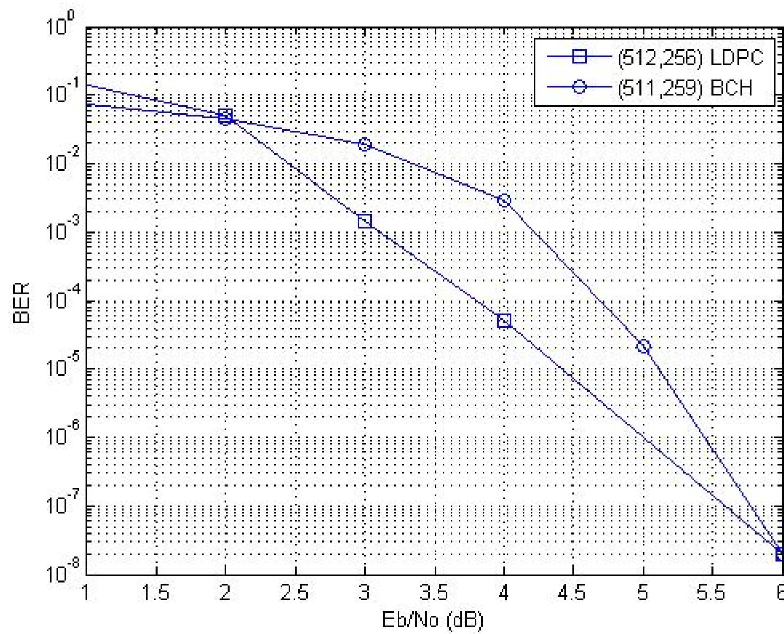


Figure 5.10: Bit error rate comparison between half rate BCH and LDPC implementations from Matlab. LDPC uses $ET = 15$ and $\alpha = 0.9$.

Stellenbosch University http://scholar.sun.ac.za

5.3.3 Discussion

Resource usage on the FPGA for the LDPC decoder is very high compared to BCH. This is due to the parallelisation introduced in Section 4.4.3.2. Reducing parallelisation by a factor of 4 would more than half its current logic usage, therefore allowing implementation on the Virtex 5 FPGA. However, this would increase processing delay almost a factor of four. Still, the total delay will be less than a millisecond and therefore an order of magnitude less than the current RTT of an ARQ packet.

Scaling by $\alpha = 0.9$ delivered an improvement at $E_b/N_o = 3$ dB in Fig. 5.6. However, it also degraded performance by almost 0.1 dB between 4 and 6 dB SNR. By using the ET technique in [29], this degrading in performance has been countered. A clear improvement ranging from 0.1 to 0.7 dB between 4 and 6 dB SNR can be noticed. This improvement only requires adding a simple multiplexer in the check node processor unit, which can enable or disable scaling by α.

The exact order of magnitude difference between Matlab and FPGA BLER simulations is again attributed to the duration of simulation. In order to complete simulations within reasonable time, only 100 000 codewords have been processed in Matlab. A million codewords have been processed on the FPGA within 3 hours. Finally, it is clear from Fig. 5.10 that LDPC outperforms BCH. Bit error rate performance, between 3 and 5 dB SNR, is at least an order of magnitude better compared to BCH. A BER$=10^{-6}$ is considered as error free [17], therefore giving LDPC a 0.5 dB lead above BCH.

5.4 TM and ARQ Protocols

Unit testing took place on both the FIT-PC and SH4 platforms, since both TM and ARQ use different IPC modules on these platforms. Fig. 5.11 shows the loop back test setup on both platforms. All instances of ARQ and TM run on the same computer and connect to each other via IPC. A test application, which mimes the application layer from Fig. 3.15, then connects to ARQ at A. Files now get transferred from ARQ at A to ARQ at D. This test ensures that frame and packet processing routines from TM and ARQ are handled according to the flow diagrams from chapter 4. Files ranging from 10 Mb to 400 Mb have been transferred to check for stability issues.

Stellenbosch University http://scholar.sun.ac.za

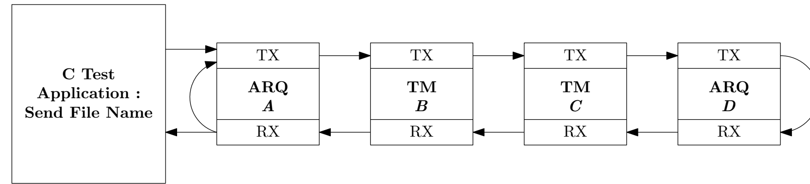


Figure 5.11: TM and ARQ testing procedure.

After both ground station and satellite platforms have been implemented, file transfer between them were interrupted at random times. An interruption has been caused by either cutting the power to one of the platforms or by terminating TM and ARQ processes. After resetting the interrupt condition, the platform which hasn't been interrupted is expected to be fully functional. For example, a receiver is expected to clear an incomplete file and have its expected sequence number adapted on the next file transfer. Files being successfully transferred are compared against its source for data corruption, by using file comparison applications such as *Meld* in Linux. Testing continued until both TM and ARQ's program flow coincided with its flow diagrams as depicted in Sections 4.6 and 4.7. After finishing, there were absolutely no remaining concerns regarding file transfer integrity and stability.

5.5 Belgium Demonstration

An aircraft borne test has been originally scheduled for August 2011. However, due to project scope changes, this test has been cancelled. A new test was then scheduled for October 2011 in Leuven, Belgium. The SAA was previously tested for functionality and compatibility with the platform, but the entire integrated system still has to be field tested and proven. This would include the SAA, satellite platform, communication subsystems and ground station with all constituent hardware and software. Fig. 5.12 illustrates the locality for the new test procedure which took place on ground.

Stellenbosch University http://scholar.sun.ac.za

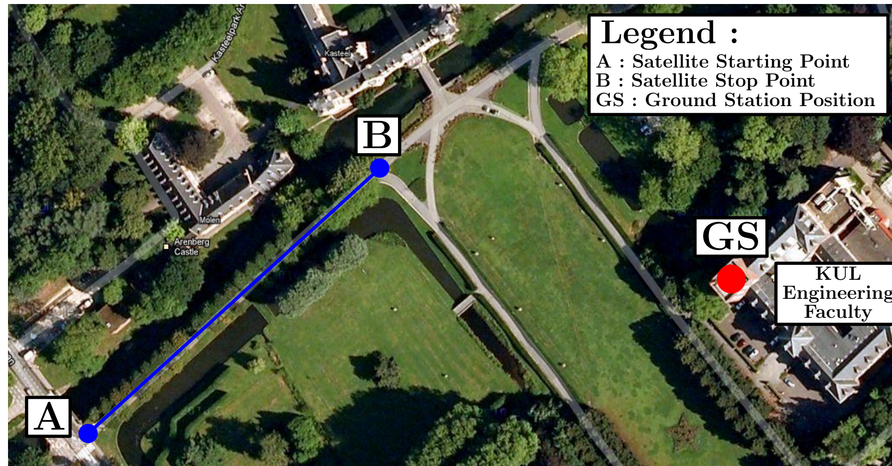


Figure 5.12: IS-HS 2 demo setup in Belgium.

