A Modular Transceiver Platform for Human Body Communications

Kim Taylor

B.Eng. (Information Technology), B.Sc. (Applied Mathematics)

College of Engineering and Science

Victoria University

SUBMITTED IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTERS BY RESEARCH

March 2016

Abstract

Increasing interest in self health monitoring and performance analysis has resulted in a growing number of human health sensor devices. Such devices usually take the form of small, wearable and battery powered electronic systems that emphasise low power consumption and minimum discomfort to the user. While such sensors may be helpful on their own, increased benefits can be realised though the implementation of an integrated sensor network around the human body. In 2012, the IEEE 802.15.6 standard was introduced to provide a common communications protocol that may be adopted by sensor devices using three types of physical layer communications methods: ultra wideband radio frequency, narrow band radio frequency and human body communications. While narrow band and ultra wideband communications utilise traditional electromagnetic wave propagation, human body communications relies on electrostatic coupling between the transmitter and receiver. Electrostatic coupling has the benefit of reduced attenuation when used in body area networking applications.

This thesis presents an implementation of an IEEE 802.15.6 compliant transceiver using the human body communications physical layer. The thesis covers the design and implementation of the digital and analogue electronics necessary to meet the requirements of the standard, as well as describing a prototype transceiver that was developed to confirm correct operation of the design. The transceiver design has been developed in full for the purpose of this thesis.

The prototype transceiver is capable of sustaining up to 763Kbps at a bit error rate of 0.21 at the maximum data date, or for improved quality of service, lower data rates may be used with a minimum bit error rate of 0.06. Complete FPGA proven

hardware descriptions of the digital system are provided in addition to a mixed signal testbench, allowing end-to-end simulation of the entire communications channel.

Master by Research Declaration

"I, Kim Taylor, declare that the Master by Research thesis entitled A Modular Transceiver Platform for Human Body Communications is no more than 60,000 words in length inlcuding quotes and exclusive of tables, figures, appendices, bibliography, references and footnotes. This thesis contains no material that has been submitted previously, in whole or in part, for the award of any other academic degree or diploma. Except where otherwise indicated, this thesis is my own work."

Kim Taylor, March 2016

Acknowledgments

I would like to express my sincere gratitute towards a number of people whose contributions have made this thesis possible.

My profound thanks to my supervisor, Dr. Daniel Lai, for his guidance throughout the entire process. Thank you Daniel for providing an introduction into the field of human body communications, helping with the development of a research concept, for providing constructive crititism and suggestions for improvements and for your help with all aspects of completing a research degree.

I would also like to thank Prof. Michael Faulkner for his advice on hardware design and for sharing his profound knowledge on the subject of RF design and, in particular, fault diagnosis. Your timely advice has helped to make this project a success.

In addition to those who have helped me academically, I would also like to thank my mother and father, Mark and Pauline Taylor for their continuous assistance and for always encouraging my interests in the field of electronics, and my partner Anne, for her tireless support and patience.

List of Publications

Refereed Conference Papers

- K. Taylor and D. Lai, "An Empirical Measurement of Signal Attenuation and BER of IEEE 802.15.6 HBC using a phantom solution," in Proceedings of the 10th International Conference on Body Area Networks, Bodynets '15, (Sydney, Australia), 2015.

Contents

List of Tables

List of Figures

List of Symbols

ACL	Asynchronous Connection-oriented Logical Transport
ADC	Analogue to Digital Conversion
AGC	Automatic Gain Control
AGC	Automatic Gain Control
BAN	Body Area Network
BJT	Bipolar Junction Transistor
BPSK	Binary Phase Shift Keying
CMOS	Complementary Metal-Oxide Semiconductor
Comma	Packet marker for synchronisation.
CRC	Cyclic Redundancy Check
CSMA	Carrier sense multiple access with collision avoidance
DLL	Delay Locked Loop
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSSS	Direct Sequence Spread Spectrum

FM	Frequency Modulation
FPGA	Field Programmable Gate Array
FSC	Frequency Selective Code
FSDT	Frequency Selective Digital Transmission
FSK	Frequency Shift Keying
Group delay	The delay through a filter, independent of frequency.
HBC	Human Body Communications
HCI	Host Controller Interface
HDL	Hardware Description Language
HSSP	Host Sourced Serial Programming
IC	Integrated Circuit
IP	Internet Protocol
IR	Impulse Radio
IRQ	Interrupt Request
JTAG	Joint Test Action Group
L2CAP	Logical Link Control and Adaptation Protocol - Part of the Bluetooth stack
LDO	Low Dropout Linear Voltage Regulator
LFSR	Linear feedback shift register
LUT	Look Up Table
MOSFET	Metal Oxide Semiconductor Field Effect Transistor

MTU	Maximum Transmission Unit
OFDM	Orthogonal Frequency Division Multiplexing
OOK	On-Off Keying
OpAmp	Operational Amplifier
PAN	Personal Area Network
PLL	Phase Locked Loop
PPM	Pulse Position Modulation
RFCOMM	Bluetooth protocol for TTY emulation
RI	Rate Indicator
RTL	Register Transfer Level
S2P	Serial to Parallel
SFD	Start Frame Delimiter
SMPS	Switch Mode Power Supply
SNR	Signal to Noise Ratio
SPI	Serial Peripheral Interface
Spice	Circuit analysis application. Develped at Berkeley University California.
Spread Factor	The length of the IEEE 802.15.6 FSC.
TCP	Transmission Control Protocol
TTY	Teletype - serial character device

tun	Linux kernel module providing a bridge between user space and the IP stack
UART	Universal asynchronous receiver/transmitter
UWB	Ultra Wide Band
VGA	Variable Gain Amplifier
VGA	Variable Gain Amplifier

1. Introduction

The proliferation of body health monitoring devices is on the rise. Increased interest in performance monitoring in the sports industry, patient monitoring in the health industry and self-monitoring amongst the general population has given rise to a proliferation of low cost wearable sensors [5]. In most cases body health measurement devices take the form of small battery powered devices that enable gait analysis, fall detection, sampling of heart rate and glucose levels to name only a few possibilities. A comprehensive list of proposed applications has been published by the IEEE [6]. The mobile health industry is estimated to reach $8 billion USD by 2018[7].

While such sensors are often capable of providing useful information independently, there is a growing number of applications that require information from multiple sources to give the user a more holistic analysis of their health or performance. Creating a network of health sensors enables complex health monitoring applications. Integral to the implementation of a network of health sensors is a communications protocol capable of safely and securely transferring the data gathered by each sensor to a hub where more complete processing may take place. A hub may be implemented with a wrist watch or smart phone. It is important that communications be wireless, so as to minimise impact on the mobility of the user. In response to this need, the IEEE published the 802.15.6 standard describing a low power networking protocol for short range human body communications (HBC) in the vicinity of, or inside a human body [2].

Existing communications solutions for the creation of a personal area network (PAN) include IEEE 802.15.1 (Bluetooth), IEEE 802.15.4 (Zigbee) and ANT, a proprietary standard developed by Dynastream Innovations [8]. These existing technologies

have been developed for short range wireless communications. IEEE 802.15.1 serves as an alternative to wired connections for consumer electronics applications [9], while IEEE 802.15.4 places an emphasis on low complexity, low power consumption and lower data rates for connectivity between inexpensive devices [10]. The ANT protocol is targeted at short range wireless connections between low power devices and has made inroads in the sport, wellness and home health markets [8]. The point of difference between these existing technologies and the emerging IEEE 802.15.6 standard is the focus on communications over and around the human body.

In addition to traditional radio frequency based communications, IEEE 802.15.6 presents a additional option for the physical layer referred to as electric field communication technology within the standard. IEEE 802.15.6 is the first attempt to standardise a communications protocol using electric field communication. The communications channel consists of electric field propagation over human tissue, using either galvanic or capacitive coupling. The motivation behind the adoption of electric field communications is based on observations of reduced attenuation when compared to RF communications. This reduction may be exploited to enable lower transmitter power requirements, resulting in increased battery life for wearable sensors [11].

The network topology of a body area network (BAN) is specified in section 4.2 of IEEE 802.15.6. Each BAN consists of exactly one hub and several nodes. Hubs may communicate with nodes either directly in a star topology, or indirectly through a single relay node. Node to node communications are only allowed for relay purposes. Several hubs may co-exist on a single human subject, however channel access must be negotiated between hubs.

While IEEE 802.15.6 has been available since 2012, it has not yet been widely adopted and hardware implementations are not currently available to system designers interested in creating health or performance monitoring applications. This thesis aims to describe such a hardware implementation that may be used as a platform for further research into human body communications, or for implementation of IEEE 802.15.6 compliant body area networks. The IEEE 802.15.6 standard describes in detail the transmitter architecture, but leaves implementation of the re-

ceiver open to the designer. This presents a significant hurdle for anyone interested in developing a transceiver, as the designer must consider not only the implementation details, but the overall architecture of the transceiver. This thesis presents a possible receiver architecture that may be implemented using FPGA technology.

Accompanying this thesis is a code repository consisting of hardware descriptions and firmware that may enable any user to implement an IEEE 802.15.6 transceiver using low cost, readily available FPGA hardware. All transceiver hardware descriptions have been developed as part of this work. The thesis describes in detail the operation of the transceiver and presents a prototype transceiver that has been developed to test the efficacy of the standard. All software has been written in C, and the hardware descriptions are all written in VHDL. The design has been synthesised on a Xilinx Spartan 6 low end FPGA, however it has been designed with portability in mind and may be targeted to other platforms with minimal modification. Portability requirements are addressed in Chapter 3.

In addition to the hardware descriptions, a mixed signal test bench is provided, allowing for performance analysis of hardware modifications. The test bench consists of a digital time based simulation environment coupled with tools to export the transmitted data stream to a Spice model that may be used to model the transmitter mask, the communications channel and the receiver analogue electronics. The distorted data stream is then sampled back into the digital domain for analysis of the receiver hardware.

1.1. Project Goals

This thesis specifically aims to:

1. Implement the HBC component of the IEEE 802.15.6 standard using readily available, low cost FPGA technology.
2. Provide a modular platform enabling further research into human body communications, including improvements to the IEEE 802.15.6 standard.

3. Provide hardware descriptions in VHDL for a receiver architecture that is capable of reliably decoding IEEE 802.15.6 packets.

1.2. Organisation of thesis

This thesis consists of 5 parts and each is presented as a chapter. A brief outline of each chapter is presented.

- Chapter 2 presents a review of the available literature on the topic of human body communications. The concept of capacitive coupling is introduced as the basis of the human body communications channel. A study of the IEEE 802.15.6 standard follows, with an emphasis on the implications for transceiver implementation. The chapter ends with a survey of existing published transceivers and highlights the need for a modular reconfigurable transceiver implementation for research purposes.
- Chapter 3 describes the digital component of the transceiver. The chapter describes each of the hardware description modules implementing the receiver in detail. This chapter may be read in conjunction with the hardware descriptions included in the appendix. A description of the simulation and test environments follows, including a discussion of the requirements for porting the design to another hardware platform. Test results of the digital electronics are presented at the end of the chapter.
- Chapter 4 discusses the analogue electronics required by the transceiver. The chapter is divided into two sections: one for the transmitter and one for the receiver. The transmitter spectral mask requirements are established before developing a filter architecture that adheres to the IEEE 802.15.6 transmit spectrum mask limits. The simulation and test results of the filter design are presented. The remainder of the chapter is devoted to the receiver architecture, consisting of amplification and a hard decision decoder.
- Chapter 5 details the overall system architecture of the constructed prototype transceiver. This chapter covers PCB design, power supply and firmware re-

quirements. The chapter concludes with a discussion of the performance of the prototype, including test results.

- Chapter 6 presents the conclusion of the thesis and lays the groundwork for future developments to improve the throughput and noise immunity of the transceiver.

2. Literature Review

This chapter aims to investigate existing literature on the topic of human body communications, focussing on three key topics. A brief review of the difference between different transceiver to body coupling methods, a discussion of the requirements of the IEEE 802.15.6 standard, and a survey of HBC transceiver architectures that have been presented in literature. The survey will conclude with a basis for a transceiver architecture that allows further research into HBC using a configurable modular platform.

2.1. Communications channel

Human body communications systems are typically characterised by the type of coupling method used for the transceiver/body interface. The two main methods that have been explored in literature are galvanic coupling and capacitive coupling [11].

When using galvanic coupling, both the transmitter electrode and ground terminals are coupled onto the user's body. In this case, the body may be considered as a transmission line with the electromagnetic wave propagating along the limbs or torso. Capacitive coupled communications rely on the capacitive coupling between the human body and its surrounding environment.

The two methods differ considerably when considering attenuation at different frequencies. Wegmueller *et al.* concludes that galvanic coupling is promising in the 10KHz to 1MHz band, while Bae *et al.* demonstrated good performance in the

40MHz to 120MHz band using capacitive coupling [12] [13]. The fundamental frequency of the communications protocol presented in IEEE 802.15.6 is 21MHz and as such is better suited to capacitive coupling. This is the method employed by the prototype transceiver.

Figure. 2.1 shows a lumped element model first proposed by Zimmerman to model capacitive coupling. The origin of each coupling is listed in Table. 2.1, along with approximate values found by Zimmerman for a wrist watch based implementation.

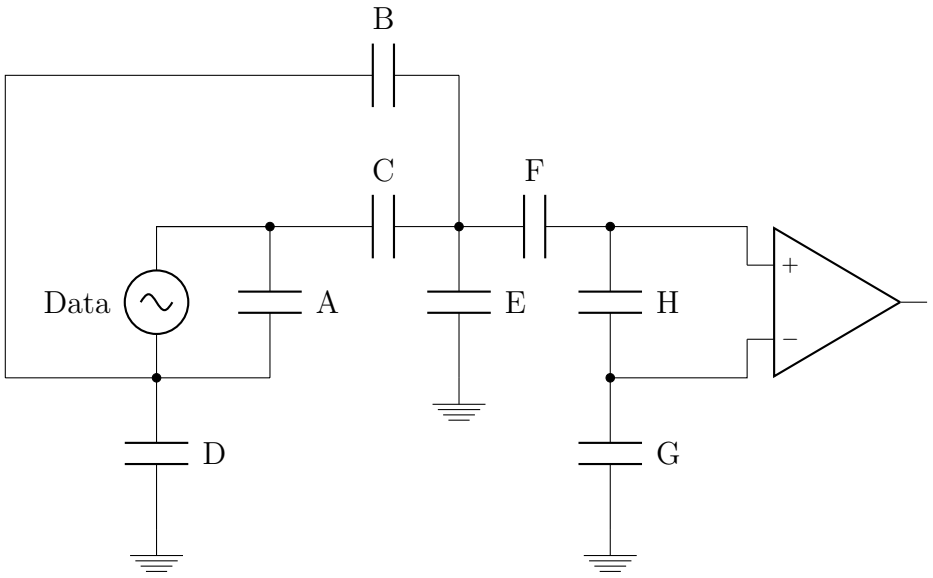

Figure 2.1.: Zimmerman's capacitive coupling model [1]

It is possible to modify the prototype transceiver for galvanic coupling through the use of an external dual contact electrode. The centre frequency of the transceiver may be freely modified within the specifications of the FPGA device, in order to improve attenuation when using galvanic coupling, however this will break compliance with IEEE 802.15.6 which specifies a fundamental transmit frequency of 21MHz.

The prototype achieves contact with the user's skin through the use of an electrode placed on the analogue PCB. The electrode dimensions have been chosen to approximately match the dimensions of typical electrocardiogram (ECG) electrodes. The

Table 2.1.: Zimmerman's capacitive coupling model

Capacitor	**Coupling**	**Approximate value**
A	Transmitter electrode to transmitter PCB ground plane	1nF
B	Transmitter PCB ground plane to body	Not given
C	Transmitter electrode to body	10pF
D	Transmitter PCB ground plane to earth	10fF
E	Body to earth	110pF
F	Body to receiver electrode	10pF
G	Receiver PCB ground plane to earth	1nF
H	Receiver electrode to receiver PCB ground plane	10fF

coupling between the transceiver and the surrounding environment is a consequence of the PCB ground plane.

2.2. The IEEE 802.15.6 standard

The IEEE 802.15.6 standard describes three separate communications systems for "short range, wireless communications in the vicinity of, or inside, a human body" [2]. The three systems described are narrowband RF communications, ultra wideband communications and human body communications (HBC). This thesis is concerned with human body communications, as described in IEEE 802.15.6 chapter 10.

The standard describes the transmitter requirements, leaving receiver implementation up to the designer. A description of the standard, relevant to HBC follows.

2.2.1. Modulation

The modulation scheme used by IEEE 802.15.6 HBC is referred to as frequency selective digital transmission (FSDT). A frequency selective code (FSC) consists of an alternating chip pattern[1] with a length of 8, 16, 32, or 64 bits. Throughout this thesis, the term spreading factor will be used to refer to the length of the FSC. One chip pattern is sent for each bit of the transmit stream. The chip pattern is XORed with each transmitted bit.

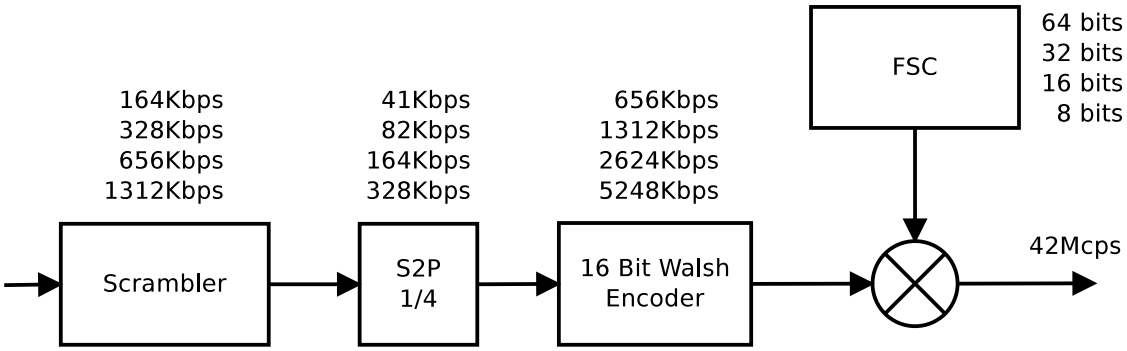

Figure 2.2.: FSDT modulation for IEEE 802.16.6 HBC [2]

Figure. 2.2 shows a graphical representation of FSDT, as well as the 16 bit Walsh code stage for error correction. The chip rate is held constant at 42Mcps, so that different data dates may be selected according to the choice of FSC. Lower data rates require longer FSC sequences, improving the BER at the receiver. This is discussed in Section. 3.1.5.

From the receiver's point of view, the transmitted chips appear as a binary phase shift keyed (BPSK) signal, with a constant carrier frequency of 21MHz.

2.2.2. Packet Structure

The HBC packet structure is shown in Figure. 2.3. A preamble is provided for receiver synchronisation, followed by a start frame delimiter (SFD) The packet header

[1]The FSC of length 8 consists of the sequence [0 1 0 1 0 1 0 1]. This sequence is repeated for longer FSCs.

contains the payload size, the data rate, scrambler seed as well as optional pilot and burst information. The header also contains an 8-bit cyclic redundancy check (CRC). The CRC generator polynomial, $G(x) = x^8 + x^7 + x^3 + x^2 + 1$ is stipulated by the standard [2].

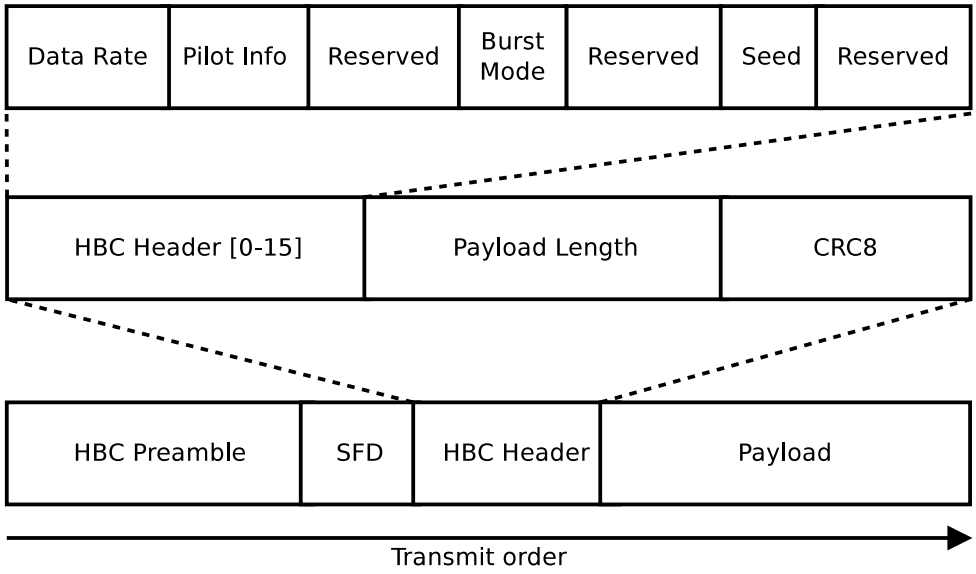

Figure 2.3.: HBC packet structure [2]

The transmitter architecture diagram from page 237 of IEEE 802.15.6 (figure 159) is reproduced here as Figure. 2.4. It shows the main components of the transmitter and the order that they are combined to create a single transmitted packet. A packet is generated by applying each of the modules of Figure. 2.4 top down, starting with the preamble generator. The preamble and SFD/RI are constant, while the header must be updated for each packet. RI, or rate indicator in Figure. 2.4 is referring to the method used to indicate the current spreading factor to the receiver. The details of this mechanism are described in Section. 2.2.2.2. The packet payload is loaded in from the TX FIFO, into the scrambler module.

The receiver must track the transmitter through each state, requiring detection of the preamble and SFD/RI. A state machine in the receiver is responsible for this tracking and is described in Section. 3.1.2.

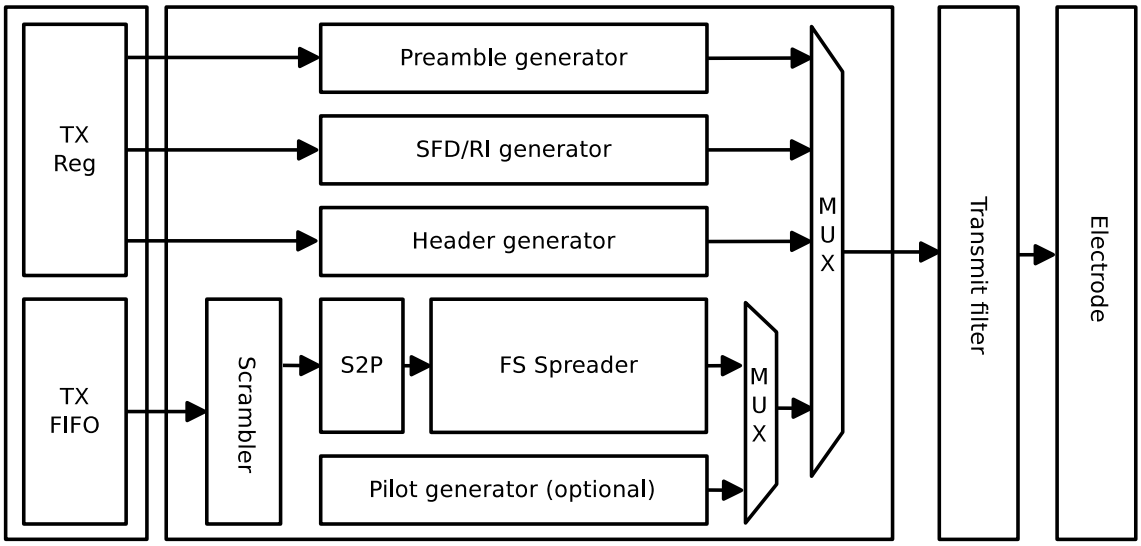

Figure 2.4.: Transmitter architecture [2]

2.2.2.1. Preamble

The preamble consists of a known sequence that allows the receiver to synchronise with the transmitted data stream. This is also known as a comma. IEEE 802.15.6 stipulates a 64 bit preamble based on a Gold code.

According to the standard, the preamble has been derived from a 10 stage Gold code generator. IEEE 802.15.6 provides the initialisation vector as well as an algorithm for generating a Gold sequence for the preamble. In addition to this, a 64 bit sequence is presented in table 76 on page 238 of the standard for implementation by the transmitter.

It is unclear how the 64 bits presented in the standard have been selected from the $2^n - 1$, or 1023 bits an n stage Gold sequence yields. The standard states "the code set is a kind of truncated sequence, which (has been) selected (from) the Gold code sequence" [2]. Reproducing the original 1023 bit sequence from the algorithm presented in the standard does not result in a sequence that may be truncated in any way to match table 76.

Since it is not possible to reproduce table 76 of the standard, a comparison of the

required sequence and the presented Gold code algorithm has been performed. The autocorrelation function of a discrete sequence c is given by:

$$R_c(m) = \sum_{n=1}^{L} c_n c_{n+m} \tag{2.1}$$

Where L is the period of the sequence c and c_k is the kth element of c. The phase difference m is zero when the receiver is synchronised.

The left plot of Figure. 2.5 shows the autocorrelation function with respect to phase difference for the sequence provided in table 76 (in bold) and a random selection of 64 bit sequences from the Gold code algorithm provided by IEEE 802.15.6. The chosen sequence shows the desirable property of low autocorrelation for small phase shifts, which helps with receiver synchronisation as discussed in Section. 3.1.6.