A trolley, carrying the satellite, moved from A to B, while transmitting files to the ground station at GS. These files contained either statistical information or webcam images, obtained from a camera connected to the ASE's PC, as per Fig. 2.3. Statistical information included the trolley's current GPS position, CRC error counts on the uplink and other relevant SAA information. A special register on the FPGA allowed retrieval of the current CRC error count. The SCSS on the SH4 has been removed for this demonstration and replaced by an application which transmits these files.

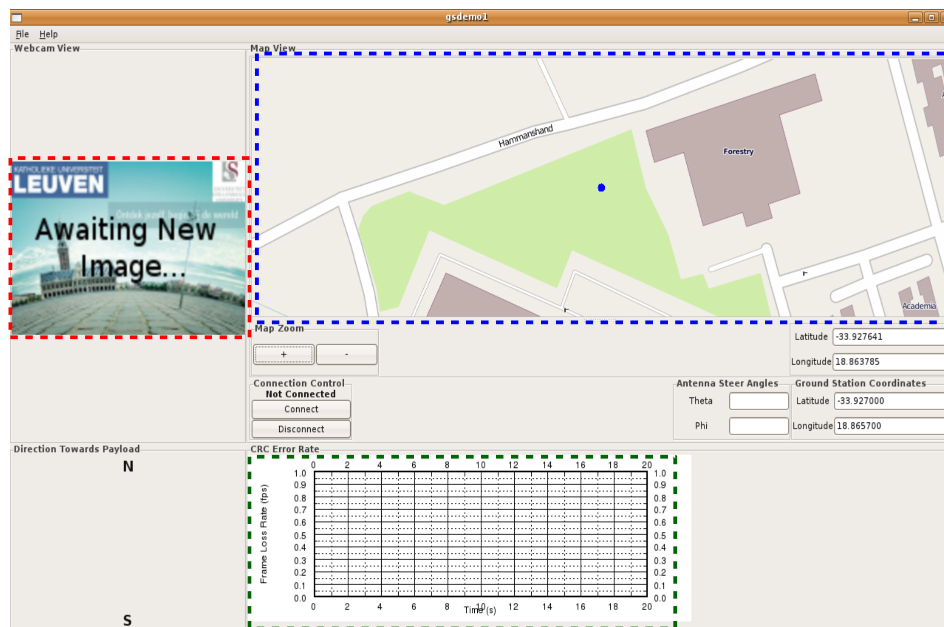


Figure 5.13: Application receiving files from ARQ on ground station FIT-PC.

Stellenbosch University http://scholar.sun.ac.za

Fig. 5.13 shows the ground station application which received these files. It connected to ARQ and is informed by the reporter thread, see Section 4.7, when new files arrive. Webcam images are displayed in the red dashed box. The blue dashed box updated the trolley position on a map, by using the received GPS coordinates. Cyclic redundancy check information is displayed in the green box. During the final demonstration, the CRC counts have been logged, as shown in Fig. 5.14.

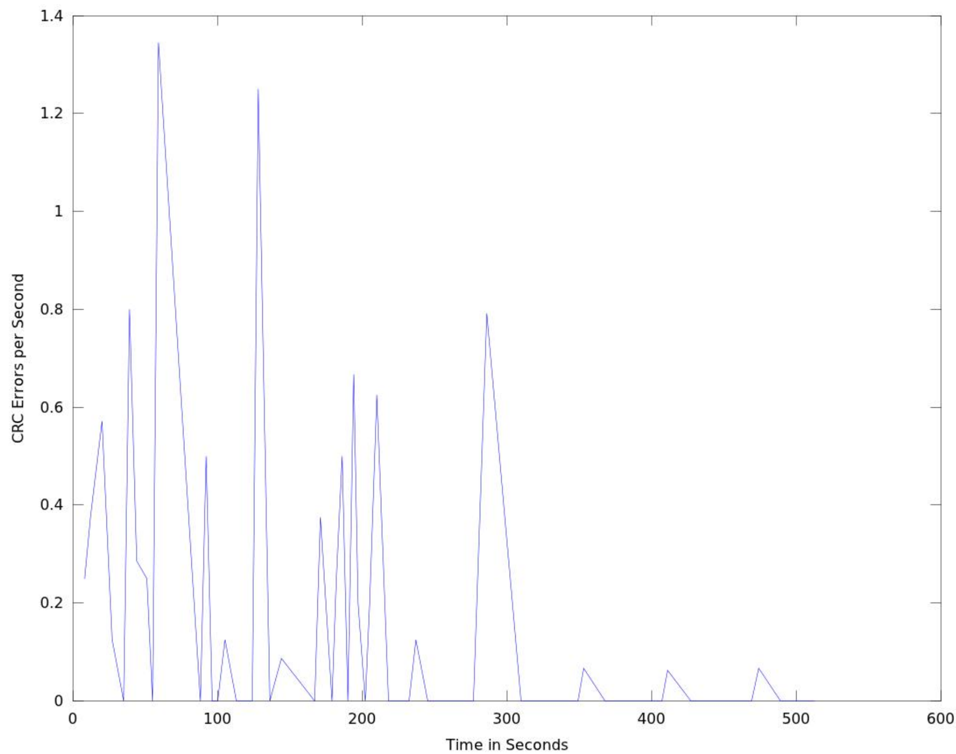


Figure 5.14: CRC error rate while moving from A to B in Fig. 5.12.

Between 0 and 300 seconds the error rate has been quite high. In the map of Fig. 5.12, a row of trees are visible next to the blue line between the satellite and ground station. As these have been wet, it severely attenuated the uplink's signal power at the particular frequency concerned. This in turn lead to many decoding failures on the (511,484) BCH decoder. These errors were then caught by the CRC module. All trees have been out of sight at destination B, hence the low CRC error count between 300 and 500 seconds as per Fig. 5.14.

Stellenbosch University http://scholar.sun.ac.za

Figure 5.15: Images received on the ground station after file transfer from the satellite platform.

A few samples of the received webcam images are shown in Fig. 5.15. Low resolution images have been chosen to enable fast transfer over the link. Corrupted data are generally always visible on an image. Apart from the exact error count measurements implemented, the lack of visible errors in the images is further proof of the correct functioning of TM and ARQ in the satellite - to ground station platform file transfer. Given the high CRC error rate from Fig. 5.14, it also shows that channel coding ensures data integrity. Corrupt TM frames are dropped while error free frames are allowed through.

5.6 Summary

This chapter presented unit - and system testing procedures to verify the correct functioning of the components as per Chapter 4. By implementing FEC on a FPGA, it is observed that data processing speeds are at least an order of magnitude faster than the RTT of an ARQ packet. Given the amount of processing required for BCH and LDPC, the FPGAs truly lower the work load on both SH4 and FIT-PC platforms. It has been confirmed that (511,484) BCH delivers the error correcting performance as intended. Both $R = 0.5$ BCH and LDPC are worthy competitors for the (511,484) BCH. Furthermore, properly optimising a $R = 0.5$ QC-LDPC will enable it to outperform the equivalent $R = 0.5$ BCH. Finally, the successful demonstration at KUL concluded that both ground station and satellite platforms function exactly as designed.

Stellenbosch University http://scholar.sun.ac.za

Chapter 6

Conclusion, Contributions and Recommendations

6.1 Conclusion and Summary

The IS-HS 2 project is quite complex, requiring very thorough integration of a relatively large number of hardware and software modules. At the commencement of this Masters project, a significant portion of the overall system has been completed, but some important gaps remained. These mostly entailed the ground-satellite platform file transfer mechanisms and ensuring the reliability thereof, as well as stable integration into the rest of the system. The research and development work documented herein, therefore, focused on channel coding and implementing a suitable file transfer protocol.

The flexible modular layout of channel coding allows modules to be easily added or removed. Other channel coding standards which adopt different CRC and randomisation schemes can be implemented by simply replacing the appropriate modules. This also holds true when testing and evaluating FEC schemes other than LDPC or BCH. Interfaces of other modules utilising channel coding would not have to be redesigned.

Channel coding ensured data integrity as required. Clearly there has been a very high error rate in communication during the Belgian demonstration. Due to low signal power, the (511,484) sometimes failed to correct all errors within a codeword. This is to be expected, but even when data containing errors passed through the decoder, CRC detected those errors and dropped the frame. Addition of FEC to channel coding meant that a communication time window could indeed be optimally used. Both $R = 0.5$ BCH and LDPC, significantly lowered the required signal power for reliable communications, as shown by simulation. An uncoded system required approximately $E_b/N_o = 16$ dB for a block error rate of 10^{-6}. By contrast, LDPC requires an $E_b/N_o = 6$ dB for the same block error rate, hence an improvement of 10 dB.

The FEC simulations also confirmed the findings in [1], that LDPC does

Stellenbosch University http://scholar.sun.ac.za

outperform BCH. Properly optimising the low hardware complexity MS decoder allows it to match BCH's performance at 6 dB SNR. At SNRs lower than 6 dB it shows a clear improvement over BCH. Reducing the parallelisation for both BCH and LDPC implementations, would allow implementation on smaller FPGAs, reducing the total system cost. Retransmitting an ARQ packet 5 times, has been shown to introduce a total delay of 460 ms per packet. This was still sufficient for transferring 10 kB files during a 15 minutes communications window. Using no parallelisation, it is estimated both decoders would introduce a latency of no more than 10 ms. Therefore parallelisation for both BCH and LDPC decoders can be reduced without affecting the required data throughput.

Interprocess communication have been separated from both TM and ARQ's frame and packet processing routines. Only the IPC module has to be changed when switching between Windows or Linux Ubuntu on a ground station. Furthermore, the message passing API hides shared memory and semaphore details from the user. Therefore, it provides a simple way of connecting two OSI layers to each other in order to exchange data.

Finally, the protocol layers ensured reliable file transfer between a ground station and the satellite platform. The simple stop-and-wait ARQ strategy managed to assemble files correctly, even when many lost packets had to be retransmitted. Apart from ARQ and TM's testing procedures, reliable file transfer was confirmed when photos have been transferred without error during the demonstration in Belgium. A very satisfying outcome was the field proven capability of the coding and protocol combination to enable data transfer reliability under quite pathological practical conditions.

6.2 Contributions to the Project

This thesis documents specific inputs and contributions to the IS-HS 2 project as well as a number of interesting results of a more general nature. Contributions specific to the project are :

- Provided the software and firmware components necessary for final communication system integration on both satellite and ground station platforms.
- Added reliable and efficient file transfer mechanisms to the communications channel.
- Designed specific subunit and integration testing procedures to verify the implemented strategies.

Further contributions to the project, but also of more general nature, include :

Stellenbosch University http://scholar.sun.ac.za

- Confirmed the findings of previous investigations that BCH and LDPC are viable FEC strategies for this type of project, to a high degree of confidence.

- Designed a parallelisable FPGA architecture for both BCH and LDPC decoders.

- Determined speed and complexity trade-offs by adjusting the amount of parallelism in the particular FEC scheme.

- Implementation of a robust FEC performance verification strategy by using a combination of Matlab and FPGA results to establish functionality of the FPGA design beyond doubt. It is also valuable to identify dysfunctional implementations.

- To provide TM and ARQ functionality to their particular layers as described by OSI specifications, thereby making the scheme available for wider applications.

- Development of a simple message passing API for operating systems other than QNX.

- Easy porting of TM and ARQ to operating systems other than QNX or Linux Ubuntu. Interprocess communication has been separated from the service provided by TM and ARQ.

- Implemented a TM protocol according to ECSS specifications. This can be used with any other transport layer protocol, as no packet specific information from that layer is required by TM.

- Implemented an elegantly simple stop-and-wait ARQ which provided reliable and sufficient data throughput for this project, but again has wider application under poor communications environments.

6.3 Recommendations

The following recommendations for possible improvements of the current systems design, as well as for a next generation design, are presented :

- Use higher girth construction techniques for LDPC's parity check matrix. High girth reduces the number of iterations required for successful decoding. This is useful when reducing the processing latency for low complexity, non-parallel LDPC decoders.

- Program crashes frequently happen when developing application layer software. This tends to leave IPC structures of ARQ in an unknown

Stellenbosch University http://scholar.sun.ac.za

state after which all OSI layers have to be manually restarted. Self restarting routines can be added to IPC when it loses connection with another layer.

- Move parameters for TM and ARQ, such as time-out settings, to configurations files. These parameters can then be tweaked during operation to optimise protocol performance for the current situation. Currently these parameters are set during compile time.