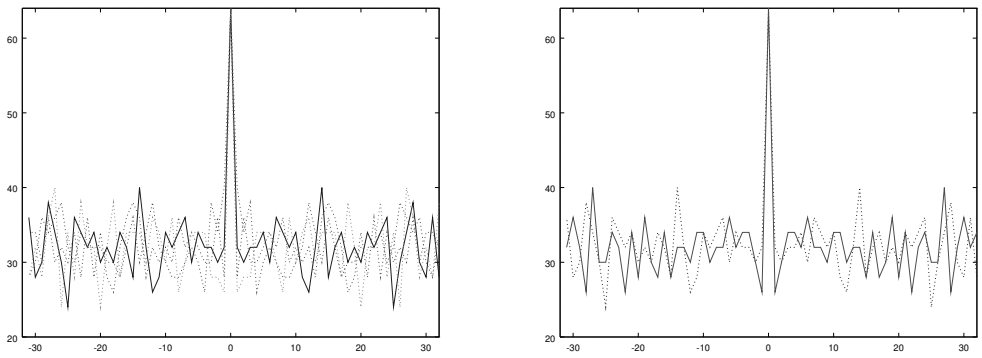

Figure 2.5.: Autocorrelation plots for preamble and start frame delimiter

The 64-bit preamble sequence is repeated four times. The sequence data provided by IEEE 802.15.6 in table 76 has been hard coded into both the transmitter and receiver. The preamble is always modulated using a spreading factor of 8. The Walsh code stage shown in Figure. 2.2 is skipped during the preamble.

2.2.2.2. Start frame delimiter and rate indicator

Following the preamble in the HBC packet is the start frame delimiter, or SFD. This is another 64-bit sequence that is also based on a Gold code using the same algorithm as the preamble with a different initialisation vector. The SFD is always modulated using a spreading factor of 8, without the use of the Walsh code stage, consistent with the preamble.

Similar to the preamble, the standard does not give any insight into how the SFD was selected from the 1023-bit Gold code sequence.

The SFD has a dual purpose, it simultaneously marks the location of the packet header and it indicates to the receiver the spreading factor that is to be applied to the rest of the packet.

The spreading factor is encoded as a delay between the last preamble and the SFD. The SFD is followed by padding so that the entire SFD plus spreading factor information is always constant at $(64+12)8 = 608$ chips. The delay and padding scheme is depicted in Figure. 2.6.

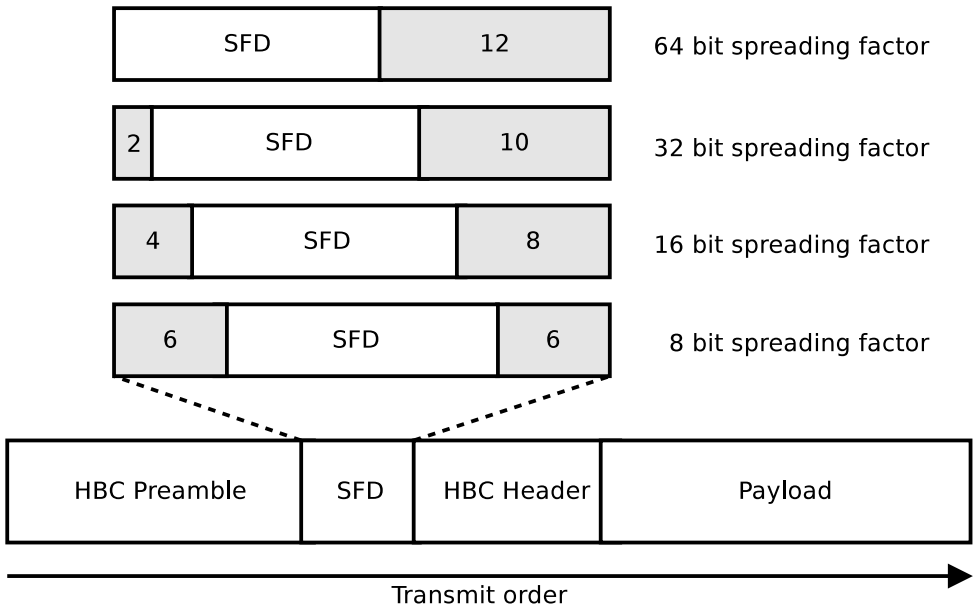

Figure 2.6.: Spreading factor indication using SFD delay

The SFD bit sequence is provided in table 79 of IEEE 802.15.6 and has been hard coded into both the transmitter and receiver. The right hand plot of Figure. 2.5 shows the autocorrelation of the SFD with respect to the phase shift, with the preamble autocorrelation function also shown as dotted for reference. The SFD has low autocorrelation for $n = \pm 1$, which is desirable for accurate detection of spreading factor.

2.2.2.3. Header and payload

The header and payload follow the SFD and any padding chips. Both the header and the packet are modulated using the spreading factor determined by the delay between the preamble and the SFD as described in the previous section. The Walsh code stage shown in Figure. 2.2 is employed for modulation of both the header and the payload.

2.2.3. Scrambler

The presence of long chip sequences without phase changes in the transmit stream may increase the chances of loss of synchronisation between the transmitter and the receiver. This is likely to occur when transmitting data with long runs of zeros or ones. Such sparse data is first whitened by a scrambler stage before transmission to prevent this.

IEEE 802.15.6 describes the implementation of the scrambler based on a linear feedback shift register (LFSR). The LFSR has 32 stages with the input to the first stage given by the recursive equation:

$$Z_1(i+1) = Z_{11}(i) + Z_{31}(i) + Z_{32}(i) \quad \text{and} \quad Z_{n+1}(i+1) = Z_n(i)$$

where Z_n is the nth stage of the shift register at the ith iteration, and $n \in [1, 32]$.

The data stream is XORed with the output of the scrambler before being mapped to a Walsh code.

The algorithm presented in IEEE 802.15.6 does not result in the familiar maximum length sequence, or m-sequence. A simulation found the period of Z to be 1073741820 which is well short of the period of an n stage m-sequence which is given by $2^n - 1$. It is interesting to note that IEEE 802.15.6 also specifies the scrambler polynomial as $P(z) = Z^{32} + Z^{31} + Z^{11} + 1$. Proakis and Salehi [14] describe a similar LFSR configuration that follows the recursive equation:

$$Z_{32}(i+1) = Z_1(i) + Z_{11}(i) + Z_{31}(i) + Z_{32}(i) \quad \text{and} \quad Z_n(i) = Z_{n+1}(i)$$

The same simulation found that this LFSR has a period of $2^n - 1$ and is thus an m-sequence. It is possible that this is merely coincidence, although the use of the same coefficients in reverse order raises the possibility of a typographical error in the standard. Unfortunately attempts to contact the standards mailing list have been unsucessful.

Given that the algorithm presented in IEEE 802.15.6 only iterates through a subset (approximately 1/4) of the $2^n - 1$ available shift register states, it is also something of an anomaly that the two scrambler seeds provided by the standard, exist within the same subset. The standard does not elaborate on the strategy behind the choice of seed values.

Given that the scrambler is reset at the beginning of each packet transmission, and that the maximum transmission length is 255 bytes, the length of the sequence is not important in practise. In order to provide IEEE 802.15.6 compliance, the algorithm presented in the standard is used by the transceiver.

2.2.4. Error correction

As shown in Figure. 2.2, scrambled data is passed through a serial to parallel converter (S2P), that maps each four bits of the data stream to a 16-bit Walsh code. There are 16 Walsh codes, created from the rows of a 16x16 Hadamard matrix. The mapping of symbols to Walsh codes is presented in table 85 of IEEE 802.15.6.

The minimum distance of a code c is given by [14]:

$$d_{min} = \min_{i \neq j} d(c_i, c_j)$$

Where c_i is the ith code word of c. For the 16x16 Hadamard matrix, d_{min} is 8. The error correction capability of a code when using a hard decision decoder is given by:

$$e_c = \frac{d_{min} - 1}{2}$$

This allows for correct symbol detection where the number of bit errors is less than or equal to 3.

Using the $[n, k, d]$ notation, where n is the codeword length, k is the symbol length and d is the minimum distance between any two codewords, the Walsh code scheme presented in IEEE 802.15.6 is written as $[16, 4, 8]$. It should be noted that this code is not length optimal in the sense that another code exists that provides higher throughput with the same minimum distance [15]. The optimal $[16, 5, 8]$ code may be generated by finding all remaining linear combinations of the codewords presented in IEEE 802.15.6 table 85.

The standard does not elaborate on the motivation behind the reduced throughput of the Walsh code scheme. It is reasonable to assert that transceiver design may be simplified though the use of a symbol size that a factor of 8 for a byte oriented channel. Possible alternative optimised codes include $[12, 4, 6]$, or $[8, 4, 4]$ both offering increased throughput, but a lower error recovery rate. For the purposes of comparison, the throughput of a code is defined as k/n and the error recovery rate as $\lfloor \frac{d-1}{2} \rfloor / n$.

Increasing the symbol size to 8 greatly increases the memory requirements of the transmitter and the complexity of the receiver, with minor improvements to throughput and bit error rate recovery. Bouyukliev *et al.* presents all optimised codes with symbol size 8 in [16]. $[30, 8, 14]$ and $[26, 8, 12]$ are the only two codes that are

able to simultaneously offer both improved throughput and error recovery rate. A comparison of all of the optimal[2] codes of dimension 4 and 8 is shown in Figure. 2.7

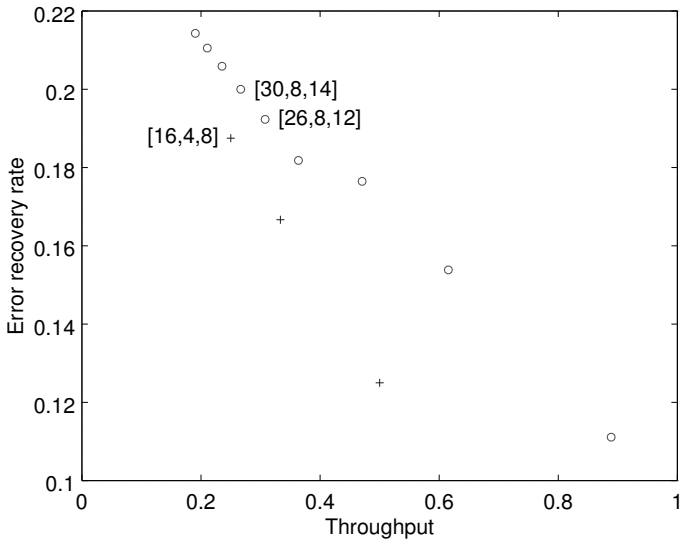

Figure 2.7.: Comparison of binary linear codes of dimension 4(+) and 8(∘)

2.2.5. Implications of IEEE 802.15.6 for transceiver design

This thesis aims to implement the standard as written, however as a research platform, improvements to the standard may be studied through modification of the transceiver. Having a complete hardware description for both the existing standard any any modifications allows for quantitative measurement of potential improvements.

The modular architecture of the transceiver is described in detail in the remaining sections. As an example, the error correction component of the receiver is implemented as the hardware description language (HDL) design unit `walsh_decoder.vhd`. This may be replaced with another module implementing a $[12, 4, 6]$ code without modification of the remaining modules (with the exception of a redefinition of the

[2]Optimal in the sense that no code exists with greater throughput and the same minimum distance.

`walsh_code_t` type). The two transceiver implementations may then be directly compared either in simulation using the complete analogue and digital simulation environment developed for this project, or in hardware using FPGA technology.

Possible improvements of the standard are further discussed in the conclusion of this thesis.

2.3. Existing transceiver architectures

Table 2.2.: History of HBC transceivers

Author	Year	Coupling Amplitude	Carrier Frequency	Modulation Technique	Data Rate	Overall Power Consumption
Zimmerman	1995	30V	330KHz	OOK/DSSS	2.4Kb/s	400mW
Partridge	2001	22V	160KHz	FSK	38.4Kb/s	Not reported
Hachisuka	2005	1V	10.7MHz	FSK	9.6Kb/s	Not reported
S. J. Song	2007	1V	1~200 MHz	PPM + DSSS	2Mb/s	2.4mW
H. Park	2010	Not Reported	32MHz	FSBT	2Mb/s	Not Reported
Y. T. Lin	2011	0.5V	200MHz	OOK	2Mb/s	4.535mW
R. Xu	2012	0.4V	20~100MHz	BPSK/QPSK	10Mb/s	Not reported
J.Bae	2012	1V	40~120MHz	Double-FSK	10Mb/s	4.4mW
C. H. Hyoung	2012	3.3V	16MHz	FSDT	2Mb/s	194mW
K. Shikada	2012	4V	30~50MHz	Impulse Radio	1.2Mb/s	Not reported
H. Cho	2014	Not reported	21MHz	802.15.6	1.3Mbps	5.16mW
H. Park	2015	Not Reported	21,32MHz	FSDT	10Mbps	23mW
Y. L. Tsou	2015	Receiver only	21MHz	802.15.6	1.3Mbps	1.74mW

Seyedi *et al.* [11] conducted a comprehensive survey of previously demonstrated HBC transceivers from 1995 to 2007, starting with Zimmerman's implementation [1]. The survey conducted by Seyedi included a table of existing transceiver implementations that has been reproduced as Table. 2.2 and updated to show recent

developments. Seyedi also included transceiver implementations based on galvanic coupling, however this review focuses on capacitive coupling only.

The first HBC transceiver in literature was presented by Zimmerman in [1]. The paper presented both a transceiver platform and the concept of capacitive coupling, using the surrounding environment as the return path of the communications channel. Zimmerman's transceiver used either on-off keying (OOK) or direct sequence spread spectrum (DSSS), requiring an integrator at the receiver sampled by an analogue to digital converter. The scheme was capable of communicating at 2.4Kb/s.

Another example of an early HBC transceiver was presented by Partridge [17] using frequency shift keying (FSK) with a centre frequency of 160KHz. The receiver was based on the Exar XR2211 FSK demodulator integrated circuit and was capable of sustaining a data rate of 38.4Kb/s. Hachisuka also described a FSK based HBC platform operating at 10.7MHz achieving a data rate of 9.6Kb/s [18]. The paper does not describe the operation of the receiver in detail.

Song *et al.* present a more sophisticated transceiver platform using DSSS followed by pulse position modulation (PPM) capable of 2Mb/s [19]. The design is notable as it is the first to present a packet structure applicable to HBC communications, in particular inclusion of a preamble and identification word based on a Gold code. The identification word provided the advantage of allowing a receiver to ignore packets not matching a predefined address without demodulating the entire packet, reducing packet reception energy requirements.

Frequency selective baseband transmission (FSBT)is first demonstrated by Park *et al.* in 2010 [20]. The modulation scheme proposed in the paper is almost identical with FSDT as described by IEEE 802.15.6, with a spreading factor of 2, 64bit Walsh codes and a chip rate of 64Mcps. From the same research group at ETRI, Hyoung *et al. [21]* proposed an alternative modulation scheme that would be eventually adopted and incorporated into IEEE 802.15.6. Their prototype transceiver operated at 32 million chips per second (Mcps). The proposed modulation technique was labelled frequency selective digital transmission (FSDT) and consists of 3 parts: A serial to parallel converter, a Walsh code modulator and a frequency spreader.

Lin [22] describes a transceiver using OOK modulation focussed on minimum power and chip area. The receiver described in the paper consists of down conversion to baseband, and hard decision decoding. A system on a chip for health monitoring is described, with the hard decision decoder directly connected to an embedded microcontroller with an analogue to digital converter peripheral. The scheme does not describe a general packet based networking architecture, however it does provide a minimalistic sensor network solution without error correction, at a data rate of 2Mb/s.

One of the more interesting alternatives to FSDT for HBC is presented by Bae *et al.* [12] using a technique referred to as double frequency shift keying, or double FSK. Double FSK is a spread spectrum technique that offers frequency domain multiple access without the complication and power consumption of DSSS, or orthogonal frequency division multiplexing (OFDM) and is described in detail by Gerrits *et al.* [23]. An ultra wideband (UWB) frequency modulation (FM) modulator spreads several sub-carriers (using narrowband FSK) over a desired spectrum. The receiver first demodulates the ultra-wideband FM signal before filtering and demodulating each sub-carrier. It is possible to provide multiple access to several users by assigning separate sub-carriers, similar to OFDM. Furthermore sub-carriers can be assigned various bandwidths allowing a trade-off between the maximum possible number of users or throughput. Gerrits shows that maximum bandwidth utilisation requires the use of triangular wave oscillators, while Bae recognises that the low power requirements of HBC transceivers precludes the use of digital to analogue converters. The transceiver presented in [12] solves this problem by using five discrete carrier frequencies from 40MHz to 120MHz that are switched by the sub-carriers. All of the discrete frequencies are generated by dividing a common 1.2GHz clock.

Each sub carrier frequency is user selectable resulting in 104 different configurations of the transceiver, ranging from a single user operating at 10Mb/s with a spreading gain of 6dB to 15 users operating at 1.25Kb/s with a spreading gain of 36dB. While the transceiver offers a wide range of configurations for a large number of use cases, it suffers from the wideband nature of the transmitted signal, which for electromagnetic compatability reasons has been strictly limited by IEEE 802.15.6.

Shikada proposed an impulse radio (IR) based system [24] in 2012, that modulated the transmitted data stream with wideband pulses. The main motivation for the use of IR was to decrease power density by spreading the power spectrum. While the IR transceiver was capable of 1.2Mbps communications it is also unlikely to see mainstream adoption due to the wideband nature of the transmitted signal.

The transceiver presented by Xu [25] performed baseband processing using an external processor, connected to the receiver using an optical link. The team were able to achieve 10Mb/s communications using QPSK modulation, however the focus of the transceiver design was quantification of a channel model, rather than development of a transceiver for general use.

The remaining published transceivers have all been based on the IEEE 802.15.6 standard, or non-compliant variations of the same FSDT modulation scheme. An improvement to Hyoung's transceiver was proposed by Kang *et al.* [26] to increase the maximum data rate to 3.9375MBps using three separate data streams, all modulated at 42Mcps and then separately decoded at the receiver, selecting Walsh codes based on hamming distance. This method is similar to multiple access methods using orthogonal coding, for example code division multiple access (CDMA). By altering the protocol, Kang *et al.* were able to improve the bit rate, without significantly altering their overall receiver architecture.

The most recent publications from the ETRI group are [3] and [27]. These papers describe an IEEE 802.15.6 standard compliant transceiver and a non-compliant transceiver respectively. The non-compliant transceiver presented by Park *et al.* increases the chip rate to 64Mcps and removes the FSC stage. This is the same architecture presented previously by Park in [20].

In [3] Cho presents a standard compliant design implemented using a 130nm complimentary metal-oxide semiconductor (CMOS) process that combines both the analogue electronics and MAC layer on the same die, similar to [28], a receiver only design. The transmit mask filter is implemented using an on-chip 9th order high pass filter, followed by a 6th order low pass filter.

The receiver operates in two configurations, for high signal to noise (SNR) ratios, the

receiver detects phase transitions by sampling the input stream at the chip rate and detecting two consecutive zeros, or two consecutive ones. This method differs from the method presented in this thesis. The advantage of the phase detection method employed for this implementation is presented in more detail in Section. 3.1.5.

For low SNRs, Cho's receiver uses an on chip analogue delay locked loop (DLL) based Costas loop architecture, detecting both the in-phase and quadrature components of the input signal and achieving receiver synchronisation by altering the local oscillator delay to minimise the quadrature component. Once receiver synchronisation is complete, the receiver disables the quadrature path and demodulates the in-phase component only, in order to decrease receiver power consumption. Tsau's receiver only includes the in-phase component, the paper does not describe how phase coherence is achieved [28]. While these direct conversion architectures are well suited to IEEE 802.15.6 demodulation, implementation is impossible using digital only FPGA technology.

This thesis aims to present a re-configurable, modular receiver architecture for IEEE 802.15.6 using only readily available FPGA technology, and is therefore limited to digital demodulation techniques only. While an ASIC implementation such as [3] allows for improved performance, it does not allow for re-configuration and has limited use as a research platform. This research aims to fill the gap by providing hardware descriptions for a software-defined receiver architecture that may be freely modified for research purposes.

3. Transceiver digital core

The majority of the transceiver implementation consists of digital logic responsible for the modulation, synchronisation and demodulation of IEEE 802.15.6 compliant packets. Other components are required for operation of the transceiver, such as a CPU for initialisation and communications with a host interface, reset logic, memory interface logic and clock domain logic.

This chapter may be read as a companion to the hardware description source files embedded into this document as `hardware_desc.tar.xz`. This repository may be extracted using the *7-zip* or *xz* utilities.

3.1. Components

IEEE 802.15.6 describes the transmitter in detail and leaves the receiver implementation unspecified. The transmitter specification can be divided into five main components: (1) The preamble generator, (2) start frame delimiter/rate indicator (SFD/RI) generator, (3) packet header, (4) parallel to serial conversion and (5) frequency selective spreading.

The prototype transmitter constructs the preamble and packet header in software, while the other components are implemented using FPGA logic.

The prototype receiver architecture can be broken into 7 stages: (1) Data synchronisation, (2) phase detection, (3) comma detection, (4) phase alignment, (5) rate detection, (6) Walsh code correlation and (7) descrambling.

The transceiver architecture is modular, meaning that each major component is implemented as a separate hardware description design unit. As a research platform, a

modular design allows for modifications to the standard, or the transceiver architecture with relative ease. Another advantage of modularity is that components that require specific FPGA primitives that are unique to the Spartan 6 platform can be separated from generic components.

All of the hardware descriptions for each receiver module are listed in Table. 3.1. Other modules are necessary for the complete implementation of the embedded system. The hardware descriptions are all contained within a single repository, along with the simulation framework, preprocessor macros, library functions and Makefiles. With the exception of any Xilinx IP cores, and the SPI cores, all modules (including all modules listed within Table. 3.1) have been developed as part of this project.

Table 3.1.: Receiver modules

Function	**Module name**
Receiver top level module	`hbc_rx.vhd`
Data to clock synchronisation	`data_synchroniser.vhd`
Phase alignment	`phase_align.vhd`
Comma detection	`hamming_lut.vhd`
Scrambler/Descrambler	`scrambler.vhd`
State machine & integration	`serial_to_parallel.vhd`
Walsh code correlator	`walsh_decoder.vhd` & `walsh_enc_lut.vhd`
FIFO control	`rx_fifo_interface.vhd`

3.1.1. Design approach

The digital component of the transceiver is implemented as an embedded system, using a Xilinx Microblaze CPU embedded into a Xilinx Spartan 6 FPGA. In order to enable rapid prototyping, the IEEE 802.15.6 protocol is implemented in software where possible.

The embedded software stack *hbc_mac* is responsible for packet framing, FIFO control, transceiver configuration and scrambling/descrambling. All other aspects of the protocol must be implemented in hardware to achieve real time performance.

The transmitter components are relatively straight forward and not described in detail here. The modules `modulator.vhd` and `parallel_to_serial.vhd` describe the core of the transmitter section. Generation of the preamble, SFD and packet header are handled by the embedded *hbc_mac* software within the `build_tx.c` module.

Since the implementation of the receiver is not described by IEEE 802.15.6, the receiver architecture will be described in detail throughout the remainder of this section.

Each module is the result of the following development approach:

1. First implement the module in software on a desktop computer and confirm correct operation.
2. Port the module to the Microblaze embedded CPU. If performance is satisfactory, the module is complete. If not goto step 3.
3. Create a high level hardware description of the module. If performance is satisfactory, the module is complete. If not goto step 4.
4. Create a low level hardware description of the module, utilising available FPGA primitives as required.

The disadvantage of completing step 4 is increased complexity, development time and decreased portability, especially if FPGA primitives are required. In practice it was found that only the hamming distance logic required completion to step 4. All other modules have been implemented to either step 2 or 3.

Modules developed to step 2 or 3 may be easily ported to other architectures. Any ANSI C compiler or IEEE 1076-1993 compliant VHDL synthesiser may be used to implement the hardware descriptions or source code without modification.

3.1.2. State machine

In order to track the state of the transmitter shown in Section. 2.2.2 a state machine has been implemented in the `serial_to_parallel.vhd` module. As well as tracking transmitter state, the state machine shown in Figure. 3.1 is responsible for initialisation and control of each of modules described throughout this chapter. As each packet is detected, the state machine transitions through 6 stages, consisting of alignment (2 stages), preamble detection, SFD detection, demodulation and packet end detection.

The alignment stages use the `comma_found` signal provided by the phase alignment module described in Section. 3.1.7 to synchronise both the state machine and the phase detection stages. After the initial detection of the preamble, the `comma_found` signal is de-asserted once alignment is complete. A second preamble must be detected before the receiver moves on to RI decoding in the SFD state. This requirement allows the phase alignment stage to possibly invert the data stream if it has detected inverted data.

Once the spreading factor has been detected, the `sfd_finished` signal is asserted and the receiver begins demodulating using the Walsh code correlator described in Section. 3.1.9.

Two methods are used to determine the end of a packet. The payload length information in the packet header is compared against the number of received bytes, once all bytes have been demodulated, the `pkt_end` is asserted. In addition to this packet end detection is achieved by detecting the absence of any carrier for a period of eight chips. This period may be modified by altering the `PKT_END_THRESH` constant. Detection of carrier loss is required in the case that either an incomplete packet is transmitted, there is an error in the transmitted packet header, or the communications channel has faded. The packet end state lasts for a single cycle, during which all receiver modules are reset and an interrupt is issued to the embedded CPU. The receive FIFO is large enough to accommodate an entire packet, so the CPU may begin reading once the packet end interrupt request (IRQ) has been asserted. Alternatively there is another interrupt assigned to the almost full

condition of the FIFO.