- At the time of writing this thesis, the satellite concept for this project has been cancelled. All channel coding components could be ported from FPGA to software running on a modern powerful PC. Removing FPGAs and the expensive SH4 will reduce system cost and hardware complexity.

- Move the SDR modem from a DSP to the SAA's FPGA. This would further reduce hardware complexity and system cost.

- Remove IPC between TM and ARQ layers as it adds unnecessary complexity to the software layers. Layers to be used for a particular implementation are chosen before the start of development. Therefore, dynamic addition or removal of layers are not really required. By using C compiler directives, only the necessary layers can be compiled to run as one process.

- Investigate new construction techniques for the parity check matrix of QC-LDPC. Some techniques allow the minimum Hamming distance for QC-LDPC to grow linearly as the block length increases. Larger block lengths would lead to better BLER performance.

- Consider the use of FPGA based FEC simulators. No slow serial connectivity between a PC and the FPGA, as with the simulator from this thesis, would be used. Hardware simulators tend to be much faster than software simulators. Design tools from Xilinx such as AccelDSP helps in translating Matlab implementations quickly to VHDL.

- A new initiative between SU and KUL would be investigating the application of the SAA concept, to mobile communications.

Stellenbosch University http://scholar.sun.ac.za

References

[1] F. Olivier, "An LDPC Error Control Strategy for Low Earth Orbit Satellite Communication Link Applications ," Master's thesis, University of Stellenbosch, 2009.

[2] L. Zhou, "Implementation of the Berlekamp-Massey algorithm and Peterson's algorithm in C programming language," Apr 2007, eCE Dept., University of Toronto.

[3] S. Lin and D. Costello, *Error Control Coding : Fundamentals and Applications*. Prentice-Hall, Inc., 1983.

[4] Z. Li, L. Chen, L. Zeng, S. Lin, and W. H. Fong, "Efficient Encoding of Quasi-Cyclic Low-Density Parity-Check Codes," *IEEE Transactions on Communications*, vol. 54, no. 1, pp. 71–81, Jan 2006.

[5] B. R. Elbert, *The Satellite Communication Applications Handbook*. Artech House, 1997.

[6] R. A. Nelson, "A Primer on Satellite Communications," *Via Satellite*, 1998.

[7] I. Kruger, "An aircraft based emulation platform and control model for LEO satellite antenna beam steering," Master's thesis, University of Stellenbosch, 2010.

[8] *SH7750, SH7750S, SH7750R Group*, 7th ed., Renesas Electronics Corporation, Oct 2008.

[9] (2009, Dec.) Internetworking Basics. [Online]. Available: http://docwiki.cisco.com/wiki/Internetworking_Basics

[10] Silberschatz *et al.*, *Operating System Concepts*, 7th ed. John Wiley & Sons, 2005.

[11] *System Architecture*, 5th ed., QNX Software Systems, May 2004.

[12] M. Beck *et al.*, *Linux Kernel Internals*. Addison Wesley Longman, 1996.

Stellenbosch University http://scholar.sun.ac.za

[13] K. Robbins and S. Robbins, *Unix Systems Programming.* Prentice Hall, 2003.

[14] (2011, Nov) mq_overview(7) - Linux man page. [Online]. Available: http://linux.die.net/man/7/mq_overview

[15] T. Moon, *Error Correction Coding.* John Wiley & Sons, 2005.

[16] P. Sweeney, *Error Control Coding : From Theory to Practice.* John Wiley & Sons, 2004.

[17] B. P. Lathi, *Modern Digital and Analog Communication Systems*, 3rd ed. New York: Oxford University Press, Inc., 1998.

[18] "DVB Fact Sheet : 2nd Generation Satellite," Digital Video Broadcast, Tech. Rep., Sep 2010.

[19] ECSS, "Space engineering : Space data links - Telecommand protocols, synchronisation and channel coding," European Space Agency, Tech. Rep. ECSS-C-50-04A, Nov 2007.

[20] R. G. Gallager, "Low-Density Parity-Check Codes," Ph.D. dissertation, Cambridge, Mass, Jul 1963.

[21] D. J. MacKay, "Good Error-Correcting Codes Based On Very Sparse Matrices," *IEEE Transactions on Information Theory*, vol. 45, no. 2, pp. 399–431, Mar 1999.

[22] D. MacKay and R. Neal, "Near Shannon Limit Performance of Low Density Parity Check Codes," *Electronics Letters*, vol. 33, no. 6, pp. 457–458, Mar 1997.

[23] N. Wiberg, "Codes and Decoding on General Graphs," Ph.D. dissertation, Dept. Elect. Eng. Linköping Univ., Linköping, Sweden, Oct 1996.

[24] "IEEE Standard for Information Technology - Part 11 : Wireless LAN Medium Access Control and Physical Layer Specifications," IEEE, Tech. Rep. 802.11-2007, Jun 2007.

[25] T. Lestable, E. Zimmerman, M.-H. Hamon, and S. Stiglmayr, "Block-LDPC Codes Vs Duo-Binary Turbo-Codes for European Next Generation Wireless System," *IEEE*, Feb 2007.

[26] Turbo Codes. [Online]. Available: http://www.francetelecom.com/en_EN/innovation/intellectual_property/turbo_codes

[27] W. E. Ryan, "An Introduction to LDPC Codes," Dept. Elec. Eng. and Comp. Eng., Univ. Arizona, University of Arizona, Box 210104, Tucson, AZ 85721, Tech. Rep., Aug 2003.

Stellenbosch University http://scholar.sun.ac.za

[28] J. Lu and M. Moura, "Structured LDPC Codes for High-Density Recording : Large Grith and Low Error Floor," *IEEE Transactions on Magnetics*, vol. 42, no. 2, pp. 208–213, Feb 2006.

[29] *Efficient LDPC Decoder Implementation for DVB-S2 System*, Apr 2010.

[30] T. Richardson and R. Urbanke, "Efficient Encoding of Low-Density Parity-Check Codes," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 638–656, Feb 2001.

[31] R. Ziemer and W. Tranter, *Principles of Communications*, 5th ed. John Wiley & Sons, 2002.

[32] H. Killen, *Digital Communications with Fiber Optics and Satellite Applications*. Prentice-Hall, Inc., 1988.

[33] P. Z. Peebles, *Probability, Random Variables and Random Signal Principles*. McGraw-Hill Book Co, 2001.

[34] S. J. Johnson, *Iterative Error Correction : Turbo, Low-Density Parity-Check and Repeat-Accumulate Codes*. Cambridge University Press, Nov 2009.

[35] J. Botha, "A Reusable Signal Processing Architecture for Satellite based Communication Systems," Master's thesis, University of Stellenbosch, 2011.

[36] H. Aliakbarian, V. Volski, and G. Vandenbosch, "Maximum gain of the antenna," University of Leuven, Tech. Rep., 2009.

[37] H. Aliakbarian, "An Estimation of SAA Sensitivity," University of Leuven, Tech. Rep., 2009.

[38] ECSS, "Space engineering : Communications Guidelines," European Space Agency, Tech. Rep. ECSS-E-HB-50A draft 1.4, Apr 2008.

[39] S. J. Johnson. (2008, Apr) Introducing Low-Density Parity-Check Codes. [Online]. Available: http://materias.fi.uba.ar/6624/index_files/outline_archivos/SJohnsonLDPCintro.pdf

[40] C. A. Cole, E. K. Hall, S. G. Wilson, and T. R. Giallorenzi, "Analysis and design of moderate length regular LDPC codes with low error floors," Univ. of Virginia and L-3 Communications, Tech. Rep., May 2006.

[41] C. A. Cole, S. G. Wilson, E. K. Hall, and T. R. Giallorenzi, "Regular (4,8) LDPC codes and their low error floors," Univ. of Virginia and L-3 Communications, Tech. Rep., May 2006.

Stellenbosch University http://scholar.sun.ac.za

[42] L. Sun, H. Song, V. Kumar, and Z. Keirn, "Field-Programmable Gate-Array-Based Investigation of the Error Floor of Low-Density Parity Check Codes for Magnetic Recording Channels," *IEEE Transactions on Magnetics*, vol. 41, no. 10, pp. 2983–2985, Oct 2005.

[43] Y. Han and W. E. Ryan, "LDPC Decoder Strategies for Achieving Low Error Floors," Dept. Elec. Eng. and Comp. Eng., Univ. Arizona, Tech. Rep., Dec 2007.

[44] S. J. Johnson and S. R. Weller, "A Family of Irregular LDPC Codes With Low Encoding Complexity," *IEEE Communications Letters*, vol. 7, no. 2, pp. 79–81, Feb 2003.

[45] M. P. C. Fossorier, "Quasi-Cyclic Low-Density Parity-Check Codes From Circulant Permutation Matrices," *IEEE Transactions on Information Theory*, vol. 50, no. 8, pp. 1788–1793, Aug 2004.

[46] S. Myung, K. Yang, and J. Kim, "Quasi-Cyclic LDPC Codes for Fast Encoding," *IEEE Transactions on Information Theory*, vol. 51, no. 8, pp. 2894–2901, Aug 2005.

[47] W. Stallings, *Data and Computer Communications*, 7th ed. Pearson Prentice Hall, 2004.

[48] *Synthesis and Simulation Design Guide*, v11.4 ed., Xilinx, Dec 2009.

[49] S. Ruckmani and P. Angbalagan, "High Speed Cyclic Redundancy Check for USB," *DSP Journal*, vol. 6, no. 1, pp. 45–50, Sep 2006.

[50] I. L. W. Couch, *Digital and Analog Communication Systems*, 7th ed. Pearson Prentice Hall, 2007.

Stellenbosch University http://scholar.sun.ac.za

Appendices

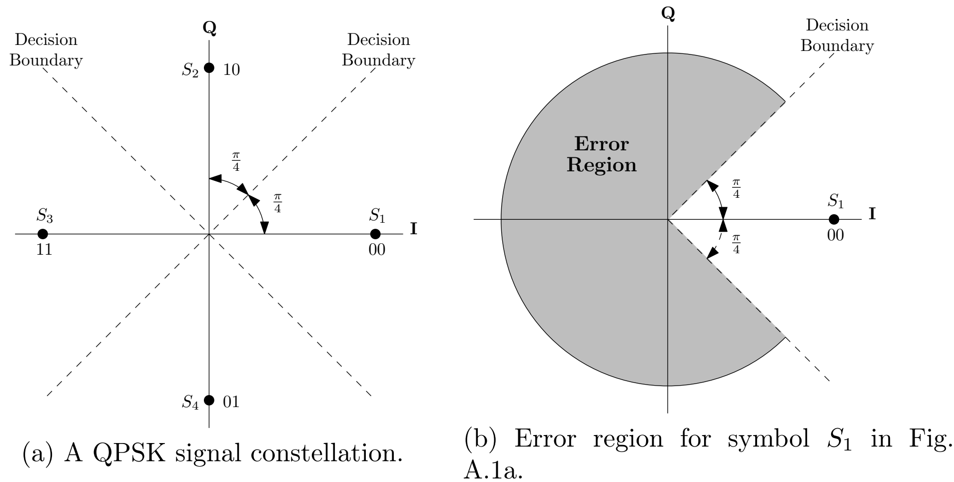
Stellenbosch University http://scholar.sun.ac.za

Appendix A

Mathematical Derivations

A.1 QPSK Bit Error Probability Analysis

Fig. A.1a shows a QPSK signal constellation with four two-bit symbols S_1 to S_4. Symbols are indicated by the black dots on both axis. The in-phase axis is indicated by **I** while **Q** represents the quadrature axis. Each symbol differs by $\frac{\pi}{2}$ radians in phase from its neighbour. If a received symbol's phase deviates by more than $\frac{\pi}{4}$ radians due to phase noise, the modem makes an error. The dashed line represents the boundary which a symbol may not cross before being mistaken for another symbol. For example if symbol S_1's phase deviates into the grey area of Fig. A.1b, it will be mistaken for another symbol. All symbols will be equally affected by noise, therefore BEP analysis will continue by using S_1 as an example.



(a) A QPSK signal constellation.

(b) Error region for symbol S_1 in Fig. A.1a.

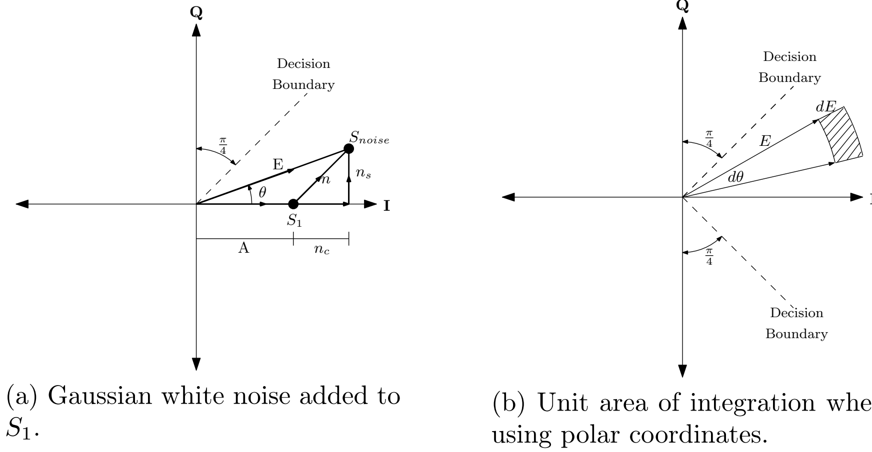
Figure A.1: QPSK symbol decision - and error regions.