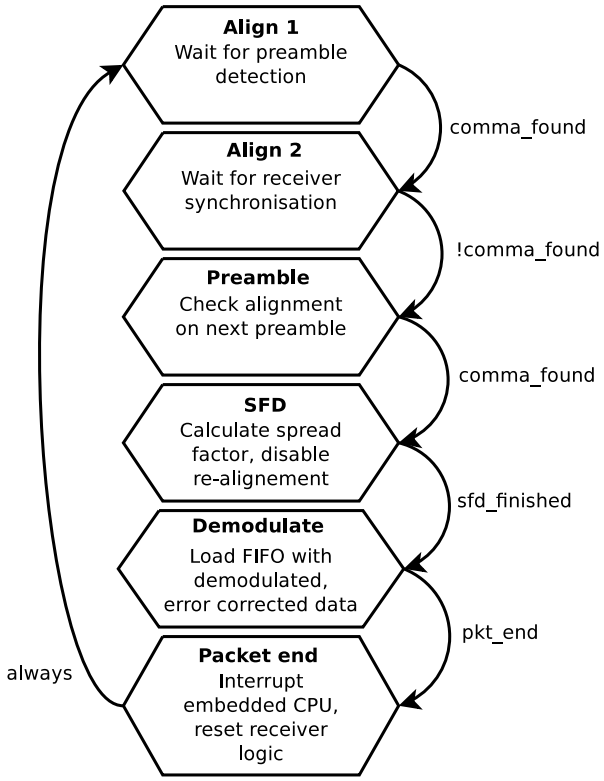

Figure 3.1.: Receiver state machine

3.1.3. Data flow

A data flow diagram of the receiver is shown in Figure. 3.2. The `data_in` signal represents the input data stream after re-synchronisation with the receiver clock domain as described in Section. 3.1.4. During the align 1, align 2 and preamble states, the `phase_align` module constantly searches for the best receiver alignment before passing the data stream on to the `detect_phase`. The details of this alignment process are described in Section. 3.1.7.

The `demodulator` module does serial to parallel conversion of the input stream for detection of the SFD when in the SFD state and Walsh code detection during the demodulate state.

Each decoded symbol at the output of the `walsh_decoder` is loaded into a shift register by the `decode_word` module before being loaded into the receiver FIFO, ready for access by the embedded CPU.

The `choose_rate` module is responsible for detection of the packet spreading factor, using the rate indicator method described in Section. 3.1.8. The output of this module `r_sf` is propagated to all modules involved in demodulation. During all states except for demodulate, `r_sf` is set to 8, as required by IEEE 802.15.6 and described in Section. 2.2.2.1 and Section. 2.2.2.2.

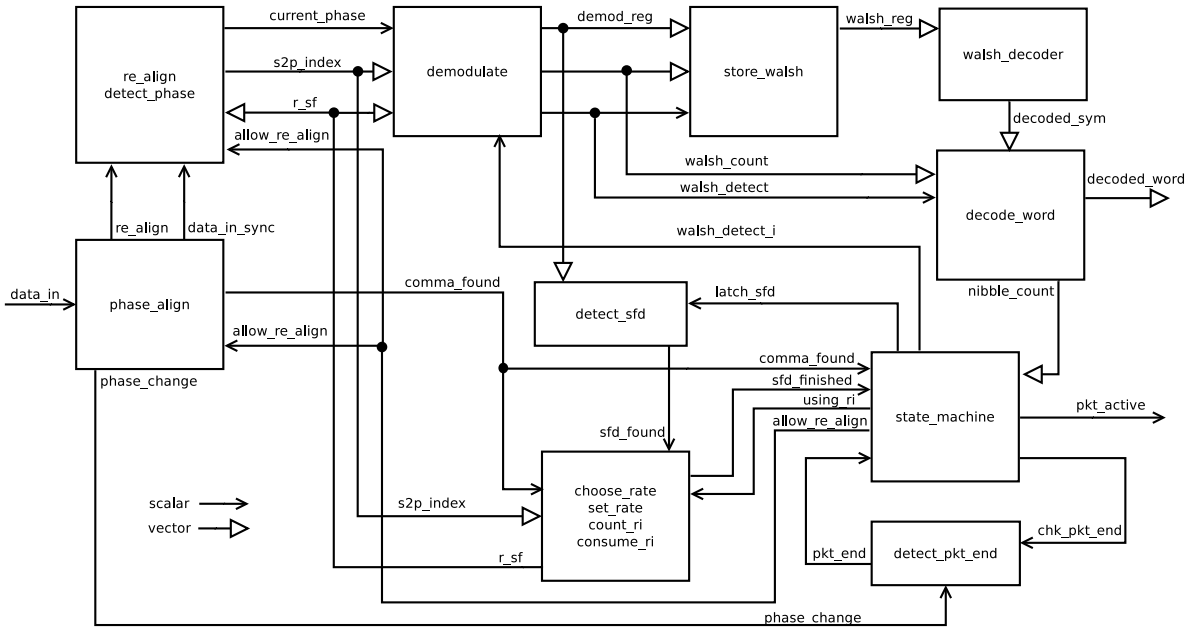

Figure 3.2.: Receiver architecture

3.1.4. Data synchronisation

The data stream arriving at the receiver will not necessarily be aligned to the receiver's clock. It is necessary for the receiver to realign the data to its own clock before any demodulation can begin. It is the responsibility of the data synchronisation module to eliminate meta-stability and to provide a clock aligned signal to

the later stages of the receiver. The data synchronisation module deign closely follows the implementation suggested by Xilinx in their application note 225, “Data to Clock Phase Alignment” [29], with some modifications including increasing the input buffer size to accommodate larger clock drift tolerances between the transmitter and receiver.

The module uses a four phase clock, providing four times oversampling of the input stream. Data that is sampled on the 270° clock phase must be re-sampled at 90° rather than 0° in order to improve metastability performance. This results in the 12 flip-flop configuration shown in Figure. 3.3.

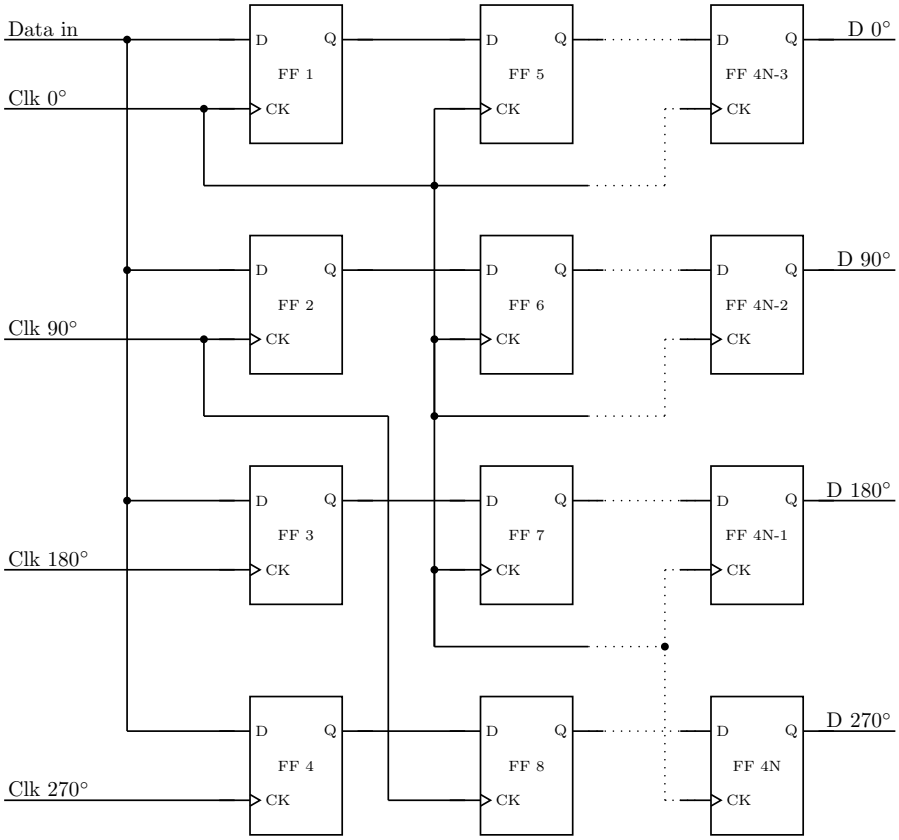

Figure 3.3.: Data synchroniser logic

With the data re-sampled back onto the 0° clock, it is necessary to select the most appropriate stream from the four alternatives: 0°, 90°, 180° or 270°. The `select_phase`

process chooses the best phase for data sampling by searching for either positive or negative going transitions across the current and previous data samples for each phase. This decision process is illustrated in Figure. 3.4, where an E denotes a detected edge and N represents no change.

The decision logic is:

$$
\begin{aligned}
USE_0 &= E_0 E_{90} N_{180} N_{270} \\
USE_{90} &= E_0 E_{90} E_{180} N_{270} \\
USE_{180} &= E_0 E_{90} E_{180} E_{270} \\
USE_{270} &= E_0 N_{90} N_{180} N_{270}
\end{aligned}
$$

The `hold_phase` process maintains the current data stream selection in the absence of any edges.

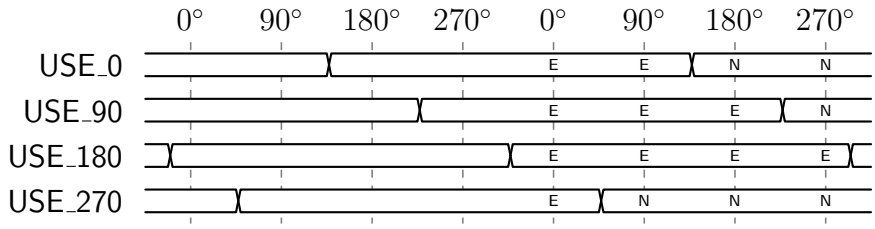

Figure 3.4.: Edge criteria for phase selection in data synchroniser

When the transmitter clock frequency is slightly lower than the receiver frequency, the selected phase will steadily transition in the direction 180°, 270°, 0°. With each transition, the receiver is effectively decreasing the delay between the data sample time and the succeeding demodulation stages. Each time the selected phase passes from 270° to 0°, it is necessary to decrease the delay by one period. The configuration shown in Figure. 3.3 is capable of doing this by introducing a N stages of flip-flops. On reset, the centre flip-flop at stage $N/2$ is selected as the source for the succeeding stages. Each time the required delay wraps from 270° to 0°, or vice

versa, the selected data source is either incremented or decremented to $N/2+1$ or $N/2-1$.

A full IEEE 802.15.6 frame requires a transmission time of t_p, where:

$$t_p = t_c \left(4n_p + n_{sfd} + \frac{8}{R_c} s(l_h + l)\right) \tag{3.1}$$

and t_c is the duration of 1 chip, n_p is the number of chips in the preamble, n_{ri} is the number of chips in the rate indicator, R_c is the code rate for the Walsh code, s is the spreading factor, l is the length of data to be transmitted, and l_h is the length of the packet header. IEEE 802.15.6 stipulates that: $t_c = 23.81ns$, $n_p = 512$, $n_{sfd} = 608$, $R_c = \frac{4}{16}$ and $l_h = 4$. The spreading factor and packet data length are user selectable such that $s \in \{8, 16, 32, 64\}$ and $0 \leq l \leq 255$.

The minimum and maximum packet durations are therefore:

$$t_{p_{min}} = 87.6\mu s, \quad t_{p_{max}} = 12.7ms \tag{3.2}$$

The maximum clock drift possible is a function of the size of N, specifically:

$$\frac{N}{2} \frac{t_{c_r}^2}{t_{c_t} - t_{c_r}} > t_{p_{max}} \tag{3.3}$$

must hold, where t_{c_r} is the chip period of the receiver and t_{c_t} is the chip period of the transmitter. Expressing the difference in chip rates such that $t_{c_t} = t_{c_r} + \delta$:

$$|\delta| < \frac{N}{2} \frac{t_{c_r}^2}{t_{p_{max}}}, \quad or \quad e < \left| \frac{f_r}{f_r \pm \frac{N}{2t_{p_{max}}}} - 1 \right| \tag{3.4}$$

where e is the transceiver crystal oscillator frequency tolerance, such that $f_t = f_r(1 \pm e)$. A 64 stage delay circuit allows for an error of approximately 60ppm, well

within the 20ppm tolerance of the prototype crystal oscillator. A 22 stage delay circuit allows for 20ppm crystal accuracy.

When testing this implementation using actual data sent over a phantom solution as described in the paper "An Empirical Measurement of Signal Attenuation and BER of IEEE 802.15.6 HBC using a phantom solution" [30], it was found that spurious detections were resulting in transitions between 180° and 0°, or 270° and 90°, skipping one phase transition. In these cases it is not possible to determine whether this should be a skip forward or skip backward. As an improvement to the Xilinx proposed module, the prototype data synchroniser keeps track of the direction of the most recent transition and applies it in the case where direction is ambiguous. This was found to greatly improve the likelihood of synchronisation during testing.

Improvements to the Xilinx module are user selectable at implementation time using the preprocessor macros `DOUBLE_SHIFT` and `SLOW_SHIFT`. `DOUBLE_SHIFT` selects the aforementioned direction tracking. `SLOW_SHIFT` inhibits phase selection until each phase has been selected for a predefined number of cycles. `SLOW_SHIFT` greatly improves the synchronisers robustness when receiving a noisy input, by ignoring spurious phase shifts. This has a low pass filter effect on the current phase selection.

3.1.5. Phase detection

The current phase of the incoming data stream is detected by comparing each incoming chip state with the expected chip state. This is only possible once the receiver is synchronised, as described in Section. 3.1.7. A counter is incremented for each matching chip. If the counter exceeds half of the current spreading factor, the phase is interpreted as representing a binary '1'.

A simplified version of the phase sum register transfer level (RTL) logic is shown in Listing 3.1, with the re-synchronisation logic removed for clarity. The original process may be found in *serial_to_parallel.vhd* with the process name: `re_align_proc`.

The summing method allows for bit errors in the incoming stream, as long as the majority of chips agree with the expected phase. Correct detection is possible with

Listing 3.1: Phase detection RTL

```
sum_phase : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
                -- Process must be synchronised with transmit stream
                if sync_index = spreading_factor - 1 then
                        sync_index <= (others => '0');
                        phase_sum <= (others => '0');
                        expected_phase <= '1';
                else
                        if (data_in = expected_phase) then
                                phase_sum <= phase_sum + 1;
                        end if;
                        sync_index <= sync_index + 1;
                        expected_phase <= not expected_phase;
                end if;
        end if;
end process sum_phase;

current_phase <= bool_to_bit(phase_sum >= spreading_factor/2);
```

$\frac{n}{2}-1$ bit errors, where n is the current spreading factor. For larger spreading factors the probability of correct phase detection increases, while the data rate decreases.

Given the probability of error for chip detection P_c, the probability of phase detection error P_b may be expressed as the sum:

$$P_b = \sum_{k=n/2}^{n} \binom{n}{k} P_c^k (1-P_c)^{n-k} \tag{3.5}$$

This is the probability mass function of a binomial random variable [14] summed over the region from $n/2$ to n, since a phase will be incorrectly detected if there are more than $n/2$ chip detection errors.

A plot of equation 3.5 is shown in Figure. 3.5 for the spreading factors allowed by IEEE 802.15.6. The plot demonstrates a substantial improvement in reliability of

communications when higher spreading factors are used.

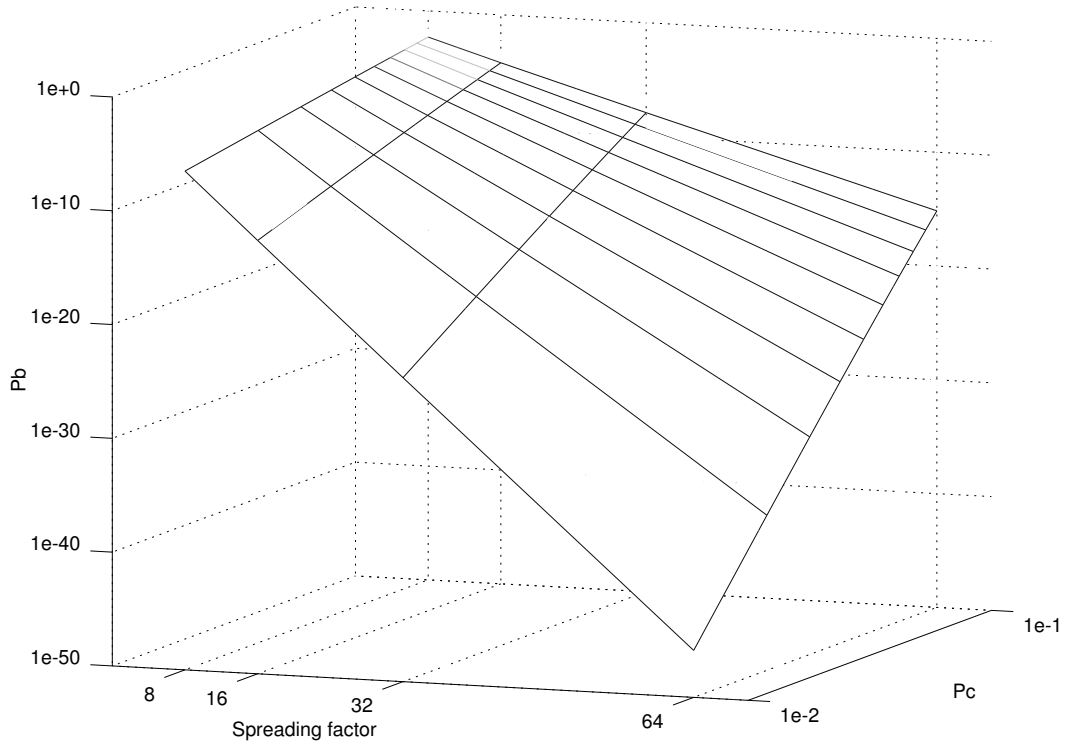

Figure 3.5.: Phase detection error probability P_b as a function of spreading factor and chip error probability P_c

After phase detection, the identified phase is shifted into a register that is used both for comma detection and Walsh code detection. The shift register is 64-bits long to accommodate a full preamble or SFD. The Walsh code detector reads the most recent 16-bits of the same register once the SFD has been detected.

As mentioned in Section. 2.3, another method of detecting phase is to identify two consecutive chips. Cho [3] presents the digital detector logic reproduced in Figure. 3.6. The inputs to the SR latch go positive when both flip flops are either asserted, or de-asserted. This method may be used as an alternative phase detection method and is implemented in the module `cho_demod.vhd`, however it is not possible

to achieve the same BER performance as the counter method described above. This is due to the fact that increasing spreading factor does not decrease the likelihood of phase transition detection, as the detector is only searching for consecutive chips.

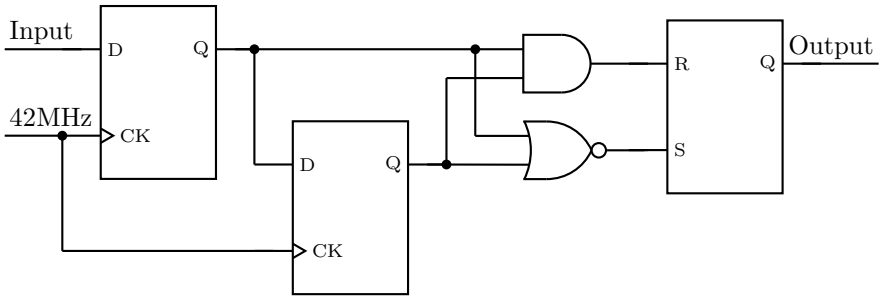

Figure 3.6.: Phase transition detection circuit presented by Cho *et al. [3]*

3.1.6. Comma detection

Each IEEE 802.15.6 packet begins with a 64-bit preamble sequence that is repeated four times in order to facilitate receiver synchronisation. The preamble is modulated at 8 chips per bit resulting in a 512 chip sequence. The receiver recognises the preamble sequences by calculating the Hamming distance between the known preamble sequence and the incoming stream. A 64-bit Hamming distance calculation must be performed within one chip duration (23ns) in order to allow the phase alignment stage to select the best phase.

Listing 3.2 describes a naïve algorithm for calculating the Hamming distance between the input `slv` and `target`. When synthesising Listing 3.2 using Xilinx's XST, the resulting logic contains a combinatorial stage for each iteration of the for loop. This topology is acceptable for small vectors, however for a 64-bit vector, the combinatorial logic requires 64 propagation delays and does not meet the timing constraints required for the receiver of 23ns.

A more efficient topology is suggested by Sklyarov and Skliarova [31] and shown in Figure. 3.7. The design leverages the 6 input lookup table (LUT) a primitive

Listing 3.2: Hamming weight RTL

```
function calc_Hamming(slv, target : std_logic_vector) return natural is
        variable sum : natural := slv'length;
begin
        for i in slv'range loop
                if slv(i) = target(i) then
                        sum := sum - 1;
                end if;
        end loop;
        return sum;
end function calc_Hamming;
```

available in many FPGAs including the Xilinx Spartan 6. The topology is capable of calculating the Hamming weight of a 36 bit input vector in 4 propagation delays.

The Hamming weight of the vector present at the input of LUT_1_2 is multiplied by four by left shifting the inputs to the adders. Similarly the output of LUT_1_1 is multiplied by two. This topology allows for the Hamming weight calculation of a 36-bit vector using only 27 LUTs and two adders. The synthesis of the two adders requires an additional 9 LUTs when synthesised for the Spartan 6 device, requiring a total of 36 LUTs per 36-bit Hamming distance calculator. The low end Spartan 6 device used in the prototype has a total of 5720 LUTs.

Each LUT triplet must be configured for operation as a 6-bit Hamming weight calculator at device power up. The Xilinx tool-chain provides the INIT generic for LUT6 primitives that allows the user to specify the output of each LUT6 for all possible input states. The required INIT string may be generated by simply manually calculating the Hamming weight of all 64 possible input combinations. The code in Listing 3.3 prints the initialisation strings by looping though all possible inputs and calculating the resulting Hamming weight using `calc_Hamming()`, resulting in the following output:

```
INIT_0 = 6996966996696996

INIT_1 = 8117177E177E7EE8
```

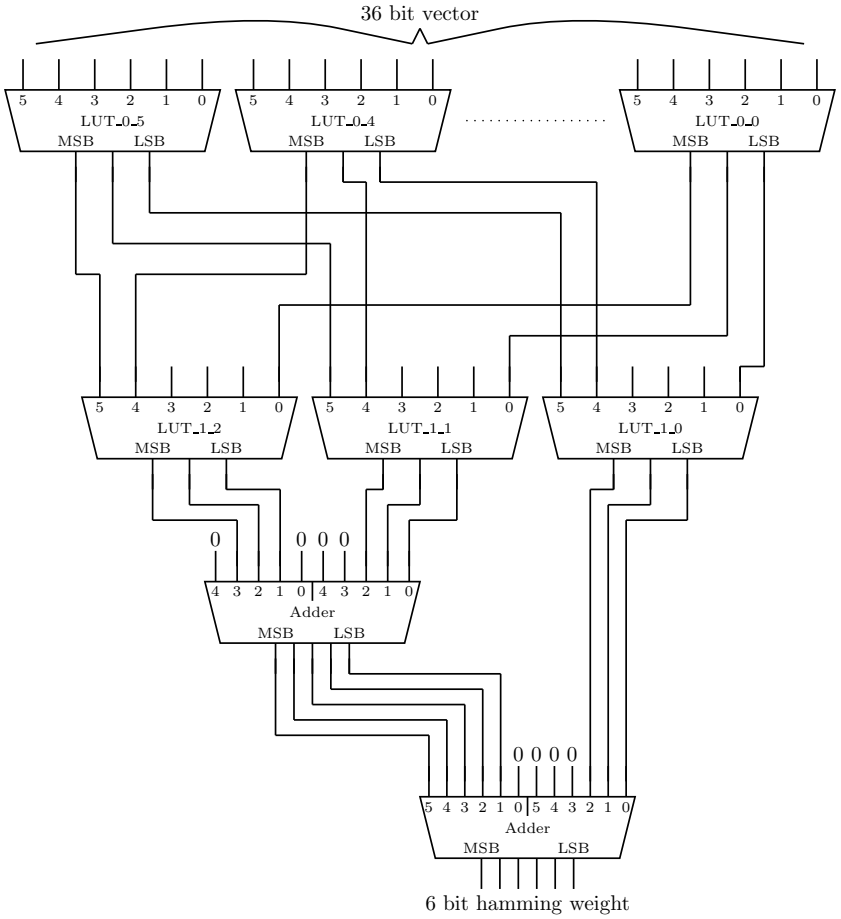

Figure 3.7.: Optimised Hamming weight logic, requiring 4 propagation delays.

```
INIT_2 = FEE8E880E8808000
```

The Hamming distance between an arbitrary 64-bit input vector and a 64-bit target sequence may be calculated by first XORing the two vectors and then passing the result to the Hamming weight logic. The resulting logic is capable of meeting the timing constraints of 23ns on a Spartan 6 device, allowing the following phase alignment stage to recognise synchronisation in a single cycle.