Stellenbosch University http://scholar.sun.ac.za

Eq. A.1.1 shows noise vector $n(t)$ [17] consisting of both in-phase and quadrature components, $n_i(t)$ and $n_q(t)$ respectively :

$$n(t) = n_i(t) + n_q(t) \tag{A.1.1}$$

Adding $n(t)$ to symbol S_1, represented by phasor $y_{S_1}(t) = Ae^{j\omega t}$, results in S_{noise} with $y_{noise}(t) = Ee^{j\omega t + \theta}$ as shown in Fig. A.2a. Scalar E is symbol S_1's amplitude with added amplitude noise while θ indicates phase noise added by $n(t)$.



(a) Gaussian white noise added to S_1.

(b) Unit area of integration when using polar coordinates.

Figure A.2: Vector representation of Gaussian noise added to symbol S_1.

Since $n_i(t)$ and $n_q(t)$ of Eq. A.1.1 are statistically independent Gaussian random variables, we have :

$$\begin{aligned}
\rho(n_i, n_q) &= \rho(n_i)\rho(n_s) \\
&= \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-n_i^2}{2\sigma^2}} \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-n_q^2}{2\sigma^2}} \\
&= \frac{1}{2\pi\sigma^2} e^{\frac{-(n_i^2+n_q^2)}{2\sigma^2}}
\end{aligned} \tag{A.1.2}$$

The term $\rho(n_i, n_q)$ is a joint Gaussian PDF. In order to write Eq. A.1.2 i.t.o amplitude, E, and phase noise, θ, we first show the following :

$$\begin{aligned}
E^2 &= (A + n_i)^2 + n_q^2 \\
n_i^2 + n_q^2 &= E^2 - A^2 - 2An_i \\
&= E^2 - 2A(A + n_i) + A^2 \\
&= E^2 - 2AEcos(\theta) + A^2
\end{aligned} \tag{A.1.3}$$

Substituting Eq. A.1.3 into Eq. A.1.2 leads to :

Stellenbosch University http://scholar.sun.ac.za

$$\begin{aligned}\rho(\theta, E) &= \left(\frac{1}{2\pi\sigma^2}\right) e^{\frac{-(E^2-2AEcos(\theta)+A^2)}{2\sigma^2}} \\ &= \left(\frac{1}{2\pi\sigma^2} e^{\frac{-A^2}{2\sigma^2}}\right) e^{\frac{-(E^2-2AEcos(\theta))}{2\sigma^2}} \end{aligned} \tag{A.1.4}$$

Symbol S_{noise} in Fig. A.2a will always be interpreted by the modem as S_1, as long as it stays in the white area of Fig. A.1b. Integrating PDF A.1.4 over this area gives the probability of S_{noise} falling in this region, hence the probability of receiving S_1 successfully. Using polar coordinates and the unit area in Fig. A.2b, leads to the following :

$$\begin{aligned}P_{success_S_1_unit} &= \rho(\theta, E).E.d\theta.dE \\ P_{success_S_1} &= \int_{\frac{-\pi}{4}}^{\frac{\pi}{4}} \left(\int_0^\infty \rho(\theta, E).E.dE\right) d\theta \end{aligned} \tag{A.1.5}$$

Evaluating the integral between brackets of Eq. A.1.5 marginalises the PDF from Eq. A.1.4. This is shown below :

$$\begin{aligned}\rho_\theta(\theta) &= \int_0^\infty \rho_{\theta E}(\theta, E).E.dE \\ &= \frac{1}{2\pi\sigma^2} e^{\frac{-A^2}{2\sigma^2}} \int_0^\infty \left(e^{\frac{-(E^2-2AEcos(\theta))}{2\sigma^2}}\right) EdE \\ &= C_0 \int_0^\infty \left(e^{\frac{-(E^2-2AEcos(\theta))}{2\sigma^2}}\right) EdE \end{aligned} \tag{A.1.6}$$

where

$$C_0 = \frac{1}{2\pi\sigma^2} e^{\frac{-A^2}{2\sigma^2}} \tag{A.1.7}$$

Re-arranging Eq. A.1.6 we have :

$$\begin{aligned}\rho_\theta(\theta) = &\left(-2\sigma^2 C_0\right) \left(\frac{1}{2}\right) \int_0^\infty \left(\frac{-2E + 2Acos(\theta)}{2\sigma^2}\right) \left(e^{\frac{-E^2+2AEcos(\theta)}{2\sigma^2}}\right) dE \\ &+ \left(2\sigma^2 C_0\right) \left(\frac{1}{2}\right) \int_0^\infty \left(\frac{2Acos(\theta)}{2\sigma^2}\right) \left(e^{\frac{-E^2+2AEcos(\theta)}{2\sigma^2}}\right) dE \\ = &\left(-\sigma^2 C_0\right) \left(e^{\frac{-E^2+2AEcos(\theta)}{2\sigma^2}}\right) \Bigg|_0^\infty + \left(AC_0cos(\theta)\right) \int_0^\infty \left(e^{\frac{-E^2+2AEcos(\theta)}{2\sigma^2}}\right) dE \\ = &\left(-\sigma^2 C_0\right) (0-1) + \left(AC_0cos(\theta)\right) \int_0^\infty \left(e^{\frac{-E^2+2AEcos(\theta)}{2\sigma^2}}\right) dE \end{aligned}$$

Stellenbosch University http://scholar.sun.ac.za

$$
\begin{aligned}
=&\sigma^2 C_0 + (AC_0 cos(\theta)) \int_0^{\infty} \left(e^{\frac{-(E^2 - 2AEcos(\theta) + A^2cos^2(\theta) - A^2cos^2(\theta))}{2\sigma^2}} \right) dE \\
=&\sigma^2 C_0 + (AC_0 cos(\theta)) \int_0^{\infty} \left(e^{\frac{-((E-Acos(\theta))^2 + A^2cos^2(\theta))}{2\sigma^2}} \right) dE \\
=&\sigma^2 C_0 + (AC_0 cos(\theta)) \left(e^{\frac{A^2cos^2(\theta)}{2\sigma^2}} \right) \int_0^{\infty} \left(e^{\frac{-(E-Acos(\theta))^2}{2\sigma^2}} \right) dE \qquad \text{(A.1.8)}
\end{aligned}
$$

Now let $y = \frac{(E-Acos(\theta))}{\sigma^2}$, then :

$$
\begin{aligned}
\frac{dy}{dE} &= \frac{1}{\sigma} \\
dE &= \sigma dy \qquad \text{(A.1.9)}
\end{aligned}
$$

Using transformation A.1.9 on Eq. A.1.8 we get :

$$
\begin{aligned}
\rho_\theta(\theta) =&\sigma^2 C_0 + (\sigma AC_0 cos(\theta)) \left(e^{\frac{A^2cos^2(\theta)}{2\sigma^2}} \right) \int_{\frac{-Acos(\theta)}{\sigma}}^{\infty} \left(e^{\frac{-y^2}{2}} \right) dy \\
=&\sigma^2 C_0 \left[1 + \left(\frac{\sigma Acos(\theta)}{\sigma^2} \right) \left(e^{\frac{A^2cos^2(\theta)}{2\sigma^2}} \right) \left(\frac{\sqrt{2\pi}}{1} \frac{1}{\sqrt{2\pi}} \right) \int_{\frac{-Acos(\theta)}{\sigma}}^{\infty} \left(e^{\frac{-y^2}{2}} \right) dy \right] \\
=&\sigma^2 C_0 \left[1 + \left(\frac{Acos(\theta)}{\sigma} \right) \left(e^{\frac{A^2cos^2(\theta)}{2\sigma^2}} \right) \left(\sqrt{2\pi} \right) \right. \\
&\left. \times \left(1 - Q\left(\frac{Acos(\theta)}{\sigma} \right) \right) \right] \qquad \text{(A.1.10)}
\end{aligned}
$$

where

$$
Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^{\infty} \left(e^{\frac{-a^2}{2}} \right) da \qquad \text{(A.1.11)}
$$

Substituting Eq. A.1.10 into Eq. A.1.5 gives :

$$
P_{success_S_1} = \int_{\frac{-\pi}{4}}^{\frac{\pi}{4}} \rho_\theta(\theta) d\theta \qquad \text{(A.1.12)}
$$

Phase noise is of interest when performing BEP analysis for QPSK. Note that $P_{success_S_1}$ indicates the probability of successfully receiving a symbol in the presence of phase noise θ. Using Eq. A.1.12, the symbol error probability is determined :

$$
P_{error_S_1} = 1 - P_{success_S_1}
$$

Stellenbosch University http://scholar.sun.ac.za

$$= 1 - \int_{\frac{-\pi}{4}}^{\frac{\pi}{4}} \rho_\theta(\theta) d\theta \tag{A.1.13}$$

However, a BEP is of interest in this analysis. Most symbol errors occur when the modem mistakes the correct symbol with its neighbour [17]. Looking at Fig. A.3, each symbol is arranged such that it differs only in one bit from its neighbour. This is called Gray coding [17]; a technique that minimises the amount of bit errors per symbol error.

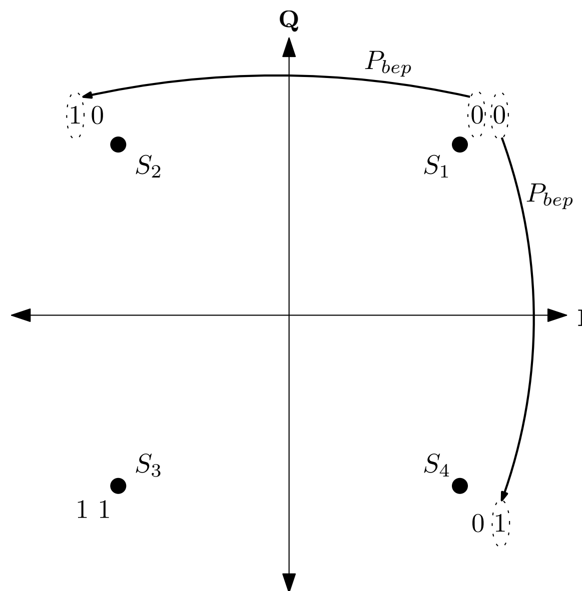


Figure A.3: Gray coding scheme for the symbols of a QPSK constellation.

Fig. A.3 also assumes equiprobability, P_{bep}, for each bit of a symbol changing its value. This leads to the following BEP :

$$P_{bep} = \frac{P_{error_S_1}}{2} \tag{A.1.14}$$

A FEC codeword will consist of a sequence of bits as discussed in Chapter 2, Section 2.5. The bit error probability of Eq. A.1.14 can be used to determine the length of such a codeword by means of Bernoulli trials. Assuming a t-error correcting code, leads to codeword error probability :

$$P_{cep} = \sum_{i=t+1}^{n} P(i,n) \tag{A.1.15}$$

Since FEC can correct up to t errors, only the occurrence of errors greater than t bits affects the codeword error probability. Therefore term $P(i,n)$ of

Stellenbosch University http://scholar.sun.ac.za

Eq. A.1.15 is the probability of making $i > t$ errors in a n-bit codeword. Since there are $\binom{n}{i}$ to make i errors in a codeword [17], $P(i, n)$ can be written as :

$$P(i, n) = \binom{n}{i} P_{bep}^{i} (1 - P_{bep})^{n-i} \tag{A.1.16}$$

Eq. A.1.16 is also known as a Bernoulli trial [33]. Substituting Eq. A.1.16 into Eq. A.1.15 gives the final codeword error probability :

$$P_{cep} = \sum_{i=t+1}^{n} \binom{n}{i} P_{bep}^{i} (1 - P_{bep})^{n-i} \tag{A.1.17}$$

Stellenbosch University http://scholar.sun.ac.za

A.2 Signal-to-Noise Ratio for Simulations

This section derives the relationship between signal-to-noise ratios A^2/σ^2 and E_b/N_o for QPSK. The latter is expressed as energy per bit to noise power spectral density whereas the first ratio indicates average signal power to average noise power. A time domain binary phase shift keying (BPSK) signal is shown in Fig. A.4.