Listing 3.3: Hamming weight LUT configuration

```
#include <stdio.h>
#include <stdint.h>

#define INPUT_WIDTH 6

int calc_Hamming(uint8_t reg) {
    int i;
    int weight = 0;

    for (i = 0; i < INPUT_WIDTH; i++) {
        if ((reg >> i) & 1) weight++;
    }

    return weight;
}

int main (void) {
    /* 6 -> 3 Hamming weight lookup table */
    uint8_t reg;
    uint64_t weight;

    uint64_t init_0, init_1, init_2;

    init_0 = 0;
    init_1 = 0;
    init_2 = 0;

    for (reg = 0; reg < (1 << INPUT_WIDTH); reg++) {
        weight = calc_Hamming(reg);
        init_0 |= (weight & 1) << (reg - 0);
        init_1 |= (weight & 2) << (reg - 1);
        init_2 |= (weight & 4) << (reg - 2);
    }

    printf("INIT_0 = %016lX\n", init_0);
    printf("INIT_1 = %016lX\n", init_1);
    printf("INIT_2 = %016lX\n", init_2);

    return 0;
}
```

3.1.7. Phase alignment

The phase alignment stage takes the re-synchronised data from the previous stage and attempts to determine the location of phase shifts in the transmit stream. A block diagram shown in Figure. 3.8 shows the main components of this stage.

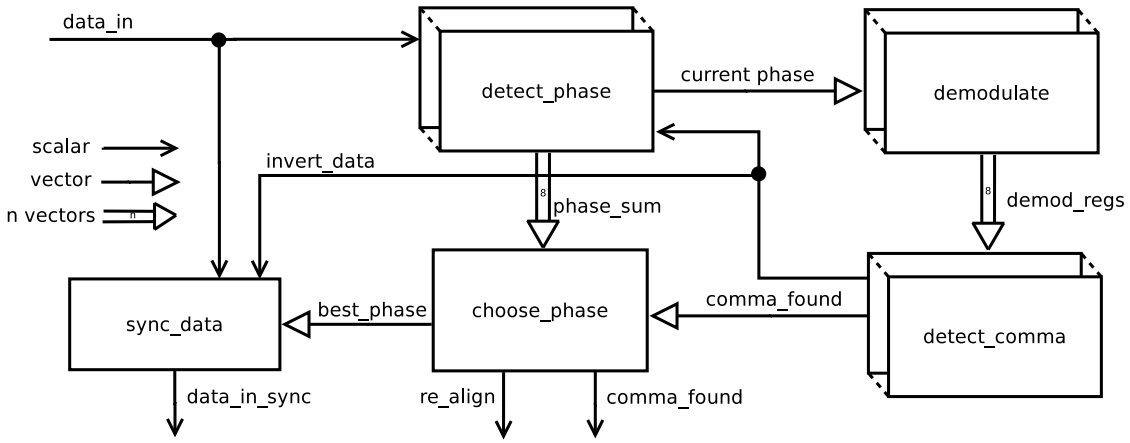

Figure 3.8.: Phase alignment block diagram

IEEE 802.15.6 stipulates that each packet preamble is modulated using a spreading factor of 8 chips per bit. There are therefore 8 possible alignments between the incoming data stream and the receiver state. It is the responsibility of the phase alignment module to select the alignment that results in the lowest Hamming distance between the incoming data stream and the known preamble sequence. If two or more alignments have equal Hamming distance results, the alignment with the highest "phase sum" is selected. The phase sum is calculated by comparing the incoming chip state with the expected chip state.

The phase alignment exploits the low logic, low propagation delay Hamming distance logic architecture mentioned previously to simultaneously detect the Hamming distance for all 8 possible alignments. Using this architecture, the receiver is able to select the best alignment 8 cycles after detecting the presence of the preamble sequence.

Figure. 2.5 shows that the preamble minimum autocorrelation $R_c(m)$, for $m \neq 0$

is 40. To allow for bit errors, the phase alignment stage asserts the comma found signal for any autocorrelation greater than 55. This value was chosen as it allows for simple comparison logic. Checking that $R_c > 55$ is equivalent to checking that the upper 4 bits of R_c are equal to the values 7 or 8.

There are 8 `detect_phase`, `demodulate` and `detect_comma` blocks within the phase alignment module as depicted in Figure. 3.8. The `current_phase` vector contains one element for each possible alignment that is constantly updated by the `detect_phase` block. Serial to parallel conversion is performed by the `demodulate` block, resulting in a vector of vectors, `demod_regs` that may be fed into the 8 separate `detect_comma` blocks.

The `choose_phase` block is responsible for selecting the best alignment based on the Hamming distances produced by the `detect_comma` block, as well as the `phase_sum` vector provided by the `detect_phase` stage.

Another useful application of the phase alignment stage is recognition of inverted data.

The autocorrelation function of a discrete sequence c is given in equation 2.1. The equation may be modified for the autocorrelation between the sequence c and the same sequence inverted, c':

$$R_c(m) = \sum_{n=1}^{L} c_n c_{n+m} = L - \sum_{n=1}^{L} c_n c'_{n+m} \tag{3.6}$$

It can be seen from equation 3.6 that any Hamming distance greater than the threshold $T' = L - T$ is equivalent to a Hamming distance less than the threshold T with the input data sequence inverted.

The `comma_detect` block will assert the `invert_data` signal whenever R_c is less than 8. Similar to the case for $R_c > 55$, it is only necessary to perform a comparison with the upper four bits of R_c and the value 0.

3.1.8. Rate selection

The spreading factor, or rate detection stage is required to decode the spreading factor information encoded as a delay between the last preamble and the SFD as described in Section. 2.2.2.2.

This is achieved using a counter that is reset whenever the preamble is detected and freely increments at the end of each phase detection cycle otherwise. As soon as the SFD is detected, the value in the counter is used to identify the current spreading factor according to the method described in Section. 2.2.2.2.

The spreading factor signal `r_sf` shown in Figure. 3.2 is then propagated to the phase detection and demodulation stages for correct packet interpretation.

Interpretation of the packet header cannot begin until any remaining chips have been consumed, following the SFD. The same counter is also responsible for signalling the beginning of the packet header using the `sfd_finished` signal. The state machine shown in Figure. 3.1 then moves to the demodulate state, enabling the Walsh code detector.

3.1.9. Walsh code correlation

The forward error correction scheme described in Section. 2.2.4 uses a Walsh or Hadamard [16, 4, 8] code located directly after the scrambler in the transmitter architecture.

Detection of the transmitted symbol is achieved by searching for the codeword with the minimum Hamming distance from the received data stream. As the symbol size is small, it is practical to iterate through all 16 possible code words in real time. The time that each Walsh code is present in the demodulator shift register described in Section. 3.1.5 depends on the current spreading factor. In the worst case scenario, there are 128 serial clock cycles available for Walsh code detection.

Figure. 3.9 shows the architecture of the Walsh code detector. `data_in` is a vector representing the most recent 16-bits loaded into the demodulator shift register. The

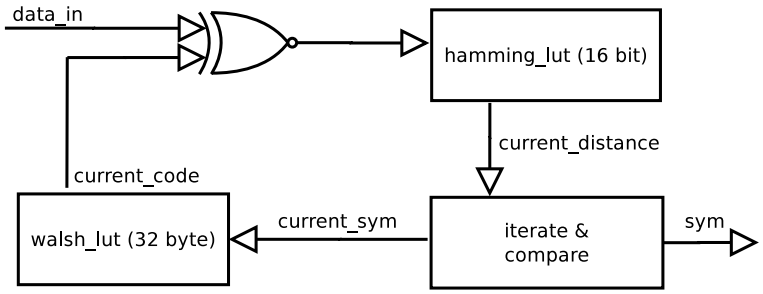

Figure 3.9.: Walsh code correlator

`iterate & compare` block contains a register that stores the minimum Hamming distance observed at the output of the `hamming_lut` block. This register is reset every 16 clock cycles. At the same time, the `iterate & compare` block cycles though each possible 4-bit symbol and addresses a 32 byte LUT containing all of the Walsh code words. The input data is then XORed with the current test symbol and compared against the minimum hamming distance register. The output of the `iterate & compare` block constantly updates to the closest code word to `data_in`. An example of the operation of the Walsh code correlator is shown in Figure. 3.10.

The `hamming_lut` block uses the same architecture as the comma detection block described in Section. 3.1.6, except that it is 16-bits wide. The `walsh_lut` LUT uses two of the Spartan 6 block ram `RAM16X8S` primitives.

3.1.10. FIFO

After forward error correction, the demodulated symbols are shifted into a 32-bit shift register. Once a full 32-bit word is accumulated, consisting of 8 symbols, the receiver issues a FIFO write enable signal.

This signal is mapped to a 128x32-bit FIFO primitive provided by the Spartan 6. The FIFO allows for two completely independent read and write clocks. The receiver is synchronised to the write clock, while the Microblaze CPU is synchronised to the CPU clock. The use of a FIFO with a depth of 128 allows for two complete IEEE 802.15.6 packets to be stored without CPU intervention.

3.1.11. Descrambler

Scrambling and descrambling is performed by the embedded Microblaze CPU as part of the *hbc_mac* software stack described in Section. 5.1.3.

As discussed in Section. 2.2.3 the output of the transmitter scrambler is XORed with the data stream before the Walsh encoding block of the transmitter. Descrambling requires XORing the data with the same sequence at the receiver. The scrambler seed is known to the receiver by reading the IEEE 802.15.6 packet header. The scrambler is reset at the beginning of each packet.

While it is possible to generate the scrambler sequence in software, it is not possible to do this in real time without interruption to the transmitter packet stream. Real time descrambling is possible by providing the embedded CPU with a hardware module that is capable of producing 32 bits of the scrambler sequence within one clock cycle.

The scrambler module consists of a linear feedback shift register implemented in hardware, with a CPU interface.

3.1.12. Multiple access

IEEE 802.15.6 stipulates that multiple access to the human body communications channel is arbitrated using either carrier sense multiple access with collision avoidance (CSMA/CA) or slotted Aloha. Multiple access is handled by the *hbc_mac* firmware, however in the case of CSMA/CA the MAC layer must be able to detect when other nodes are transmitting. The `pkt_active` signal in Figure. 3.2 is asserted by the state machine for all states except for pre-alignment or packet end. The *hbc_mac* firmware is able to check this signal before attempting to transmit a packet.

3.2. Tool-chain

Simulation and implementation of each of the previously described modules using FPGA technology was achieved using two software tools. GHDL for simulation and the Xilinx ISE tool-chain for synthesis. For the embedded CPU, GCC was used for compilation of the *hbc_mac* software repository described in Section. 5.1.3.

The HDL repository *transceiver* contains the hardware descriptions for all of the transceiver modules written in VHDL. The hardware descriptions are VHDL93 compliant, with the exception of the use of a preprocessor to automate repetition of component instantiation and to enable common headers to be used by the embedded CPU software stack as well as the hardware descriptions.

The GCC C preprocessor is first passed over all hardware description design units, before being passed to the synthesis or simulation tools. This process is automated through the use of GNU Make and associated Makefiles throughout the repository.

3.2.1. Synthesis

For most modules, the Xilinx synthesis tool, XST is capable of inferring the correct logic from the hardware descriptions. The only exceptions to this are where the use of FPGA primitives are required to meet timing constraints. The hardware descriptions are intended to be hardware agnostic otherwise.

Table 3.2.: FPGA Primitives required by hardware descriptions

Module	FPGA Primitives required
`toplevel.vhd`	Xilinx Microblaze CPU IP core, **PLL_BASE, BUFG**
`hbc_rx.vhd`	Xilinx FIFO IP core
`hamming_lut.vhd`	**LUT6**
`walsh_enc_lut.vhd`	**RAM16X8S**
`ddr.vhd`	**ODDR2, IDDR2, IODELAY2, IOBUF, OBUF, OBUFT**

Porting the design to another architecture than Spartan 6 will require modification of the modules listed in Table. 3.2. While not strictly required for implementation of the transceiver, the DDR memory interface has been included in the table as it allows for large packet buffers, as well as stand-alone operation without connection to a host.

In addition to the transceiver modules described in detail in Section. 3.1, there are several other modules that are required to create the complete embedded system. These include SPI interfaces for communication to a host, a DDR memory interface, the embedded Microblaze CPU, Flash memory interface, PSoC programming interface as well as clock and reset logic. All of these modules are instantiated by `toplevel.vhd`, and are included in the hardware description repository.

Device utilisation for the full embedded system is 25% of registers, 66% of LUTs and 100% of block RAMs. The block RAMs have been fully utilised to provide the maximum FIFO and CPU RAM capacity.

Extensive power usage simulation has been completed for the transmitter stage using gate level simulation of the transmitter section and the Xilinx power estimation tools. The results have been published in [30]. The dynamic power consumption of the design, when implemented in Virtex 5 technology is 4.5nJ per bit with a spreading factor of 8. The power consumption varies approximately linearly with spreading factor.

3.2.2. Simulation

A complete simulation environment has been developed as part of this project for both the digital and analogue components of the design. Digital simulation is performed using GHDL and analogue simulation is performed using *ngspice*.

A simulation run consists of the following work-flow:

1. Define the packet spreading factor by specifying `PACKET_RATE`.
2. Set up the analogue interfaces using schematic capture. The repository contains a spice model of the transmit filter and receiver comparator stages for

loop back testing.

3. Set up an analogue model of the communications channel using spice elements.
4. Run the transmitter digital simulation, by issuing “make transmitter_tb”.
5. Run the analogue simulation by issuing “make” in the analogue subdirectory.
6. Run the receiver digital simulation, by issuing “make receiver_tb”.

The makefiles automate the process of exporting the data from the digital to analogue domains. This is achieved through the use of several small applications developed as part of this project. The source code for the simulation tools is included within the hardware description repository.

Alternatively for digital domain only testing, there is a complete transmitter to receiver path test bench implemented. This can be run by issuing “make loopback_tb”.

An example of the output of the simulator is shown in Figure. 3.10. The transmitter stream can be seen synchronised to `serial_clk`. `s_data_in` represents the output of the analogue simulation and is no longer synchronised to `serial_clk`. `s_data_in_sync` is the output of the data synchroniser stage, shown the input stream re-synchronised as expected. The remaining signals shown the demodulator at work.

Figure. 3.10 confirms correct operation of the Walsh correlator. `walsh_reg` is compared with `cur_walsh`, which cycles through all of the Walsh codes. `cur_distance` shows 16 minus the current hamming distance between the test symbol and the data symbol. `max` is updated where appropriate and the correct symbol is eventually selected as 6.

In addition to the simulation test benches mentioned above, there are also individual test benches for each module of the system. These are all located within the hardware description repository under the subdirectory “testbench”.

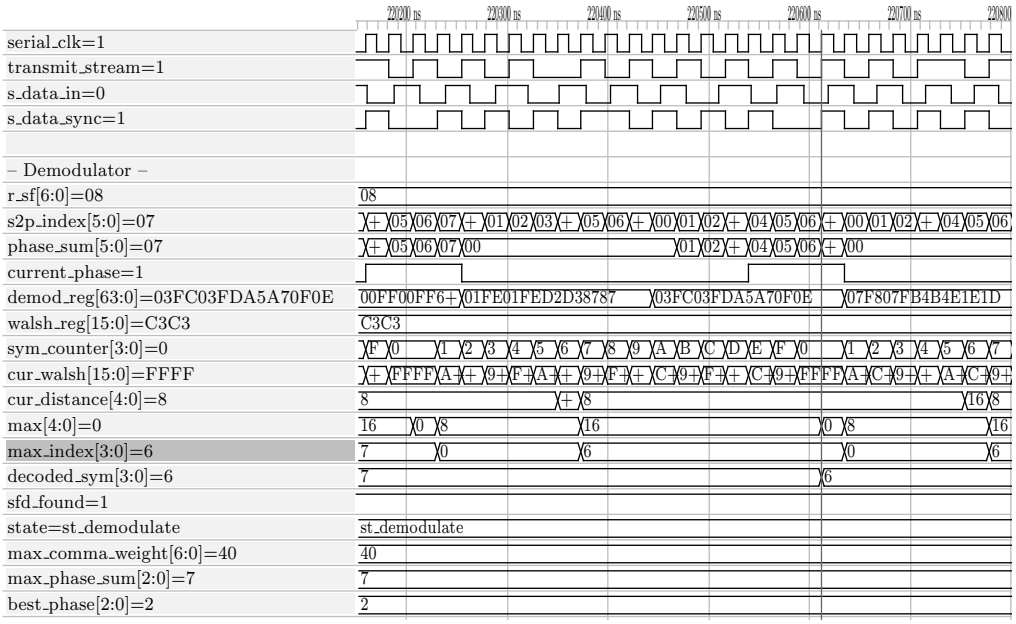

Figure 3.10.: Simulation of digital and analogue components

3.3. Testing

Both simulation and real world testing has been completed for all transceiver modules and for the system as a whole. Testing the system can be achieved using two methods. The first method involves loop-back testing using a single transceiver. The second, and more thorough method consists of passing packets through a transmitter and receiver that are directly connected to each other, eliminating the influence of the analogue electronics and communications channel.

Loop back testing of a transceiver is possible using the *psoc_ctrl* command line utility described in Section. 5.2.2. The utility instructs the transceiver simultaneously transmit and receive. The transmitted packet is then checked for errors and the error status is reported to the user.

Full digital protocol testing requires two transceivers and a method of directly connecting the transmit and receive pins. Connection is possible using the general purpose I/O connector on the prototype transceiver.

3.3.1. Sustained digital system throughput

The sustained throughput of the complete digital system is shown in Table. 3.3. This test was conducted by first loading the DRAM described in Section. 5.1.3 with 1MB (8Mb) of data. The transmitter was then instructed to send the full data set, partitioned into 8192 packets of 128 bytes each. The time taken to transmit the full 1MB includes memory fetches, header generation and packet framing. In order to characterise the performance of the transceiver, sustained data rate is compared to the maximum theoretical data rate of IEEE 802.15.6.

The protocol overhead for each packet consists of the preamble bits, SFD and header and is shown in equation 3.1. Separating the equation into protocol overhead chips c_o and payload chips c_p, equation 3.1 may be re-written as:

$$c_o = 4n_p + n_{sfd} + \frac{8}{R_c} sl_h \qquad \text{and} \qquad c_p = \frac{8}{R_c} sl$$

The maximum theoretical data rate r_p may be written in terms of the chip rate f_c as:

$$r_p = \frac{8 f_c l}{c_o + c_p} \tag{3.7}$$

The results in Table. 3.3 show that the transceiver is able to achieve a bit rate of 93% or the theoretical maximum at a spreading factor of 64, reducing to 64% at a spreading factor of 8. The reduction is a result of the relatively constant inter packet time t_i which is given by:

$$t_i = t_p \left(\frac{r_p}{r_d} - 1 \right)$$

At higher spreading factors, there is an improvement in BER as expected, however the difference is not as substantial as is implied by equation 3.5. The independence

Table 3.3.: Digital system performance while transmitting 8Mbits of data using a packet size of 128 bytes

Spreading factor	Time	Dropped packets	BER	Data rate (r_d)	IEEE 802.15.6 Maximum data rate (r_p)	Inter-packet time t_i
64	57s	228	0.028	147Kbps	158Kbps	486μs
32	30s	253	0.031	280Kbps	312Kbps	375μs
16	18s	270	0.033	466Kbps	612Kbps	524μs
8	11s	283	0.035	763Kbps	1.18Mbps	474μs

of BER on spreading factor indicates that alternative sources of packet loss are more significant at high signal to noise ratios. Packets may be dropped due to either a failed CRC check, or loss of synchronisation, or incorrect receiver state at the time of packet transmission. The data in Table. 3.3 reflects all of these effects combined.

3.4. Summary

The digital transceiver core described within this chapter is capable of meeting the requirements of the physical layer described in Chapter 10 of IEEE 802.15.6. A complete set of hardware description modules have been developed to allow for transmission and reception of IEEE 802.15.6 packets. The hardware descriptions have been tested using FPGA technology and are capable of a maximum throughput of 763Kbps. A testbench environment is provided in conjunction with the hardware descriptions that allows for both digital simulation, and mixed signal simulation using a combination of the software packages *GHDL* and *ngspice.* A full implementation of an IEEE 802.15.6 transceiver consists of both the digital core and an analogue front end that is responsible for filtering and amplification of the digital signal. This is discussed in the following chapter.

4. Transceiver analogue front end

While the majority of the transceiver design consists of modulation, demodulation and synchronisation using digital logic, it is essential that the signal is appropriately coupled onto the human body with enough gain in the receiver stage to compensate for the attenuation of the human body communications channel. The analogue sections of the prototype transceiver may be divided into two parts. The first part consists of a transmitter filter stage, followed by coupling onto to the surface of the skin. The transmitter filter is required by IEEE 802.15.6 to ensure electromagnetic compatibility with other communications standards such as MICS.

The second requirement is an amplifier and detector stage, with the responsibility of extracting the weak input signal and generating a digital data stream suitable for the input synchronisation logic of the FPGA.

4.1. Transmitter Design

The analogue component of the transmitter consists only of the transmit filter and the output electrode. The transmit filter is AC coupled to the FPGA. The FPGA was configured for 2.5 volt operation, so that the input stream to the filter appears approximately as a 21MHz square wave input with an amplitude of 2.5V peak to peak.

4.1.1. Filter requirements

The electromagnetic compatibility requirements of IEEE 802.15.6 state that the transmitter spectrum be restricted to the fundamental frequency of 21MHz. Figure. 4.1 and Table. 4.1 show the specifications of the transmit filter mask.

Table 4.1.: IEEE 802.15.6 Transmit power limits

Frequency (MHz)	Relative power (dBr)
<1	-120
<2	-80
<18.375	-3
>23.625	-3
>50	-25
>105	-34
>400	-75

To achieve compliance with IEEE 802.15.6 it is necessary to filter the output of the FPGA. Figure. 4.1 shows that there is already a natural roll-off in the spectrum of the transmit signal in either direction of the fundamental frequency, however it is not sufficient to meet the standard's requirements.

Considering the design goal of a low power transmitter, and the decision to remove any digital to analogue conversion, the remaining option is to use an analogue filter. This section deals with the design of such a filter, with an emphasis on minimising component count and power consumption.

Several filter design approximations were tested in simulation before completing the design. These included Butterworth, Bessel and Elliptical. It can be seen from Figure. 4.1 that the high pass filter has far more onerous requirements than the low pass stage, where it was found that a two pole filter is sufficient.

The following sections will deal with the two extremes of filter approximation. The first filter considered was the Bessel filter, known for its maximally flat group delay.

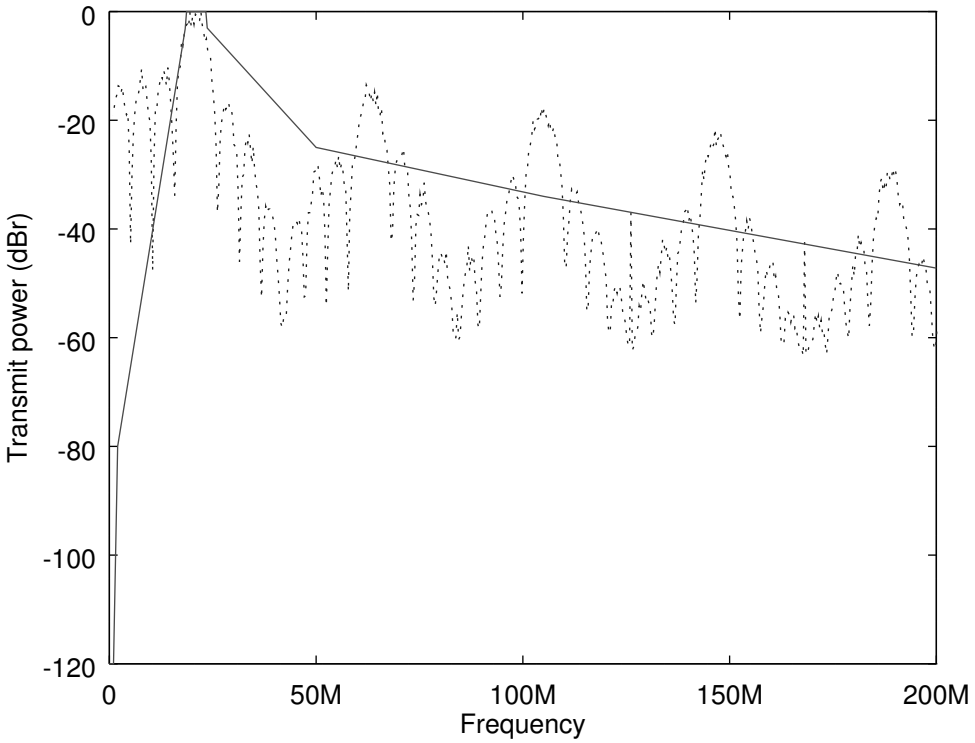

Figure 4.1.: IEEE 802.15.6 Transmit power spectrum mask. The dotted line shows the spectrum as measured using the prototype transceiver.

In particular, Bessel filters do not ring in response to the unit step function [32]. This property was considered desirable as the resulting time domain output signal of the transmitter might be easily detected using a hard decision receiver. On the other hand, an Elliptical filter approximation allows the minimum order high pass filter stage given the design parameters. Table. 4.2 outlines the design parameters that were used for both the Bessel and Elliptical approximations.

Table 4.2.: Filter design parameters

Parameter	Symbol	Value
Minimum stop band attenuation	A_s	80dB
Maximum pass band ripple	A_p	2dB
Pass band edge frequency	ω_p	18MHz
Stop band edge frequency	ω_s	2MHz

4.1.1.1. Bessel filter

Given the requirements of Table. 4.2, the minimum order required for a Bessel approximation is 6. A time domain simulation of the output data stream using a 6th order high pass filter and a 2nd order low pass filter is shown in Figure. 4.2. Unfortunately the phase transitions are difficult to see, due to the step response time of the filter and subsequent interference from previous input transitions. After comparing this output to the output of the Elliptical filter below, the Bessel approximation was considered not appropriate for this application.