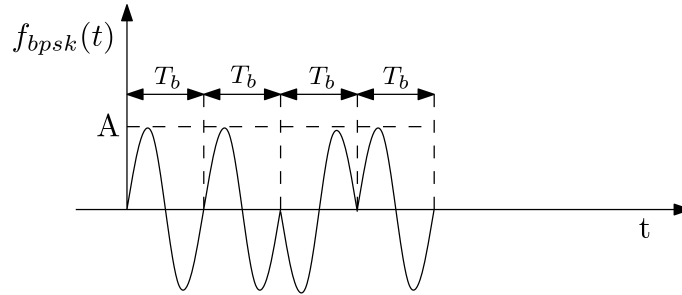


Figure A.4: Time domain BPSK signal.

Unit T_b indicates transmission time per BPSK symbol. Note that a BPSK symbol is equivalent to a bit. A single bit is represented by :

$$f_{bit}(t) = \Pi\left(\frac{t - \frac{nT_b}{2}}{T_b}\right) \times Asin(2\pi f_0 t + \theta), \quad n > 0 \quad , \quad \text{all odd } n \qquad \text{(A.2.1)}$$

where Π is a time shifted rectangular pulse of width T_b and unity amplitude. Taking the Fourier transform of $f_{bit}(t)$ leads to a BPSK bit's frequency spectrum in Fig. A.5.

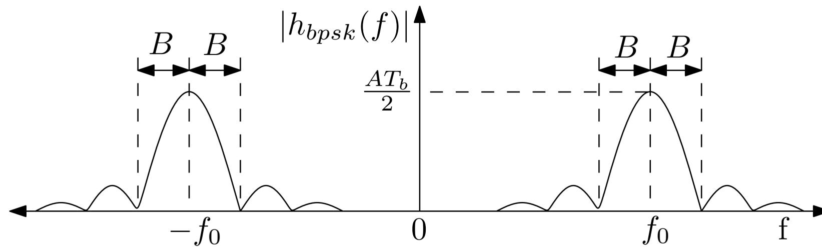


Figure A.5: Frequency domain of a BPSK bit.

The Fourier transform of Π in Eq. A.2.1 is a $sin(x)/x$ function. Although this function has a infinite bandwidth, most of its power lies in the main lobe. Therefore its effective bandwidth is $B = 1/T_b$ [17] and is shown in Fig. A.5. Noise is also present on this signal when entering the receiver. Placing a bandpass filter (BPF) of bandwidth $2B$, unity gain and centre frequency f_0 at

Stellenbosch University http://scholar.sun.ac.za

the receiver, limits this noise power. The average power of a sinusoid having amplitude A is $A^2/2$. In a AWGN channel, variance σ^2 represents average noise power [15]. Expressing this as a ratio, leads to the SNR for BPSK :

$$SNR = \frac{\frac{A^2}{2}}{\sigma^2} = \frac{A^2}{2\sigma^2} \tag{A.2.2}$$

Gaussian noise has a power spectral density of $N_o/2$, where N_o is a constant [15]. Filtering the signal in Fig. A.5 with a BPF as mentioned before, the total noise power is :

$$N_p = 2 \times \frac{N_o}{2} \times 2B \tag{A.2.3}$$

Knowing that σ^2 represents the average noise power, Eq. A.2.2 is transformed using Eq. A.2.3 :

$$\begin{aligned} SNR &= \frac{A^2}{2\sigma^2} \\ &= \frac{A^2}{2 \times 2 \times \frac{N_o}{2} \times 2B} \end{aligned} \tag{A.2.4}$$

Since $B = 1/T_b$ we have :

$$SNR = \frac{A^2 T_b}{2 \times 2 \times N_o} \tag{A.2.5}$$

The average energy of a sinusoid over period T_b is given by $E_b = A^2 T_b/2$ [50]. Using this, Eq. A.2.5 is transformed :

$$SNR = \frac{E_b}{2 \times N_o} \tag{A.2.6}$$

Comparing Eqs. A.2.2 and A.2.6 and letting $A = A_{bpsk}$, we get the following for BPSK :

$$SNR_{bpsk} = \frac{A_{bpsk}^2}{\sigma^2} = \frac{E_b}{N_o} = \frac{E_{s_bpsk}}{N_o} \tag{A.2.7}$$

where E_b and E_{s_bpsk} represent bit and symbol energies. A QPSK receiver uses a BPSK receiver for its orthogonal channels, I and Q, respectively. Fig. A.6 shows that for QPSK channels I and Q, the amplitude of symbol S_1 is

Stellenbosch University http://scholar.sun.ac.za

$A_{qpsk} = \sqrt{A_{bpsk}^2 + A_{bpsk}^2}$. Substituting A_{bpsk} with A_{qpsk} in Eq. A.2.7 leads to the following :

$$\begin{aligned} SNR_{qpsk} &= \frac{A_{qpsk}^2}{\sigma^2} \\ &= \frac{2A_{bpsk}^2}{\sigma^2} \\ &= \frac{2E_b}{N_o} \end{aligned} \tag{A.2.8}$$

Therefore we have :

$$\frac{A_{qpsk}^2}{\sigma^2} = \frac{2E_b}{N_o} = \frac{E_{s_qpsk}}{N_o} \tag{A.2.9}$$

Comparing Eqs. A.2.7 and A.2.9, it can be seen that QPSK uses twice the energy compared to BPSK.

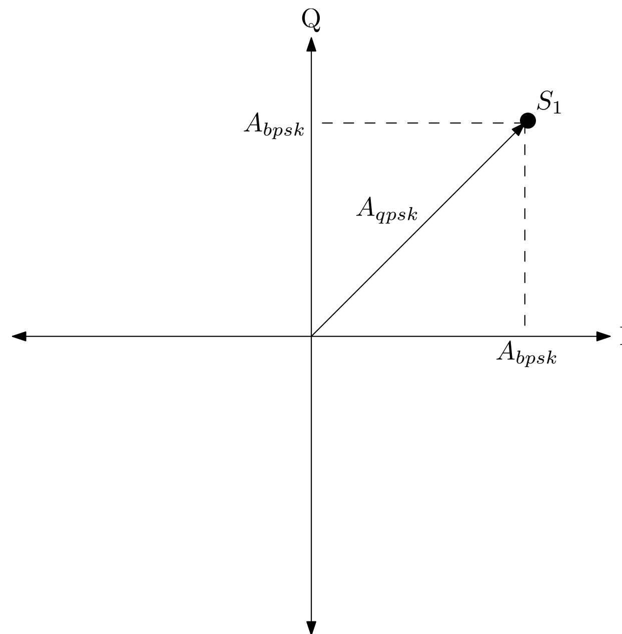


Figure A.6: A QPSK symbol amplitude i.t.o two BPSK symbols on channels I and Q.