4.1.1.2. Elliptical filter

The same simulation was run using the minimum order elliptical filter required to meet the transmit mask. The design requirements of Table. 4.2 can be met with a 3rd order high pass stage, significantly reducing component count and circuit complexity and sensitivity. It can be seen from Figure. 4.3 that the signal transitions between phases rapidly, without any spurious zero crossings.

The combination of superior time domain performance, in this application, and reduced component count resulted in the Elliptical filter approximation being selected for the transmit filter. A brief description of the Elliptical filter approximation follows:

The elliptical filter approximation takes the form:

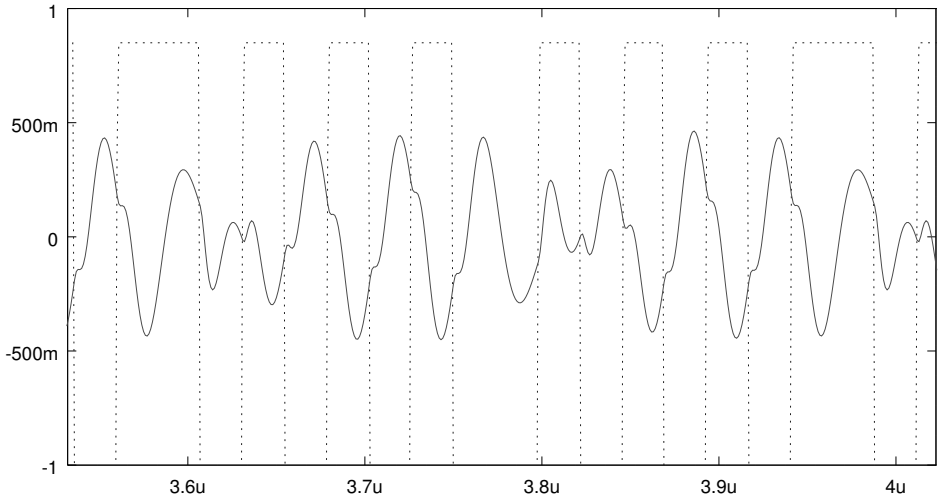

Figure 4.2.: Time domain simulation of a 2nd order low pass, followed by a 6th order high pass Bessel filter.

$$|H(j\omega)|^2 = \frac{1}{1 + \epsilon_p^2 f_N^2(\omega)}$$

Where f_n is a rational function of order N with the desirable properties that within the pass band, $f_n^2 \leq 1$, and within the stop band, $f_n^2 \geq L^2$. This results in the transfer function restrictions:

$$|H(j\omega)|^2 \in \begin{cases} [\frac{1}{1+\epsilon_p^2}, 1] & \text{for } \omega < \omega_p \\ [0, \frac{1}{1+\epsilon_p^2 L^2}] & \text{for } \omega > \omega_s \end{cases} \tag{4.1}$$

where ω_s is the stop band cut off frequency and ω_p is the pass band cut off frequency. The filter may be tuned for the desired pass band ripple by adjusting ϵ_p, while the minimum attenuation in the stop band may be tuned using L. Translating ϵ_p and L into the more familiar ripple attenuation A_p and minimum stop band attenuation A_s expressed in decibels, we have:

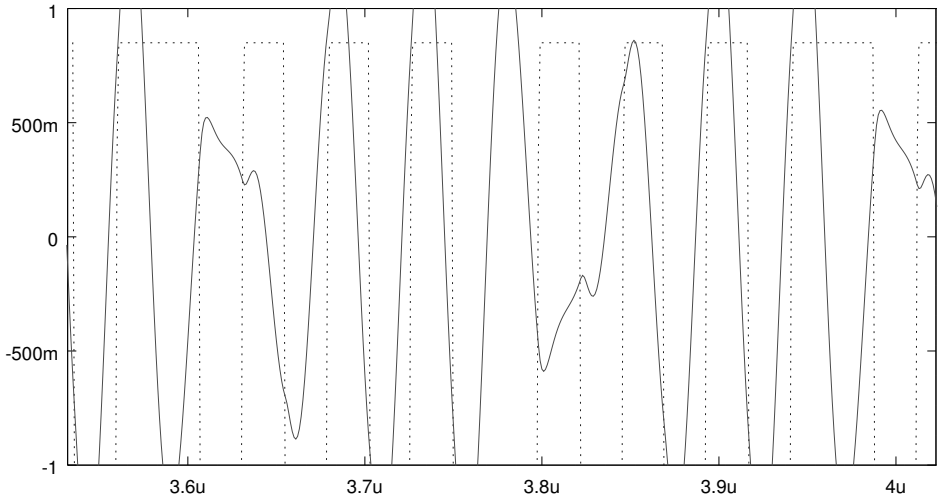

Figure 4.3.: Time domain simulation of a 2nd order low pass filter, followed by a 3rd order high pass elliptical filter. The output of the filter is shown in bold.

$$A_p = 10\log_{10}(\epsilon_p^2 + 1) \qquad \text{and} \qquad A_s = 10\log_{10}(\epsilon_p^2 L^2 + 1)$$

A realisable $H(s)$ must have real coefficients and therefore f_n must satisfy:

$$f_n(s) = \frac{P(s)}{Q(s)} \qquad \text{or} \qquad f_n^2(\omega) = \frac{P(j\omega)P(-j\omega)}{Q(j\omega)Q(-j\omega)}$$

Re-arranging for the transfer function, H gives:

$$\begin{aligned} |H(j\omega)|^2 &= \frac{Q(j\omega)Q(-j\omega)}{Q(j\omega)Q(-j\omega) + \epsilon_p^2 P(j\omega)P(-j\omega)} \\ H(s) &= \frac{Q(s)}{Q(s) + \epsilon_p^2 P(s)} \end{aligned} \tag{4.2}$$

The equiripple[1] *Chebyshev rational functions* $R_n(\omega, L)$ of order n satisfy these conditions and are used to find the poles and zeros of $H(s)$. Figure. 4.4 demonstrates an example R_N, where $N = 5$. Within the pass band, $R_N < 1$ as required. Beyond the pass band, $R_N > L$.

The general procedure to find the poles and zeros of $H(s)$ that satisfy the equiripple conditions given in 4.1 involves choosing the order N and calculating L from the design parameters, A_p and A_s. The poles and zeros of $R_N(\omega, L)$ may then be used with 4.2, along with ϵ_p to find the transfer function. The transfer function may then be frequency scaled for the desired pass band frequency, ω_p.

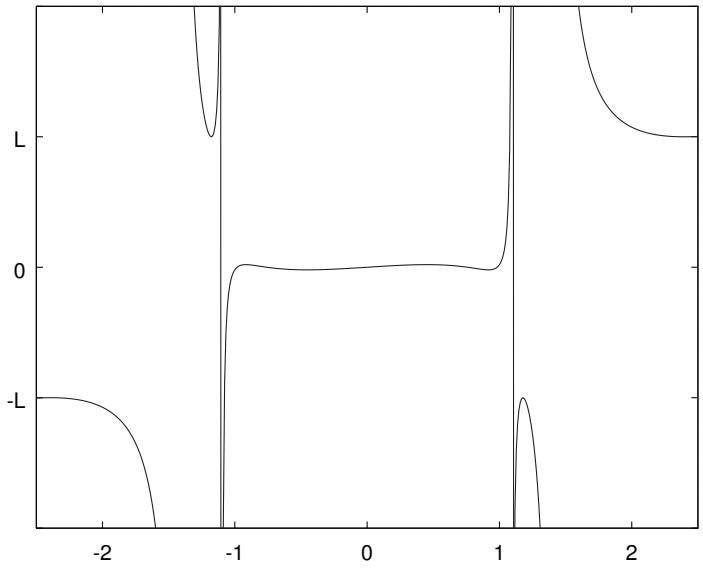

Figure 4.4.: A plot of the 5th order Chebyshev rational function $R_5(\omega, L)$ against ω showing equiripple behaviour in the pass band and stop bands. The pass band has been normalised so that $\omega_p = 1$.

The *signal* package from the software repository *Octave-Forge[33]* provides a func-

[1]Equiripple here means that within the pass band, the variation in attenuation, or 'ripple' is bound by A_p. Similarly within the stop band, the minimum attenuation is bound by A_s. These conditions impose restrictions on the ratios $\omega_p : \omega_s$ and $A_p : A_s$ so that the general rational function $f_N(\omega, \omega_p, \omega_s, A_p, A_s)$ becomes $R_N(\omega, L)$ when normalised using $\omega_p = 1$.

tion `ellip()` that computes the poles and zeros of $H(s)$ for a given set of design parameters: A_p, A_s, N and ω_p. This function was used to produce the values shown in Table. 4.3 and Table. 4.4. A graphical representation is given in Figure. 4.5

Table 4.3.: High pass filter design requirements produced using *Octave-Forge*

	Required			**Twin-T**			**HPF**		
	f (MHz)	Q	Order	f (MHz)	Q	Order	f (MHz)	Q	Order
Zeros	0	N/A	1	1	∞	2	0	N/A	2
	1.0	∞	2						
Poles	19.1	2.56	2	49.8	N/A	1	19.1	2.56	2
	49.8	N/A	1	0.02	N/A	1			

Table 4.4.: Low pass filter design requirements produced using *Octave-Forge*

	LPF		
	f (MHz)	Q	Order
Poles	20.9	1.13	2

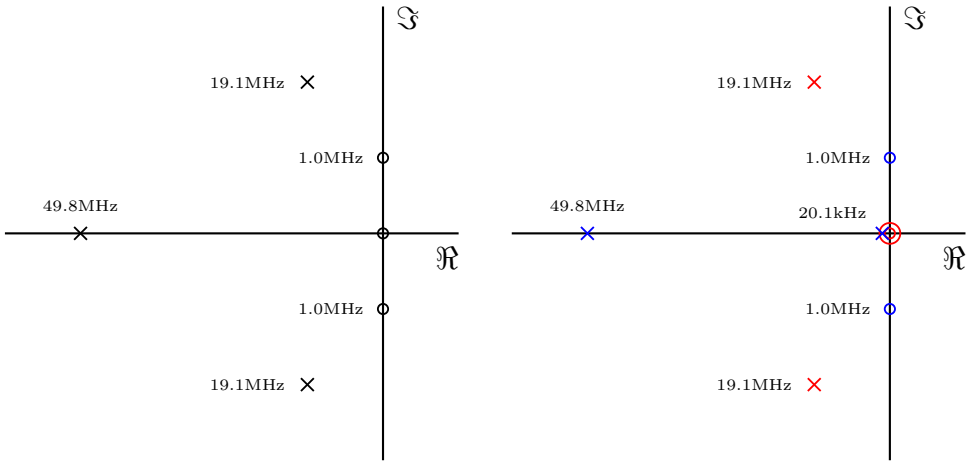

Figure 4.5.: Pole and zero locations. The required locations for a third order elliptical filter are shown on the left. The right hand side shows an approximation of the same filter using a Twin-T circuit (blue) and a bi-quad stage (red)

4.1.2. Filter implementation

The following section deals with the physical realisation of the filter requirements developed in Section. 4.1.1. The 2nd order low pass stage is based on the bi-quad filter presented below. The 3rd order high pass filter required two stages, consisting of a passive Twin-T circuit driven directly by the buffered output of the FPGA, then a Sallen & Key stage. The need for the Twin-T circuit is demonstrated in the following section.

4.1.2.1. High pass notch circuit

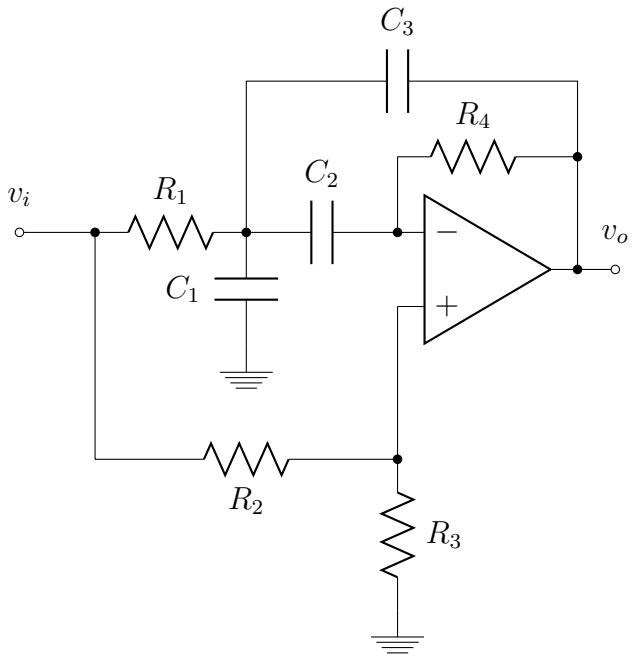

Figure 4.6.: High pass notch circuit

To understand the motivation for the Twin-T configuration adopted, it is necessary to briefly consider the single stage high pass notch circuit and its limitations. The transfer function of the circuit in Figure. 4.6 is:

$$H(s) = K\frac{s^2 + \alpha s + \omega_z^2}{s^2 + \frac{\omega_0}{Q}s + \omega_0^2}$$

Where:

$$K = \frac{R_3(C_1 + C_3)}{C_3(R_2 + R_3)}$$

$$\omega_0 = \frac{1}{\sqrt{R_1 R_4 C_2 C_3}}$$

$$\omega_z = \frac{1}{\sqrt{R_1 R_4 C_2 (C_1 + C_3)}}$$

$$Q = \frac{\omega_0 (R_4 C_2 C_3)}{(C_1 + C_2 + C_3)}$$

$$\alpha = \frac{R_1 R_3 (C_1 + C_2 + C_3) - R_2 R_4 C_2}{R_1 R_3 R_4 (C_1 + C_3) C_2}$$

In order to realise two zeros on the imaginary axis, it is necessary for $\alpha = 0$. The normalised values with $\omega_0 = 1$ given in [32] are:

$R_1 = R_2 = 1$, $C_2 = C_3 = C$ and $C_1 = kC$. In [32] it is shown that:

$$k = \frac{\omega_0^2}{\omega_z^2} - 1$$

$$R_3 = (2 + k)Q^2$$

$$R_4 = (2+k)^2 Q^2$$

Unfortunately, this quickly leads to unrealistic component values as k increases. The design requirements from Table. 4.3 result in $k = 360$. This results in $R_4 \approx 860k\Omega$. After applying a reasonable impedance scaling $R_1 = 1k\Omega$, $R_4 \approx 860M\Omega$. The large value of k, a result of the large ratio between the pole and zero frequencies renders this circuit inappropriate. As shown in Figure. 4.5, the poles and zeros may instead be approximated through the combination of a Twin-T circuit and a high pass filter.

4.1.2.2. Twin-T circuit

The general denominator $s^2 + \frac{\omega_z}{Q} + \omega_z^2 = 0$, with $Q \leq \frac{1}{2}$ may be solved as:

$$-s_{1,2} = \frac{\omega_z}{2}\left(\frac{1}{Q} \pm \sqrt{\frac{1}{Q^2} - 4}\right) = \omega_{z1,z2}$$

In the case where $Q = \frac{1}{2}$, the two zeros $s_{1,2}$ will converge at ω_z. If we decrease Q, it is possible to "spread" the roots along the negative real axis. We will use this technique to obtain roots ω_{z1} and ω_{z2}.

Looking at the relationships between the two roots, we have:

$\omega_{z1}\omega_{z2} = \omega_z^2$ and $\omega_{z1} + \omega_{z2} = \frac{\omega_z}{Q}$ It follows from this that if we seek ω_{z1} relatively large, then ω_{z2} will approach zero. This is used to approximately cancel the double zero provided by the following Sallen and Key stage. Since we are no longer interested in the value of ω_{z2} we can re-write the relationship with Q as:

$$Q = \frac{\omega_z \omega_{z1}}{\omega_{z1}^2 + \omega_z^2} \tag{4.3}$$

The simplest RC topology that can realise two zeros on the imaginary axis is the Twin-T circuit.

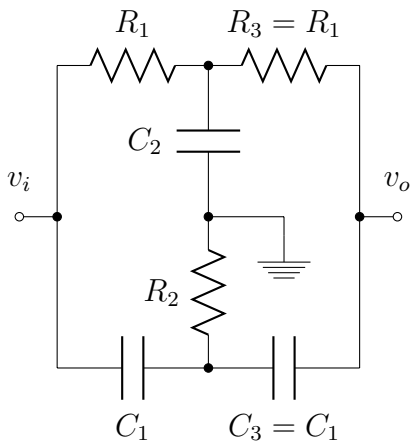

Figure 4.7.: Twin T circuit

One method of obtaining the transfer function of the circuit in Figure. 4.7 is to follow the approach taken by [34] and shown in Figure. 4.8. The star delta relationship between Figure. 4.8(a) and (b) is given by:

$$Y_{12} = \frac{Y_1 Y_2}{Y_1 + Y_2 + Y_3},\ Y_{13} = \frac{Y_1 Y_3}{Y_1 + Y_2 + Y_3},\ Y_{23} = \frac{Y_2 Y_3}{Y_1 + Y_2 + Y_3}$$

For the network of Figure. 4.7 composed of R_1, $R_3$and C_2:

$$Y_{12} = \frac{1}{R_1 R_3 C_2 s + R_1 + R_3},\ Y_{23} = \frac{R_1 C_2 s}{R_1 R_3 C_2 s + R_1 + R_3} \tag{4.4}$$

and for the network of Figure. 4.7 composed of C_1, C_3 and R_2:

$$Y'_{12} = \frac{R_2 C_1 C_3 s^2}{R_2(C_1 + C_3)s + 1},\ Y'_{23} = \frac{C_3 s}{R_2(C_1 + C_3)s + 1} \tag{4.5}$$

Looking at the divider in Figure. 4.8 (d) the relationship between v_o/v_i is:

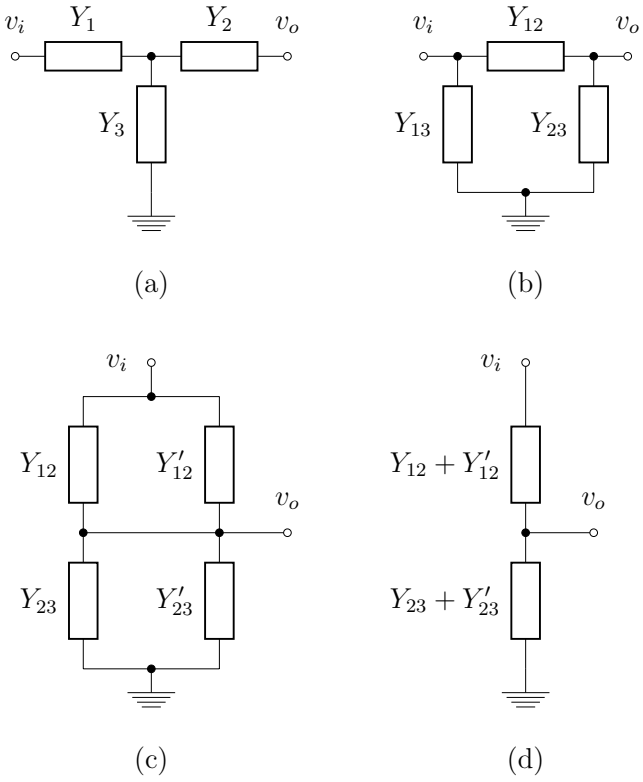

Figure 4.8.: Transformations for analysis of the Twin-T circuit. (a) represents one half of the circuit in Figure. 4.7, (b) is the star delta transformation of (a), (c) represents the parallel combination of the two delta circuits comprising Figure. 4.7, (d) shows the final voltage divider form.

$$H(s) = \frac{v_o}{v_i} = \frac{Y_{12} + Y'_{12}}{Y_{12} + Y'_{12} + Y_{23} + Y'_{23}}$$

The resulting transfer function of Figure. 4.7 is given by equation 4.6

$$H(s) = \frac{s^3 + (\frac{1}{R_1C_2} + \frac{1}{R_3C_2})s^2 + (\frac{1}{R_1R_3C_1C_2} + \frac{1}{R_1R_3C_2C_3})s + \frac{1}{R_1R_2R_3C_1C_2C_3}}{s^3 + (\frac{1}{R_1C_2} + \frac{1}{R_3C_2} + \frac{1}{R_2C_1} + \frac{1}{R_3C_1} + \frac{1}{R_3C_3})s^2 + (\frac{1}{R_1R_3C_1C_2} + \frac{1}{R_1R_3C_2C_3} + \frac{1}{R_1R_2C_1C_2} + \frac{1}{R_2R_3C_1C_2} + \frac{1}{R_2R_3C_1C_3})s + \frac{1}{R_1R_2R_3C_1C_2C_3}}$$

(4.6)

Similar to the analysis in [35] the following substitutions may be made to simplify equation 4.6:

$\alpha = \frac{1}{R_1 C_2}$, $\beta = \frac{1}{R_3 C_2}$, $\gamma = \frac{1}{R_3 C_1}$, $\sigma = \frac{1}{R_2 C_1}$ and $\delta = \frac{1}{R_3 C_3}$ which yields:

$$H(s) = \frac{s^3 + (\alpha+\beta)s^2 + (\alpha\gamma + \alpha\delta)s + \alpha\sigma\delta}{s^3 + (\alpha+\beta+\sigma+\gamma+\delta)s^2 + (\alpha\gamma+\alpha\delta+\alpha\sigma+\beta\sigma+\sigma\delta)s + \alpha\sigma\delta} \quad (4.7)$$

Imposing the condition that the zeros fall on the imaginary axis, the numerator of 4.7 has the form:

$$(s - z_3)(s^2 + \omega_z^2) \equiv s^3 - z_3 s^2 + \omega_z^2 s - z_3\omega_z^2$$

Where z_3 is the zero lying on the negative real axis. The restrictions on parameters α, β, γ, σ and δ thus become:

$$\alpha + \beta = -z_3, \quad \alpha(\gamma+\delta) = \omega_z^2 \quad and \quad \sigma\delta = -z_3(\gamma+\delta) \quad (4.8)$$

The denominator can now be re-written as:

$$s^3 + (\sigma+\gamma+\delta-z_3)s^2 + (\omega_z^2 - z_3(\sigma+\gamma+\delta))s - z_3\omega_z^2 \equiv (s-z_3)(s^2 + (\sigma+\gamma+\delta)s + \omega_z^2)$$

Therefore the condition that the zeros fall on the imaginary axis results in a common root for both the numerator and the denominator, reducing the order of the transfer function by one:

$$H(s) = \frac{s^2 + \alpha(\gamma+\delta)}{s^2 + (\sigma + (\gamma+\delta))s + \alpha(\gamma+\delta)} \quad (4.9)$$

Equation 4.9 demonstrates that the transfer function is no longer dependent on β. Also, a degree of freedom is removed since $\gamma + \delta$ may be combined, simplifying component selection. Therefore, without loss of generality the Twin-T circuit may be reduced to the symmetrical configuration shown in Figure. 4.7 with $R_3 = R_1$ and $C_3 = C_1$. Consequently, $\alpha = \beta$ and $\gamma = \delta$. The conditions given in 4.8 now reduce to:

$$\alpha = \frac{-z_3}{2}, \quad \sigma = -2z_3 \quad and \quad 2\alpha\gamma = \omega_z^2$$

It can be seen that $\sigma = 4\alpha$, or equivalently, $4 = \frac{R_1 C_2}{R_2 C_4}$.

Following the approach taken by [36], the normalisation factors $g = \frac{2\gamma}{\alpha}$ and $p = \frac{s}{\alpha}$, may be used to simplify 4.9 as:

$$H(p) = \frac{(p^2 + g)}{p^2 + (4 + g)p + g}$$

This may be solved for the required Q from 4.3 by manipulating g. The poles exist at $p^2 + (4 + g)p + g = 0$, so we have $Q = \frac{\sqrt{g}}{4+g}$. Solving for g:

$$g_{1,2} = \frac{1 - 8Q^2 \pm \sqrt{1 - 16Q^2}}{2Q^2} \tag{4.10}$$

The zeros may be found using $p^2 + g = 0$, which is equivalent to: $s^2 + \frac{2}{R_1^2 C_1 C_2} = 0$, or:

$$\omega_z = \frac{1}{R_1}\sqrt{\frac{2}{C_1 C_2}} \tag{4.11}$$

Combining equations 4.10 and 4.11 then expressing the values in terms of C_2:

$$C_1 = \frac{2C_2}{g}$$

$$R_1 = \frac{1}{\omega_z}\sqrt{\frac{2}{C_1 C_2}}$$

$$R_2 = \frac{R_1 C_2}{4C_1}$$

Note that there are two solutions for g, namely g_1 and g_2. The component values chosen in Table. 4.5 were chosen based on the solution that produced the most sensible values for C_1, R_1 and R_2.

Table 4.5.: Twin-T component values

Parameter	Value	Component	Value	E12 Value
$\frac{\omega_z}{2\pi} = f_z$	1.0 MHz	C_1	17.8pF	18pF
$\frac{\omega_{z1}}{2\pi} = f_{z1}$	49.8 MHz	C_2	22nF	22nF
C_2	22nF	R_1	360Ω	360Ω
		R_2	111kΩ	110kΩ

4.1.2.3. Low pass stage

The low pass circuit shown in Figure. 4.9 provides two complex conjugate poles. An analysis of the low pass stage may be completed by summing the currents at nodes a and b.

$$\frac{1}{R_4} v_a = -C_4 s v_o$$

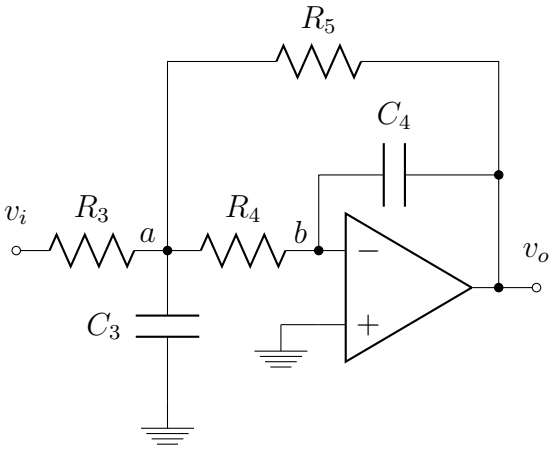

Figure 4.9.: Low pass circuit

$$\frac{1}{R_3}(v_i - v_a) = C_3 s v_a + \frac{1}{R_4} v_a + \frac{1}{R_5}(v_a - v_o)$$

Combining these we have:

$$H(s) = \frac{-R_5}{R_3} \frac{\frac{1}{R_4 R_5 C_3 C_4}}{s^2 + (\frac{1}{R_3 C_3} + \frac{1}{R_4 C_3} + \frac{1}{R_5 C_3})s + \frac{1}{R_4 R_5 C_3 C_4}} \tag{4.12}$$

which is of the general form:

$$H(s) = -K \frac{\omega_0^2}{s^2 + \frac{\omega_0}{Q} s + \omega_0^2}$$

Such that the DC gain $K = \frac{R_5}{R_3}$ may be freely adjusted along with Q and ω_0

$$Q = \frac{R_3}{C_4} \frac{\sqrt{R_4 R_5 C_3 C_4}}{R_3 R_4 + R_4 R_5 + R_3 R_5} \tag{4.13}$$

$$\omega_0 = \frac{1}{\sqrt{R_4 R_5 C_3 C_4}} \tag{4.14}$$

Normalising the equation such that $R_3 = \frac{1}{K\alpha}R$, $R_4 = \alpha R$, $R_5 = \frac{1}{\alpha}R$, $C_3 = \beta C$ and $C_4 = \frac{1}{\beta}C$ gives:

$$Q = \frac{\beta}{K\alpha + \alpha + \frac{1}{\alpha}}$$

$$\omega_0 = \frac{1}{RC}$$

Looking at Q, we first consider fixing $\beta = 1$:

$$Q = \frac{1}{K\alpha + \alpha + \frac{1}{\alpha}}$$

and note for $K, \alpha > 0$, we are limited to $0 \leq Q < \frac{1}{2}$. A better solution is to use β to arbitrarily scale Q. Based on the design parameters: Q, ω_0 and K the design equations become:

$$\beta = Q\left(K\alpha + \alpha + \frac{1}{\alpha}\right)$$

$$C_3 = \beta^2 C_4$$

$$R = \frac{1}{\omega_0 \sqrt{C_3 C_4}}$$

$$R_4 = \alpha R$$

$$R_5 = \frac{1}{\alpha} R$$

$$R_3 = \frac{1}{K} R_5$$

It is important to minimise loading on the previous passive Twin-T circuit so a design goal is to maximise the input impedance. Keeping K, Q and ω_0 constant, we try and increase R_3 using the remaining degree of freedom α. As shown in Figure. 4.10 R_3 increases as α approaches zero. At the same time, however, there is a large increase in C_3 with $\alpha < \frac{1}{2}$. This increase is not desirable as it has the effect of shunting much of the signal energy to ground. As a compromise, $\alpha = \frac{1}{2}$ provides a sufficiently large input impedance.

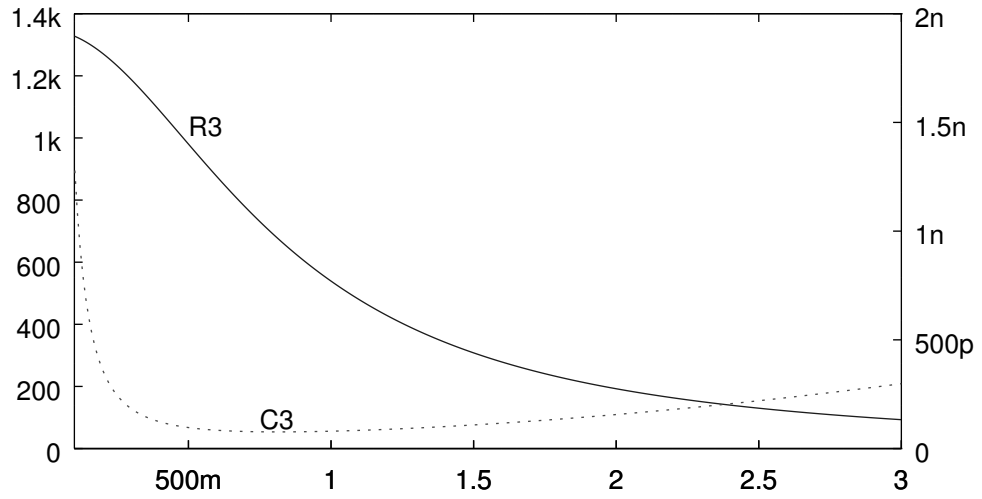

Figure 4.10.: Effect of α on R_3 and C_3

Table 4.6.: Low pass filter component values

Parameter	Value	Component	Value	E12 Value
$\frac{\omega_0}{2\pi} = f_0$	20.9 MHz	R_3	980Ω	1KΩ
Q	1.13	R_4	123Ω	120Ω
K	0.5	R_5	490Ω	470Ω
α	0.5	C_3	96.6pF	100pF
C_4	10pF	C_4	10pF	10pF

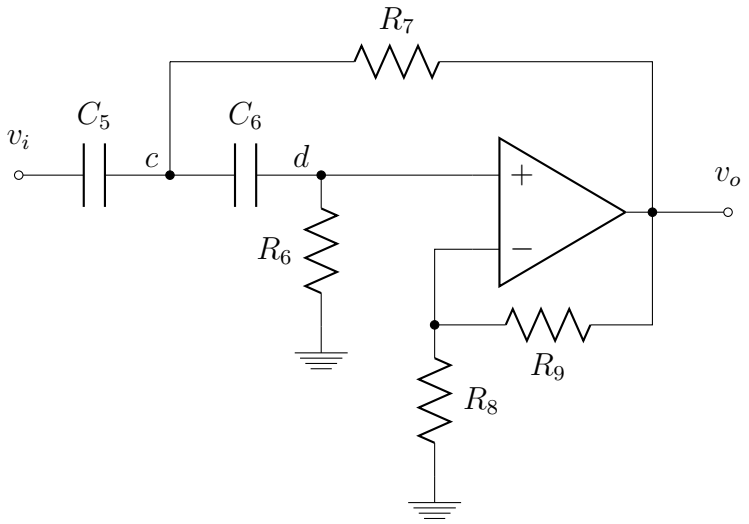

Figure 4.11.: High pass circuit

4.1.2.4. High pass stage

The final stage of the transmit filter is the Sallen & Key circuit shown in Figure. 4.11. The circuit provides two complex conjugate poles and two zeros at the origin. An analysis of the currents entering nodes c and d gives:

$$v_d = \frac{1}{K} v_o$$

$$C_5 s(v_i - v_c) + \frac{1}{R_7}(v_o - v_c) = C_6 s(v_c - v_d) = \frac{1}{R_6} v_d$$

$$v_c = \left(\frac{1}{KC_2 s}\right)\left(\frac{1}{R_6} + C_2 s\right) v_o$$

where $K = (1 + \frac{R_9}{R_8})$ is the high frequency gain. The above equations may be manipulated to give the transfer function:

$$H(s) = K \frac{s^2}{s^2 + (\frac{1}{R_6 C_5} + \frac{1}{R_6 C_6} + \frac{1}{R_7 C_5} - \frac{K}{R_7 C_5})s + \frac{1}{R_6 R_7 C_5 C_6}}$$

Mapping this to the general 2nd order equation:

$$H(s) = K \frac{s^2}{s^2 + \frac{\omega_0}{Q} s + \omega_0^2}$$

Results in Q and ω_0 as:

$$Q = \frac{\sqrt{R_6 R_7 C_5 C_6}}{R_7(C_5 + C_6) + R_6 C_6 (1 - K)}$$

$$\omega_0 = \frac{1}{\sqrt{R_6 R_7 C_5 C_6}}$$

In comparison to equation 4.13, the Q value of the positive feedback configuration chosen here is less sensitive to variations in component values. This is evident as both the numerator and denominator contain all of the element values, while there is no entry in the denominator of 4.13 for C_3. Since the inverting low pass stage may be configured to reduce the gain of the overall filter, a non-inverting configuration

is preferable for the high pass stage. Since this stage doesn't require impedance modifications, it is sufficient to just set $R_6 = R_7 = R$ and $C_5 = C_6 = C$ so that:

$$K = 3 - \frac{1}{Q}$$

$$\omega_0 = \frac{1}{RC}$$

Thus the design equations become:

$$R_{6,7} = \frac{1}{\omega_0 C}$$

$$R_9 = \left(2 - \frac{1}{Q}\right) R_8$$

Table 4.7.: High pass filter component values. The E12 values were adjusted after simulation.

Parameter	Value	Component	Value	E12 Value
$\frac{\omega_0}{2\pi} = f_0$	19.1 MHz	R_6	833Ω	820Ω
Q	2.56	R_7	833Ω	820Ω
$K(Q)$	2.61	R_8	1KΩ	1KΩ
C	10pF	R_9	1.61KΩ	1.6KΩ

4.1.2.5. Simulation

To quantify the performance of the filter, time domain, frequency domain and sensitivity simulations were run using *ngspice* [37]. The results of the simulations were

used to further fine tune the selected resistor and capacitor values. The simulations were run using a Spice, with models for the op-amps provided by the manufacturer. The op-amp models included parasitic capacitance on the input terminals.

The simulated circuit is shown in Figure. 4.12. A DC bias of 1.65V has been applied throughout, as the final design must run from a single rail power supply rated at 3.3V.

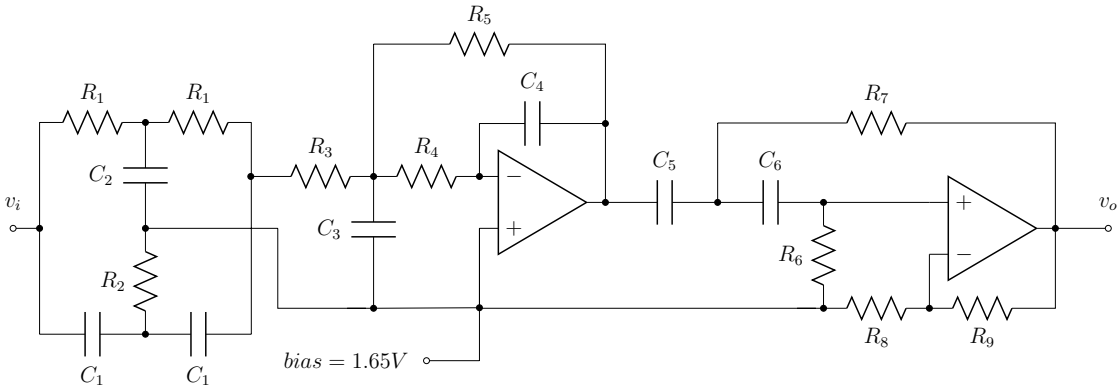

Figure 4.12.: Complete filter with biasing

To give an accurate representation of the output of the filter, the input voltage was imported from the digital simulations. Figure. 4.13 shows the output of the filter in the time domain over a section of an IEEE 802.15.6 packet including two phase changes. This may be compared with the simulated results of the elliptical filter presented in Figure. 4.3. The filter shows similar behaviour across phase changes with reduced high frequency components. From the receiver's point of view, reduced spurious zero crossings will improve detection accuracy. The reduction of high frequency components at the transition points is something of a bonus.

Figure. 4.14 shows the frequency domain analysis of the filter, demonstrating compliance with the IEEE 802.15.6 transmit mask. For the upper plot, an input waveform with spreading factor 8 is used, while the lower plot shows a spreading factor of 64. It can be seen that there is increased energy in the lower end of the spectrum when using a spreading factor of 8. This is due to the increased number of phase

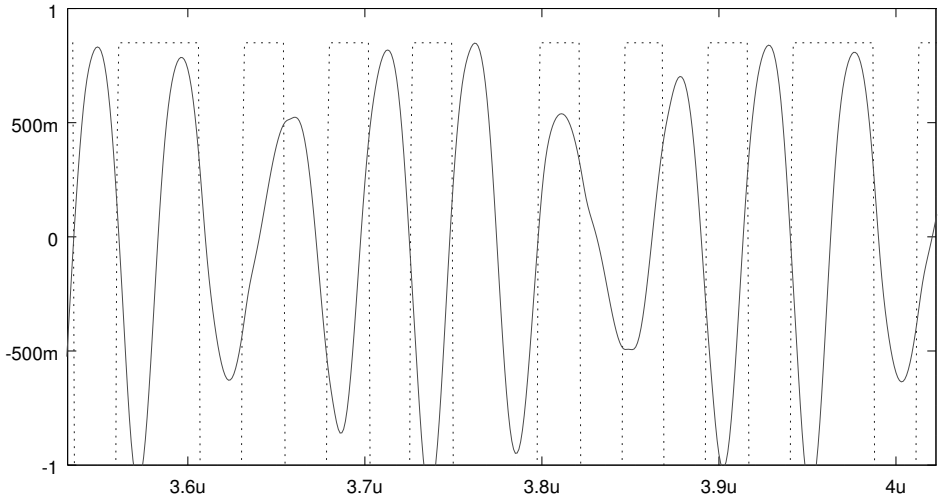

Figure 4.13.: Time domain simulation of complete filter

transitions. In both cases the filter is able to adequately meet the requirements of the transmit mask.

The frequency response of each stage of the filter can be seen in Figure. 4.15. This simulation also includes an AC analysis of an unloaded Twin-T stage identical to that used within the filter. It is possible to see the effect of the loading on the Twin-T of the low pass stage. The loaded filter shows reduced Q, however the effect is approximately linear over the frequencies of interest.

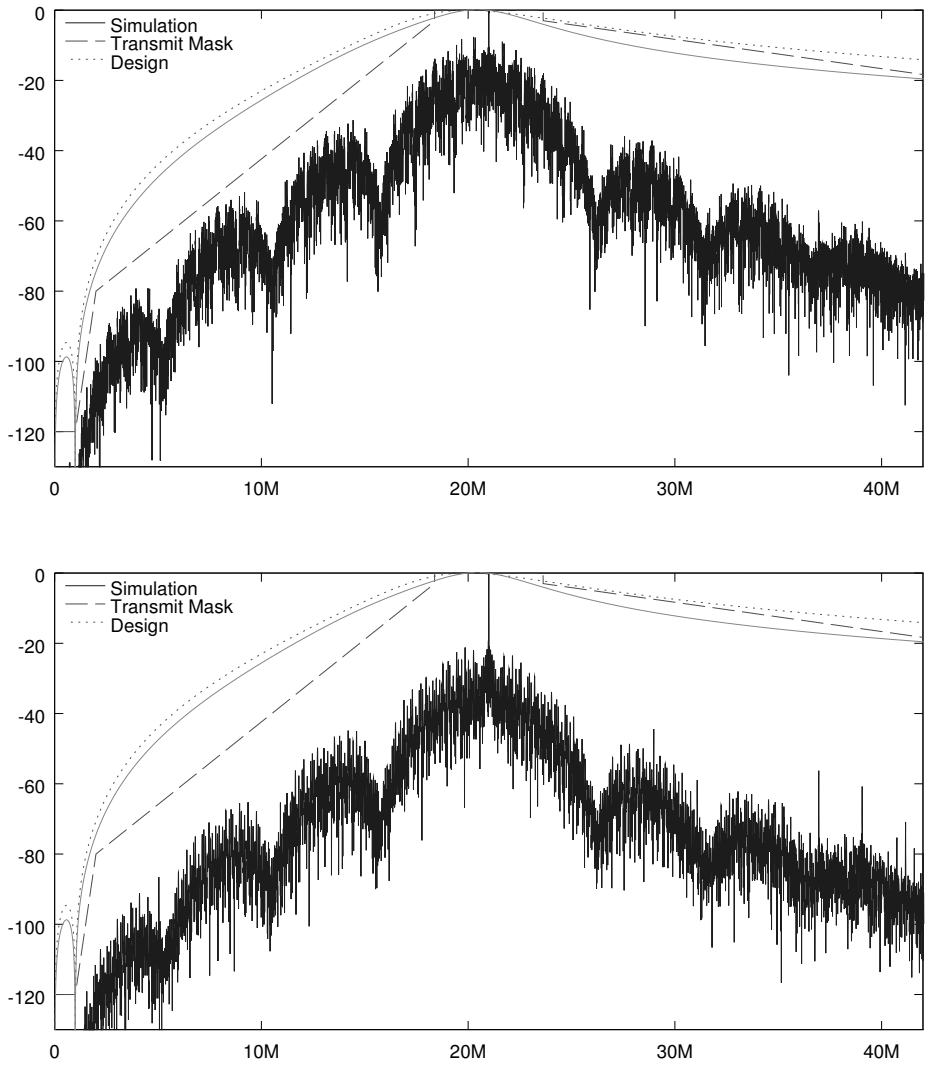

Figure 4.14.: Simulated filter output for a spreading factor of 8 (top) and 64 (bottom)

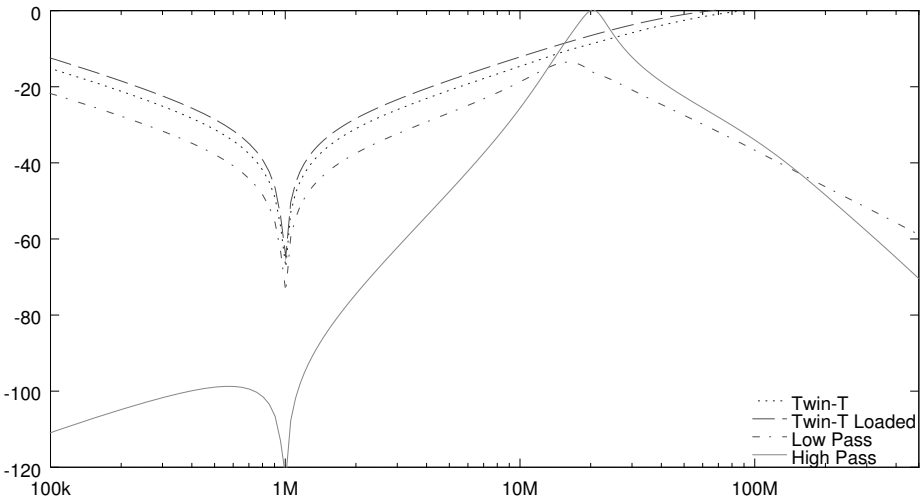

Figure 4.15.: Simulation outputs at each filter stage

4.1.2.6. Sensitivity analysis

Spice produces values p_{C_N} for every component attribute C_N by perturbing C_N and then performing an AC analysis:

$$p_{C_N} = \frac{V(C, C_N + \delta C_N) - V(C)}{\delta C_N}$$

Where $V(C)$ is the complex AC analysis voltage performed without perturbation, and C is the set of component attributes $C = \{C_1, C_2, ..., C_M\}$. Each component attribute is varied independently, using a constant value for δ that may be set by the user. In order to find an upper and lower bound of the filter output for all variations of component attribute it is necessary to minimise (or maximise) the function:

$$V(C + \Delta C) = V(C) + \sum_{N=1}^{M} p_N C_N \Delta_N$$

Where $\Delta = \{\Delta_1, \Delta_2, ..., \Delta_M\}$ is the set of perturbations for each component, scaled from $-T$ to T, for some user defined component tolerance, T. To find the worst case maximum and minimum of $V(C + \Delta C)$, it is necessary to perform the calculation for all permutations of Δ. The number of calculations required is $(2s)^M$, where s is the number of discrete steps between 0 and T. The filter has 17 resistors and capacitors, thus the number of calculations required is $(2s)^{17}$. It is not feasible to perform this analysis for any value of s other than unity. Using $s = 1$, the analysis will consider the worst case perturbation for all components.

A small application for *ngspice* was written to perform this analysis and produce Figure. 4.16 using 5% tolerance (top) and 1% tolerance (bottom) E12 resistor and capacitor values. The enlarged view of the transmitter filter gain at 21MHz is shown at the bottom right of Figure. 4.16. It can be seen that the worst case maximum gain is 1dB and the worst case minimum gain is -5dB. For practical filter realisation, it is necessary that the maximum gain be bound to less than approximately 1.5dB to avoid saturation in the operational amplifier. This condition is satisfied when 1% tolerance components are used.

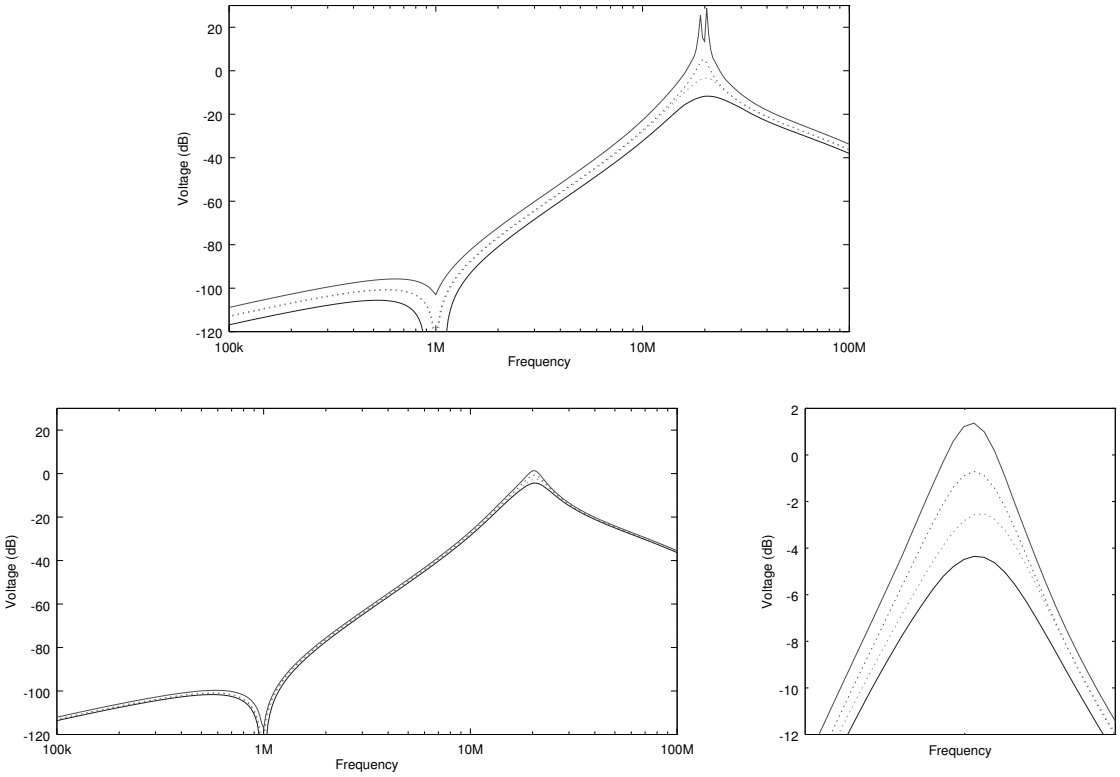

Figure 4.16.: Sensitivity analysis results.

4.1.3. Filter test results

The transmitter filter is implemented in the prototype transceiver using a Texas Instruments OPA2356 dual operational amplifier, with a gain bandwidth product of 200MHz. The output of the prototype transmit filter is shown in frequency domain in Figure. 4.17 and time domain in Figure. 4.18. The high pass filter is capable of keeping the relative power below the requirements of the standard in between 10MHz and the fundamental frequency 21MHz. Below 10MHz there are problems with parasitics, however the attenuation is 40dB down to DC.

Unfortunately the limitations of the operational amplifiers become evident above 60MHz, where the attenuation of the low pass filter is not sufficient. A future modification to the transceiver may use only passive elements to achieve the required

spectrum, followed by a buffer stage using an operational amplifier. This approach reduces the bandwidth requirements of the amplifier stages.

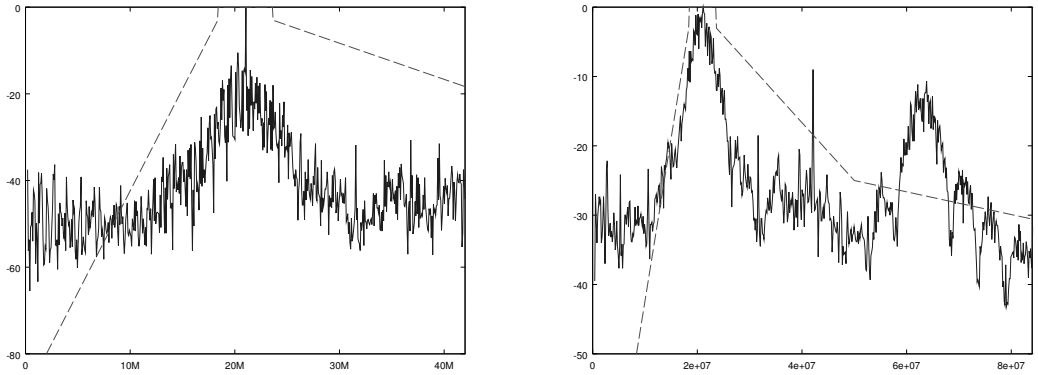

Figure 4.17.: Prototype transceiver relative output power spectrum

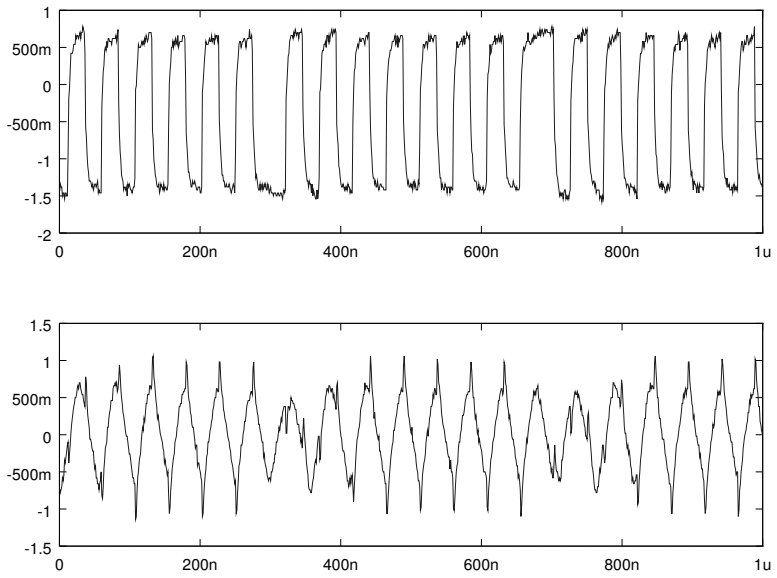

Figure 4.18.: Time domain output (bottom) of prototype transceiver compared with input (top)

4.2. Receiver Design

The receiver analogue interface consists of a gain stage, with automatic gain control (AGC) and a hard decision detector. Referring to the system schematic Figure. 5.2 U1 provides variable gain, with the comparator U2 responsible for detection.

The design around U1, an Analog Devices AD8367 variable gain amplifier (VGA) closely follows the reference design provided by the manufacturer, with the exception of an additional buffer U7, providing a stable DC offset for use by both the detector and the transmit filter. This DC offset allows for the operation of both the receiver and transmitter amplifiers using a single positive power supply of 3.3V. The other companion components connected to U2 consist of power supply filtering, AGC feedback filtering and input DC blocking. An additional capacitor is placed between the input terminal and ground in order to reduce high frequency components above 21 MHz.

The AD8367 is capable of 42.5dB gain up to 500MHz and has a nominal input impedance of 200Ω. Impedance matching is difficult in the case of capacitive coupling, due to variations in hydration and contact quality between the electrode and the user's skin, however 200Ω is within the range of 180Ω to 290Ω observed by Lucev et al. [38].

One of the advantages of using automatic gain control is the ability to acquire information on the attenuation level of the communications channel. The AGC feedback signal, present across C9 is sampled by the PSoC as described in Section. 5.1.4.

Following the gain stage is the detector, a Texas Instruments TLV3501 comparator, with a maximum toggle rate of 80MHz. The detector is DC biased at the same level as the VGA by an operational amplifier configured as a voltage follower, U7. The output of the comparator is fed directly to the data synchronisation logic of the FPGA as described in Chapter 3.

4.2.1. Receiver simulation

Without a Spice model of the Analog Devices AD8367 VGA it is not possible to accurately simulate the receiver input stages, however an approximation of the receiver was used during the phantom solution channel attenuation measurements for the paper "An Empirical Measurement of Signal Attenuation and BER of IEEE 802.15.6 HBC using a phantom solution" [30] written as part of this project. The approximation consisted of a fixed gain amplifier stage with clamping diodes limiting the output of the amplifier to the power supply rails. The schematic has been reproduced in Figure. 4.19.

The input data for the model was captured using a Tektronix MSO5204 10GS/s high speed digital oscilloscope connected to a receiver electrode suspended in a phantom solution. Also suspended in the solution was the prototype transmitter electrode, transmitting a known sequence of packets. The output of the TLV3501 from the Spice model was then imported into GHDL for simulation of the receiver synchronisation and demodulation logic.

As demonstrated in the paper, it was possible to correctly demodulate the transmitted data stream, demonstrating that the proposed hard decision receiver architecture, combined with the receiver digital design is sufficient for communications over the human body communications channel.

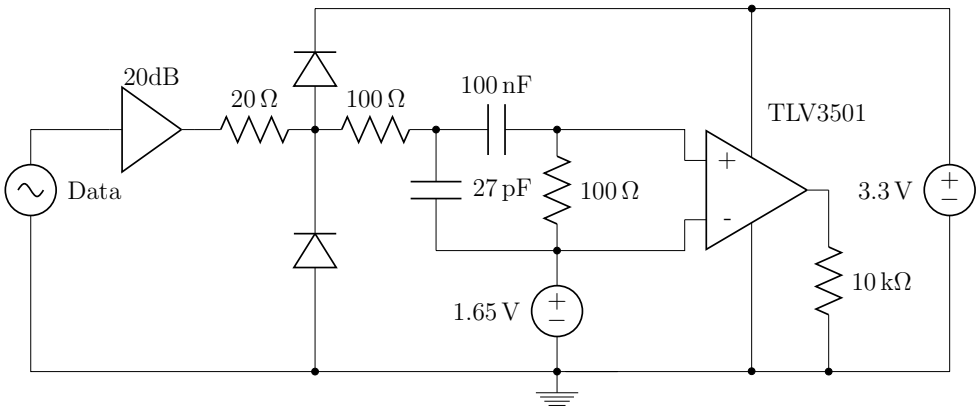

Figure 4.19.: Receiver analogue front end spice model as used in the paper "An Empirical Measurement of Signal Attenuation and BER of IEEE 802.15.6 HBC using a phantom solution"

4.3. Summary

This chapter demonstrates a filter architecture that is capable of meeting the requirements of the IEEE 802.15.6 transmit mask, as well as test results from a prototype filter that is capable of meeting compliance in the lower end of the spectrum. Further development is needed to ensure compliance beyond 60MHz. It is proposed that a passive low pass filter stage followed by a buffer amplifier will reduce the bandwidth requirements of the low pass filter operational amplifier. A hard decision receiver architecture based on a variable gain amplifier, followed by a comparator is presented. This architecture is implemented in the prototype transceiver with a discussion of performance in Section. 5.2.1. The combination of the digital transceiver core, analogue electronics and sundry requirements such as power supply and microcontroller are detailed in the following chapter, along with test results of a complete transceiver system.

5. HBC Transceiver Integration and Testing

The realisation of an effective transceiver design required the integration of both the analogue and digital components, as well as a suitable power supply and communications to a host device such as a laptop or mobile telephone. This chapter covers the design approach and construction of the prototype transceiver.

5.1. Design Considerations

The transceiver system consists of the following components:

1. Digital communications logic, as described in Chapter 3.
2. Analogue transmitter filter, as described in Chapter 4.
3. Analogue receiver electronics, also described in Chapter 4.
4. Microcontroller, used to integrate all components of the design as well as providing analogue to digital conversion of the receiver gain signal.
5. Packet storage DRAM.
6. FPGA configuration flash memory.
7. Crystal oscillator.
8. USB host interface.
9. Bluetooth host interface.

10. Battery.
11. Battery management and power supply, capable of providing all voltages required by the various system components.

In order to reduce the overall design burden, an off the shelf, minimalist FPGA board was utilised. After surveying several single board FPGA options, the FS609 from Silicon On Inspiration was selected. This PCB provides the FPGA, the crystal oscillator and the DRAM and flash memory. All power and I/O connections are routed through a PCI-Ex4 style edge connector. The selected board allowed for great flexibility in terms of power supply and I/O connectivity.

The remaining components have been placed on another PCB, referred to as the analogue front end PCB, that connects to the FPGA board using the edge connector. The rest of the chapter will focus mainly on the development of the hardware and software needed to meet the design requirements.

5.1.1. Power supply

There are several system voltages required by the system as shown in Table. 5.1. The current requirements have been taken from the integrated circuit data sheets, using worst case (maximum current) figures.

It is necessary that the required power supplies are stable regardless of battery charge state. The high energy density, reasonable cost and wide availability of Lithium-Ion (Li-Ion) technology makes it an ideal candidate for a mobile HBC transceiver. Single cell Li-Ion batteries typically vary from approximately 4.2V to 3.0V while discharging from full capacity to empty [39].

5.1.1.1. Battery charging

Battery charging is possible using the USB connector as shown in Figure. 5.1. A Microchip MCP73831 fully integrated Li-Ion single cell battery charge management IC (U4) is responsible for battery charging. The voltage at VBAT is regulated to

Table 5.1.: System voltages and currents.

Voltage	Use	Current requirement	Current capability	Type
1.25V	FPGA core	65mA	100mA	BJT
2.5V	FPGA I/O and DRAM	760mA	1A	SMPS
3.3V	Analogue transmit filter and receiver	60mA	250mA	Linear Regulator
1.65V	Analogue reference	<5mA	20mA	OpAmp
5V	PSoC analogue reference and 2.5V SMPS	<5mA	50mA	Boost Converter
3-5V	PSoC core	10mA	1A	Battery

a maximum of 4.20V by U4 with a charging current determined by R1. Selection of R1 is battery dependent. USB compliance is ensured by U4, as it will not allow current flow from VBAT to VUSB. The CHARGE_STAT signal is routed to the microcontroller to inform the user of battery state.

5.1.1.2. Voltage regulators

The two components with the largest energy requirements are the FPGA and the DRAM, both operating from the 2.5V supply. The Xilinx XPA power estimation tool calculates a worst case quiescent current of approximately 600mA. The DRAM worst case power consumption is approximately 160mA [40]. All other current requirements have been calculated by summing the power requirements of each integrated circuit and tabulated in Table. 5.1.

A switch mode power supply (SMPS) was chosen for the 2.5V supply due to its relatively large current requirements. The SMPS operates in Buck converter mode, reducing the current requirement from the battery supply. An off the shelf Texas

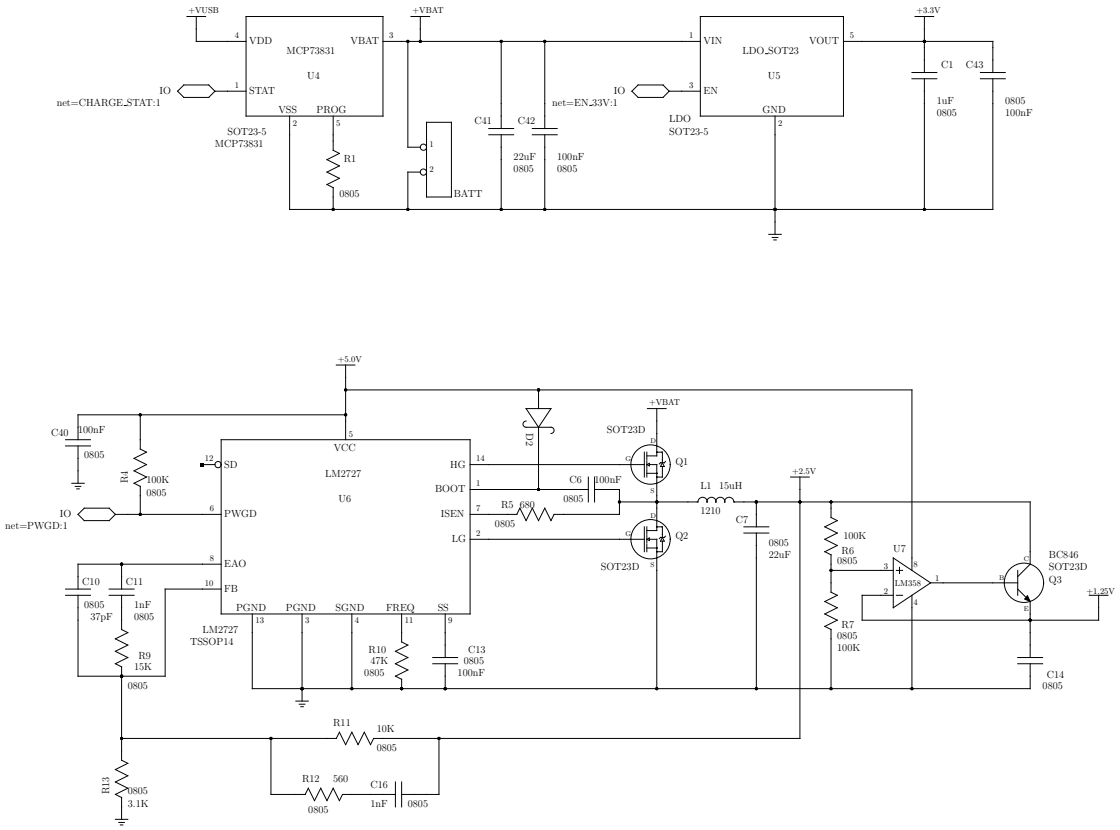

Figure 5.1.: Power supply schematic

Instruments LM2727 was chosen as the SMPS, with two N-Channel PMV31XN MOSFET transistors as the switching elements. The MOSFETs were chosen for their low on-state resistance (31mΩ) and minimal package size for the required current capacity. The Buck converter operates in synchronous mode, meaning that the fly-back diode has been replaced by a MOSFET (Q2), improving efficiency by eliminating the diode voltage drop.

The SMPS circuit is shown in figure Figure. 5.1. U6 is the LM2727 controller IC, Q1 and Q2 are the switching elements. L1 and C7 form the low pass filter. The feedback network consisting of C10-11, C16, R9, R11-13 is responsible for maintaining 2.5V at the output of the low pass filter. The filter design follows the application notes provided by Texas Instruments in the data sheet [41]. R10 is responsible for the operating frequency of the SMPS, with 47KΩ giving a switching frequency of

approximately 530KHz.

The linear regulator circuit consisting of Q1, U7(a) and R6-7, provides the 1.25V power supply. Q1 is capable of dissipating 250mW, but is current limited to 100mA. This is adequate given the 65mA requirement of the FPGA core.

A Texas Instruments LP5907 low drop-out linear voltage regulator (LDO), U5, provides the 3.3V supply directly from the battery. Given the small difference between the nominal battery supply voltage (3.7V) and the required 3.3V, a linear voltage regulator is able to operate with 88% efficiency. The use of a LDO allows for a minimum battery supply voltage of 3.42V. This voltage corresponds to a battery charge state of approximately 10%.

5.1.2. Analogue components

The analogue components are located closest to the edge connector on the PCB. They consist of an Analog Devices AD8367 variable gain amplifier (VGA) (U1), a Texas Instruments TL3501 comparator (U2), a Texas Instruments OPA2356 dual operational amplifier (U3) and the surrounding passive components. The design of the receiver and transmitter electronics are discussed in Chapter 4. Figure. 5.2 shows the schematic of the analogue electronics as implemented on the prototype transceiver PCB. The common net `HBC_PAD` is connected to the transceiver main electrode on the analogue interface PCB.

An additional connection has been provided to allow for testing without the influence of the transmit filter. This is shown on the schematic as the net `S_DATA_OUT_2`. Under normal circumstances, however the digital data stream from the FPGA transmit logic is directly connected to the transmit filter by the `S_DATA_OUT` net.

After amplification by U1, incoming packets from the transceiver electrode are detected by U2 before being routed directly to the data synchronisation stage of the FPGA through the net `S_DATA_IN`. The comparator U2 operates at 3.3V, however the FPGA input pins are 3.3V tolerant, and logic compatibility is ensured by selecting the appropriate voltage standard (LVCMOS33) when configuring the FPGA.

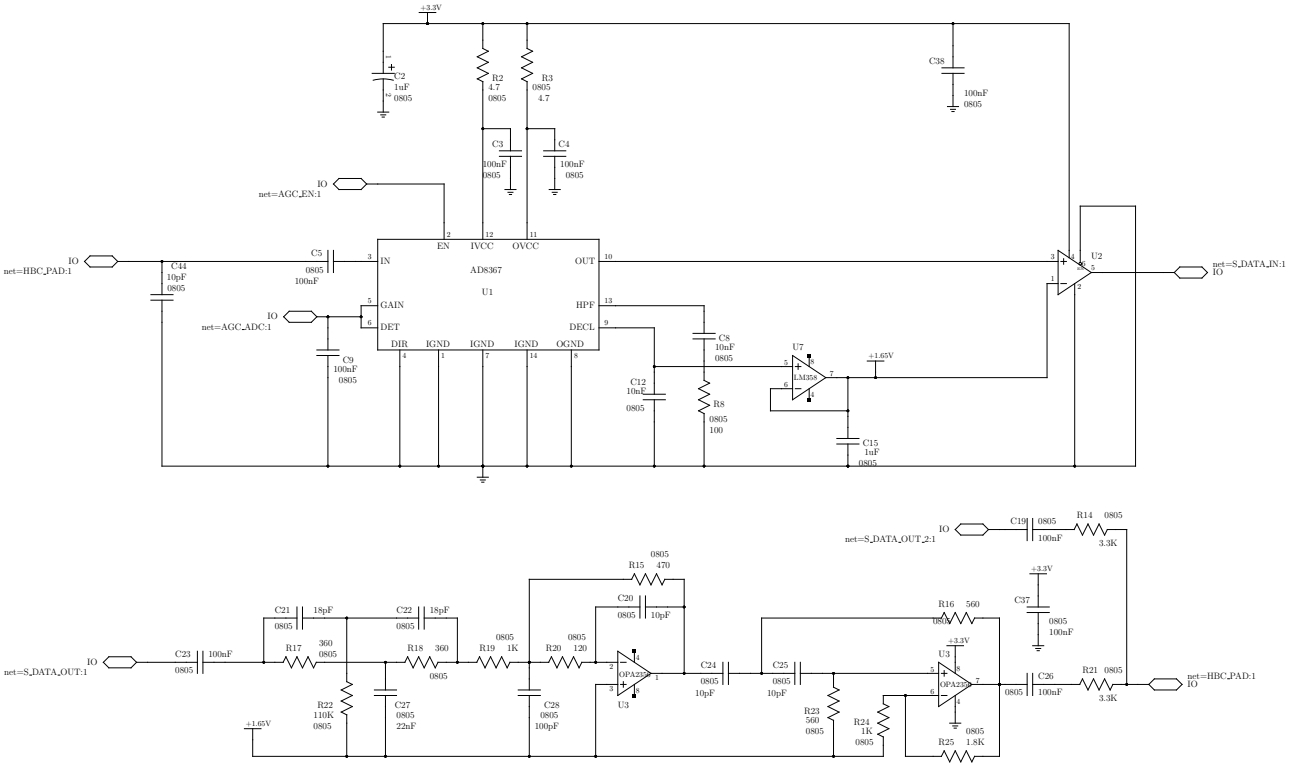

Figure 5.2.: Analogue electronics

5.1.3. FPGA

The interface between the Silicon On Inspiration FS609 Spartan 6 FPGA PCB and the analogue interface PCB is shown in Figure. 5.3. To reduce overall package size, an off the shelf PCI-Ex4 right angle adapter may be used allowing parallel alignment of the FPGA PCB and the analogue interface PCB.

The FS609 FPGA PCB contains the Xilinx Spartan 6 XC6LX9 FPGA, 32MB of DRAM, 512KB of Flash memory, a crystal oscillator and an LED. The Flash memory is arranged as 8 64KB sectors. 6 sectors are required for the FPGA configuration data, while one sector is required for the PSoC firmware and configuration. The remaining sector is not required and may be used by the selected application.

The 62.5MHz crystal oscillator is routed to the FPGA phase locked loop (PLL) which in turn supplies the 42MHz serial clock for HBC communications as well as

a 100MHz clock for the embedded Microblaze CPU and DRAM controller. The DRAM controller is implemented as FPGA logic using a core that was developed as part of this project.

In addition to the PCI-Ex4 connector, the FS609 FPGA board has a JTAG connector that is directly connected to the Spartan 6 FPGA. This connector is required when bootstrapping the PSoC firmware on the first power up after construction of the analogue interface PCB.

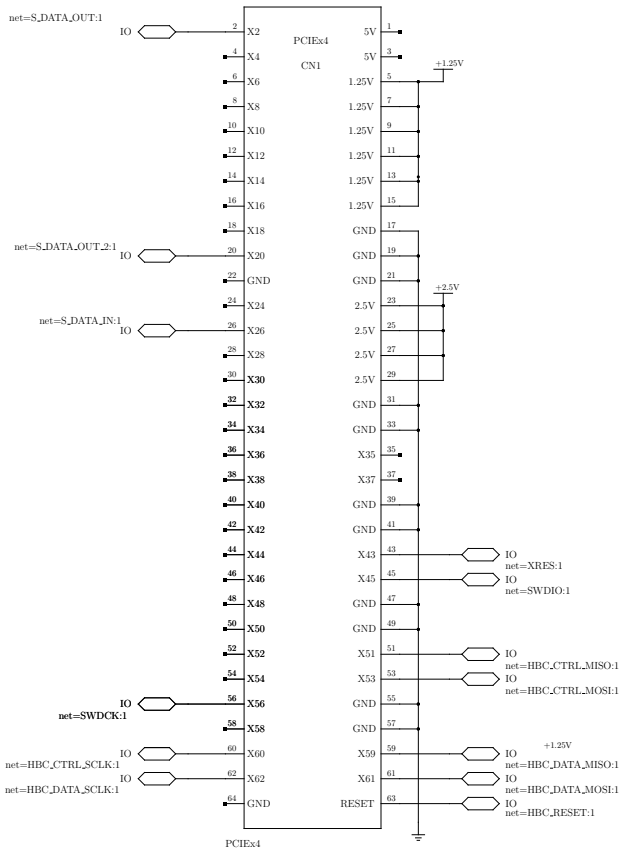

Figure 5.3.: FPGA Board Connector

A Xilinx Microblaze CPU is embedded into the FPGA and is responsible for packet framing, memory access, firmware upgrading and initialisation of the HBC transceiver modules. The FPGA configuration represents a complete transceiver solution that may be directly controlled by a host (the host is required for implementation of the

TCP/IP network stack). Communications between the FPGA and the host take place through two SPI slave interfaces. One interface is responsible for configuration and maintenance tasks, while the other transmits and receives IP packets, ready to be framed by the embedded CPU and transmitted over the HBC channel. The embedded CPU code is contained within the *hbc_mac* repository.

The 32MB of DRAM accessible to the CPU is configured as two ring buffers. One for each direction of communication. When packets are to be sent from the host over the HBC channel, they are first buffered into the DRAM. Each packet is preceded by a header describing the desired spreading factor, packet length and scrambler seed. The same is true for packets received on the HBC interface.

As soon as a packet has been received in full, it is first checked for the correct length and correct header CRC, before forwarding it over the data SPI interface to the host.

Packets to be transmitted arrive from the host as transport layer frames. For example, an application layer datagram will have both a UDP and IP header prepended to the data frame. Before transmission over the HBC interface, the embedded processor prepends the IEEE 802.15.6 HBC header. An example of a UDP/IP frame encapsulated in a IEEE 802.15.6 frame is shown in Figure. 5.4.

The FPGA embedded CPU firmware also contains the necessary code to program the management microcontroller firmware flash memory described in Section. 5.1.4, using the Cypress Semiconductor host sourced serial programming (HSSP) system. The FPGA logic contains a flash interface accessible by the embedded CPU to allow for both microcontroller firmware and FPGA configuration updates. This is controlled through the management SPI interface. The end user is therefore able to update all firmware either through the host USB connection, or using Bluetooth as described in Section. 5.2.2.

The embedded CPU source code has been embedded into this document as the repository `hbc_mac.tar.xz`. This repository may be extracted using the *7-zip* or *xz* utilities.

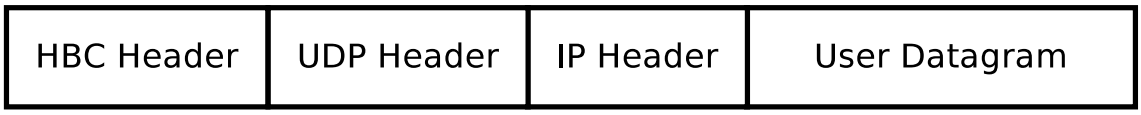

Figure 5.4.: Example of a UDP/IP frame encapsulated in a IEEE 802.15.6 frame

5.1.4. Communications and management microcontroller

A Cypress CY8C3245PVI-150 PSoC3 microcontroller in a SSOP48W package provides communications services between the HBC FPGA embedded CPU and a host machine. The PSoC3 was chosen due to its input/output (I/O) voltage flexibility, embedded USB device controller and low power consumption.

The PSoC provides 4 I/O interface buses, each providing logic voltage level translation between the embedded microcontroller core and the I/O pins. The 3.3V logic requirements of the Bluetooth module can be accommodated, along with the 2.5V FPGA interface and 5V power management signals from the LM2727 SMPS IC. Figure. 5.5 shows the interconnections between the microcontroller the Bluetooth, USB, power supply and receiver gain components.

The microcontroller provides a 12-bit Delta-Sigma analogue to digital converter (ADC) that is used to monitor the gain control feedback signal from the receiver VGA. This sampled signal may be transmitted to the host, allowing for channel attenuation measurements during HBC communications. Manual receiver gain control is also possible though the use of the on board 8-bit digital to analogue converter (DAC).

Another function provided by the PSoC is a boost converter based around D1 and L2 that provides the required 5V supply to the LM2727 SMPS IC. This boost converter is controlled by the PSoC firmware and provides a mechanism to disable the 2.5V FPGA supply. It is possible to reduce the power requirements as low as 10mA by disabling all on board power supplies with the exception of the PSoC core CPU.

The PSoC firmware is primarily responsible for the transfer of IP packet data between a host device (such as a computer or mobile telephone) and the HBC transceiver. The prototype transceiver provides this functionality through the use

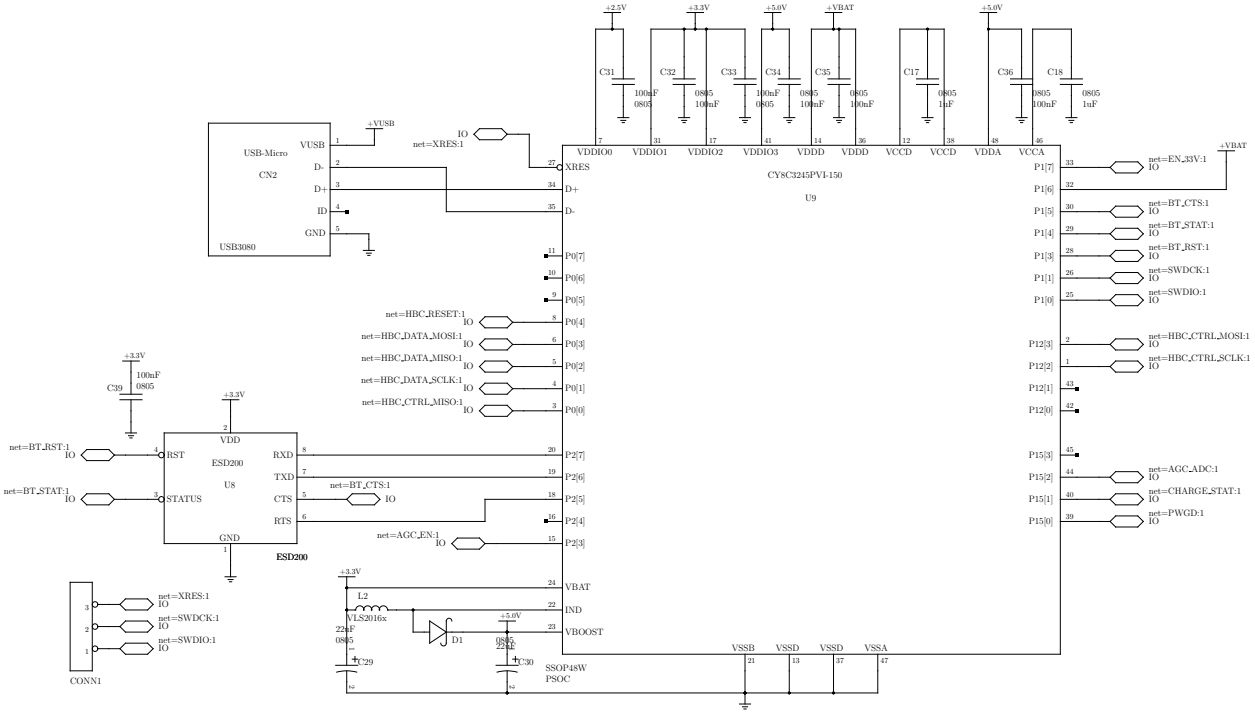

Figure 5.5.: Microcontroller schematic

of either a USB or Bluetooth channel, however the software has been written to accommodate any communications protocol that can support serial communications.

Secondary functions of the PSoC firmware include analogue to digital conversion (ADC) of the receiver automatic gain control (AGC) feedback signal, as well as power system and battery management. Other Maintenance tasks include firmware updates for both the PSoC and the Spartan 6 FPGA, as well as reporting of memory tests and diagnostics. All maintenance tasks may be accessed through *psoc_ctrl* command line application. The usage of the command is straight forward and is printed whenever the command is called without any arguments.

In order to maximise throughput from the FPGA SPI interface to the host interface, the firmware utilises the PSoC direct memory access (DMA) controller, with the CPU attending to interrupts. Interrupt sources include power management errors, SPI interface status, USB or Bluetooth endpoint management and ADC completion.

As well as in band packet communications with the host, error recovery is available

via the out of band communications mechanisms provided by USB CDC-ACM and Bluetooth RFCOMM. In the USB case, the CDC-ACM `SET_CONTROL_LINE_STATE` [42] request is monitored by the PSoC firmware. In the event of a time-out (possibly due to loss of synchronisation between the host software and the PSoC firmware) the host can reset communications using this mechanism.

Similarly in the Bluetooth case, loss of synchronisation (or any other error state) can be communicated to the PSoC firmware from the host by issuing a RFCOMM DISC (disconnect) [43] command.

The PSoC firmware source code is located in the *analogue_board* repository. When first installed, the PSOC must be programmed by the FPGA. Bootstrapping the entire system is possible through the FPGA JTAG interface. Once the firmware has been uploaded, maintenance access from either the USB or Bluetooth interfaces is possible, including firmware updates. The initial bootstrap process is only required after PCB assembly.

The PSoC firmware source code has been embedded into this document as the repository `analogue_board.tar.xz`. This repository may be extracted using the *7-zip* or *xz* utilities.

5.1.5. Analogue Front End PCB

The PCB layout for the analogue interface board is shown in Figure. 5.6. The transmit filter and receiver analogue electronics are located closest to the edge connector. For end user applications that don't require the PSoC for communications, the PCB layout may be truncated to include only these components, without changing the high frequency performance of the analogue stages.

The PCB layout also includes an exposed, gold plated electrode. This allows for immediate testing of the transceiver system on the body without any additional electrode hardware. Alternatively an external electrode may be connected to the PCB if this is not desired.

When combined with a right angle PCI-Ex4 adaptor, the analogue interface PCB

and FPGA PCB lie parallel, with ample space between for the Li-Ion battery pack. The whole system may be housed in a container 8cm long by 4.5cm wide, with a depth of 2cm.

The components were soldered to the gold plated PCB using lead free solder, ensuring safe handling and allowing direct contact with human skin. Figure. 5.7 shows the PCB fully populated with all components.

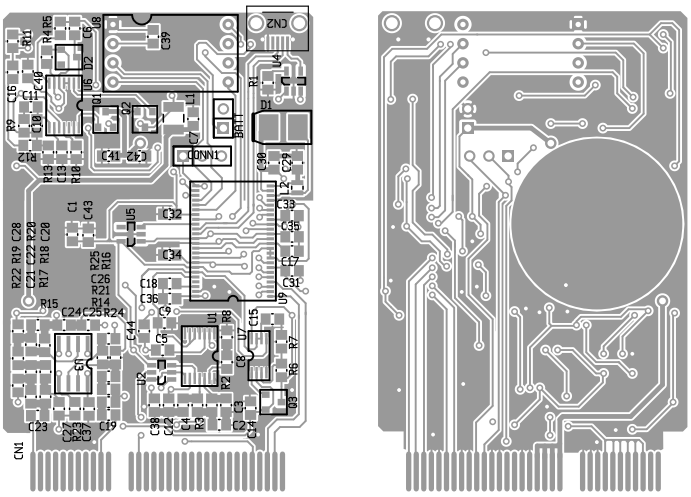

Figure 5.6.: PCB Layout for analogue interface board. Shown in actual size.

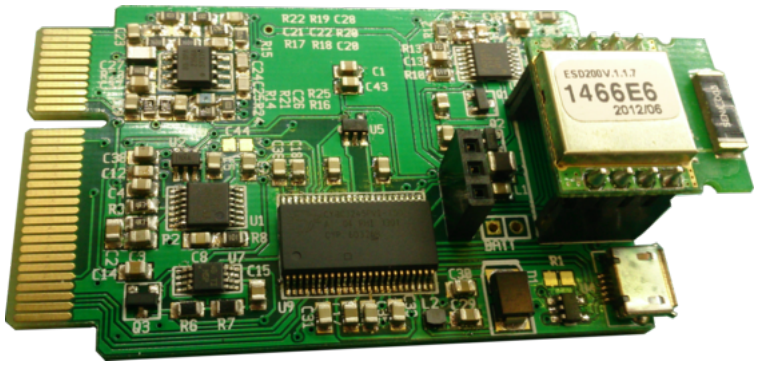

Figure 5.7.: Populated analogue front end PCB

5.2. System testing

Testing of the complete system has been conducted to characterise the performance of the prototype transceiver. This section focusses on combined testing of the analogue and digital electronics. Independent testing of the digital system is discussed in Section. 3.3.1. As shown in Section. 2.1 the communications channel consists of capacitive coupling between the transmitter and receiver electrodes, as well as capacitive coupling between the transceiver ground plane and the environment. In practise, it was found that the coupling between the ground planes increased significantly with either transceiver connected to any test equipment or power supply.

In order to remove the effect of additional ground coupling from test equipment, all tests have been conducted using batter power and a Bluetooth connection from the transceiver to a host computer.

5.2.1. Receiver sensitivity

Reading the receiver VGA gain control signal (shown in Figure. 5.2, `AGC_ADC`) using the ADC contained within the management microcontroller as described in Section. 5.1.4 allows for an estimate of the noise level present at the input of the receiver in the absence of any input signal. A plot of the gain vs control voltage of the AD8367 is shown in Figure. 5.8. In AGC configuration, the VGA maintains a constant output voltage level of 354mV RMS. Typically, with no signal present, a control voltage of 0.46V is observed, corresponding to a receiver gain of approximately 23dB or 25mV RMS of noise at the receiver input terminal. The receiver input impedance is 200Ω, so the noise power is -25dBm.

The bit error rate of BPSK modulation expressed in terms of power spectral density and energy per bit $\mathcal{E}_b$ is given by:

$$P_b = Q\left(\sqrt{\frac{2\mathcal{E}_b}{N_0}}\right) \tag{5.1}$$

where $Q(x) = 1 - \Phi(x)$ and $\Phi(x)$ is the cumulative distribution function of the Gaussian random variable x with mean 0 and unity variance.

The ratio $\mathcal{E}_b/N_0$ may be expressed in terms of the carrier power to noise power ratio C/N by observing that $\mathcal{E}_b = C/r_b$ and $N_0 = N/B$, where r_b is the BPSK bit rate and B is the receiver bandwidth.

The BPSK bit rate in IEEE 802.15.6 modulation is equal to the chip rate divided by the spreading factor and ranges from 656Kbps to 5.25Mbps. The large bandwidth of the receiver (approximately 200MHz) implies that the power spectral density N_0 of the noise is low. Under these conditions, 5.1 implies that very good performance may be achieved with -25dBm of noise.

Unfortunately, the noise at the receiver isn't white, resulting in a much higher power spectral density, and a significantly degraded BER. Attempting to observe the spectrum of the receiver amplifier output terminals proved difficult in practice, mainly due additional loading of the amplifier by the oscilloscope probes and increased feedback to the receiver input caused by coupling to the environment. The most significant noise sources on the prototype PCB include switching noise produced by the CPUs and power supplies, which is not white in nature.

It is expected that receiver sensitivity may be improved though two means. The first is the use of electromagnetic shielding around the analogue electronics in conjunction with improved PCB layout to minimise the coupling of CPU switching noise into the receiver amplifier. In addition to this a bandpass filter centred at 21MHz may significantly reduce noise power for noise sources that fall outside the IEEE 802.15.6 signal.

In practice it was found that the reduced receiver sensitivity resulted in very high bit error rates when using the transceiver platform on the body, without any additional ground coupling. For the on-body packet transfer tests carried out in Section. 5.2.3, the channel attenuation was reduced by using a ground coupling between the transmitter and receiver. This ground coupling consisted of a two lengths of wire approximately 30cm long twisted together. Each separate wire was electrically connected to each transceiver, however no electrical connection between the transceivers was

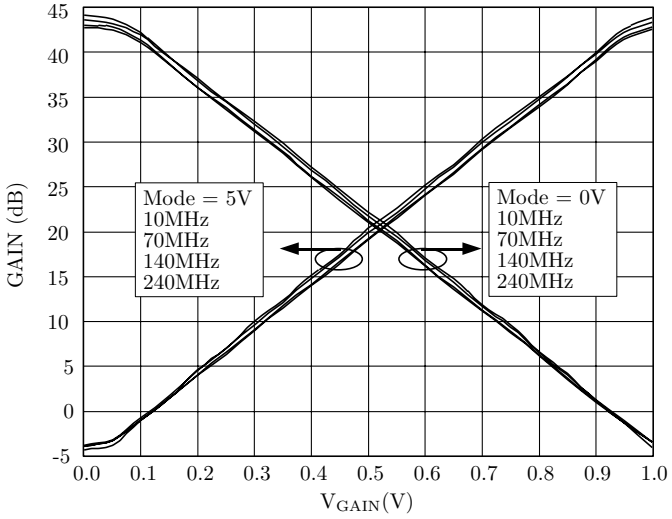

Figure 5.8.: AD8367 Gain vs control voltage[4]

present.

5.2.2. Test-bench software configuration

To enable full system testing using internet protocol (IP) packets, integration with the host system's IP software stack is required. For testing purposes, integration is achieved using link layer emulation. The use of link layer emulation allows the HBC transceiver to function as a normal network interface connected to a host device such as a laptop or smart phone. The HBC transceiver may then carry standard IP traffic across the body channel. All of the host software is maintained in the *psoc_host* repository. Communications between the PSoC and the host are possible using either a USB or Bluetooth interface.

The USB interface consists of a class compliant communications device class abstract control model (CDC-ACM) interface, using USB full speed bulk endpoints for full duplex communications. The Linux module *cdc-acm* exposes the connection as a Teletype (TTY) character device.

The Bluetooth interface uses the Bluetooth RFCOMM protocol, a connection oriented stream transport protocol providing reliable communications. By using this

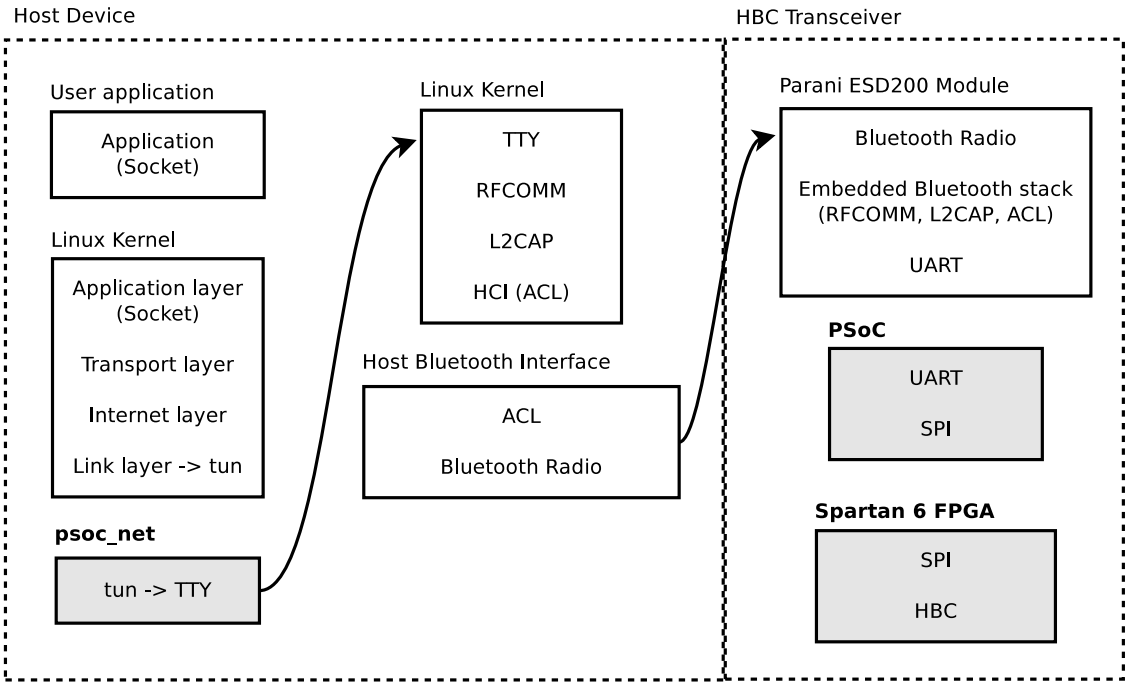

Figure 5.9.: Full protocol stack from application to physical layer. The shaded modules were developed as part of this project.

protocol, error correction is provided by the the Bluetooth layer, relieving the PSoC from this task. The Linux *rfcomm* module also exposes the communications link as a TTY character device. When using Bluetooth, reliability of the RF link between the host device and the transceiver must be considered. The RFCOMM protocol in use provides reliable communications as a consequence of the underlying Bluetooth Asynchronous Connection-oriented Logical transport (ACL) protocol [9], which retransmits dropped packets.

As both communications methods are abstracted as a TTY interface, a common control protocol may be implemented by the host software and the PSoC firmware. This protocol consists of a simple message format as defined in `host_protocol.h`, part of the *analogue_board* repository.

The Linux *tun* module provides a virtual link layer device by exporting IP packets to user-space. The user-space application *psoc_net,* part of the *psoc_host* is

responsible for transferring IP packets to the PSoC microcontroller, where they are subsequently forwarded onto the FPGA using the SPI interface. The FPGA firmware loads each packet into the DRAM of the HBC transceiver forming a queue that will be transmitted as soon as the channel is free. Received packets are similarly fed back from the HBC transceiver to the kernel through the *tun* device. The maximum transmission unit (MTU) of the forwarded packets is limited to 255 bytes, the same as the MTU of the IEEE 802.15.6 HBC protocol.

Figure. 5.9 shows the full protocol stack used by the transceiver when using Bluetooth communications between the host and the HBC transceiver. The protocol stack is transparent at the application layer of the IP stack. Reliable communications over the body channel may be provided by the IP transport layer (for example using TCP). The ACL interface between the host and the transceiver guarantees in-order delivery of packet data to the HBC transceiver.

The host software source code has been embedded into this document as the repository `psoc_host.tar.xz`. This repository may be extracted using the *7-zip* or *xz* utilities.

5.2.3. IP packet transfers

Complete testing of the entire transceiver is possible using the architecture shown in Figure. 5.10. Two separate hosts were configured with separate IP addresses. Each host was running an instance of the *psoc_net* application, communicating with one corresponding HBC transceiver using a Bluetooth link. The two prototype transceivers then complete the link using the human body as a communications channel. This configuration enables realistic testing as there are no electrical connections between the transceivers, however increased ground coupling was required as described in Section. 5.2.1.

The two laptops are able to communicate using IP packets following the protocol layer stack shown in Figure. 5.9. The standard operating system utility *ping* was used to demonstrate connectivity between the two systems as well as to provide statistics on round trip time and packet loss. The *ping* utility sends 56 byte ICMP

ECHO_REQUEST packets to a target host and measures the round trip time of the response packet. The response packets are also checked for data integrity and dropped if the response padding bytes do not match the request padding bytes.

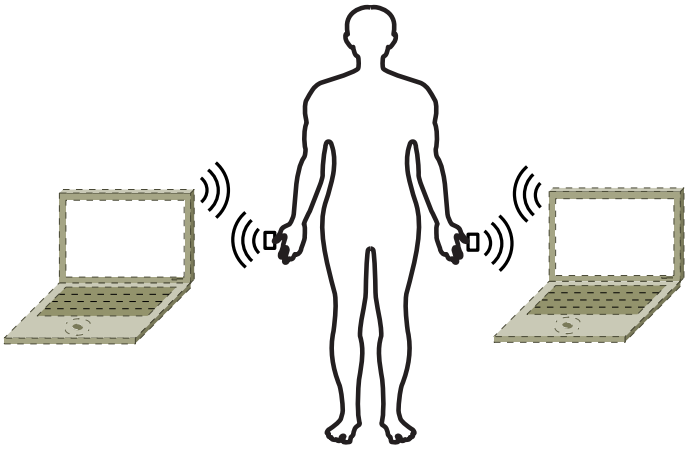

Figure 5.10.: Transceiver test configuration

In order to produce the results shown in Table. 5.2 the two transceivers were placed on a bench with the ground coupling connected. A human test subject was instructed to place their thumbs on each transceiver electrode pad on the analogue PCB shown in Figure. 5.6. The packet interval was set to 100ms and the *ping* utility was run for 1 minute. The round trip time includes any protocol delays in both the Bluetooth stack as well as the HBC layer.

Table 5.2.: Full system test results

Spreading factor	Average round trip time	Packet loss
64	221ms	21%
32	243ms	38%
16	238ms	45%
8	242ms	58%

Table. 5.2 shows an improvement in packet loss for higher spreading factors due to decreased BER. Packet loss figures represent packets dropped by the HBC transceiver

and packets that fail the integrity check on reception. Average round trip times do not significantly change with spreading factor. This result is a consequence of the time taken to process each packet through the SPI interface, the Bluetooth interfaces and the packet transfer time required by the PSoC microcontroller.

In addition to system round trip time, a sustained throughput test was conducted by transmitting 1MB of data, consisting of 8192 128 byte packets, over the communications channel in the same manner as described in Section. 3.3.1. In contrast to the preceding test, transmission is unidirectional. The results are presented in Table. 5.3. Table. 5.3 may be compared with Table. 3.3 to show the effect of the communications channel on BER. The results show decreased packet loss with higher spreading factors, consistent with Table. 5.2, but with lower error rates. The higher probability of error seen in Table. 5.2 is to be expected due to the combined risk of packet corruption for bidirectional transmission.

Table 5.3.: Transceiver BER for sustained transmission of 1MB

Spreading factor	Dropped packets	BER	Data rate (r_d)
64	518	0.063	147Kbps
32	873	0.107	280Kbps
16	1147	0.140	466Kbps
8	1693	0.207	763Kbps

5.3. Summary

In this chapter a complete transceiver is described that is capable of sending and receiving IEEE 802.15.6 packets. The system implements all of the physical layer and includes support for the lower level components of the IEEE 802.15.6 MAC layer, including packet framing, header generation and carrier sense. A test bench that allows for internet protocol packet encapsulation was described, along with

test results showing the packet loss and round-trip return time of standard ICMP ECHO_REQUEST packets using the industry standard *ping* utility that is present on most operating systems. The design incorporates all of the elements required to function as a stand alone embedded system, including power supply, battery management, USB and Bluetooth interfaces, non volatile flash memory and DDR RAM for temporary packet storage. A comparison of the system performance with the performance of the digital core demonstrates that the bit error rates of the system are closely related to receiver sensitivity and system noise. With further improvements to the analogue design the bit error rate may be reduced to meet the quality of service requirements for many body area networking applications. The combination of hardware descriptions enabling implementation of the physical layer of IEEE 802.15.6 using FPGA technology, and the basis of a system design for a complete transceiver system represents a significant step towards improving access to researchers interested in the development of IEEE 802.15.6 based body area networks.

6. Conclusion and Further Work

The transceiver architecture presented in this thesis meets the goal of providing a reference IEEE 802.15.6 transceiver that may be implemented using low cost FPGA hardware. The modular design provides all of the necessary components to test further improvements to IEEE 802.15.6, and may be used as the basis for further research into human body communications.

Complete hardware descriptions are provided for implementation of all of the required transmitter and receiver components of IEEE 802.15.6 including packet framing, header and preamble generation, modulation, receiver synchronisation, demodulation, forward error correction, rate detection, scrambling and descrambling and embedded processor integration. All hardware descriptions are accompanied by an associated testbench, and have been proven in hardware, meeting the timing requirements for a 42MHz chip rate.

6.1. Thesis review

Chapter 2 describes the communications channel and the details of the physical layer of IEEE 802.15.6. A discussion of the history of human body communications transceivers shows the evolution of the concept since the introduction of the idea by Zimmerman in 1995. A comparison of existing architectures is presented, noting that there does not exist an implementation for reconfigurable logic using FPGA technology. The motivation of the development of a reconfigurable platform is discussed, outlining the benefits such a system may offer the research community.

Chapter 3 discusses the hardware descriptions that enable implementation of all of the required transmitter and receiver components of IEEE 802.15.6 including packet framing, header and preamble generation, modulation, receiver synchronisation, demodulation, forward error correction, rate detection, scrambling and descrambling and embedded processor integration. All hardware descriptions are accompanied by an associated testbench, and have been proven in hardware, meeting the timing requirements for a 42MHz chip rate. A complete analogue and digital testbench allows for testing of receiver electronics, as well as channel modelling using *ngspice* simulation software, in conjunction with digital simulation using *GHDL*.

Chapter 4 presents a filter architecture to meet the IEEE 802.15.6 spectrum mask requirements, along with simulations showing compliance. Test results of an implementation of the filter show that the design performs adequately between 10MHz and 60MHz, with suggestions for improvement of performance above 60MHz. The receiver analogue front end, consisting of a variable gain amplifier and hard decision decoding is described. Simulations of the receiver architecture are presented in the associated paper, "An Empirical Measurement of Signal Attenuation and BER of IEEE 802.15.6 HBC using a phantom solution" [30].

A complete transceiver platform is shown in Chapter 5, incorporating the digital and analogue transceiver components as well as the power supply, USB and Bluetooth host interfaces, and memory. The system may be used as a tool for research and experiments in the field of body area networking using the IEEE 802.15.6 standard.

6.2. Design challenges

Throughout the development of a prototype IEEE 802.15.6 transceiver system, a number of design challenges have emerged that may be addressed in future revisions of the transceiver. This section describes some of these challenges and possible improvements that may be applied to the transceiver to improve throughput and noise immunity.

6.2.1. Digital system

At a spreading factor of 64 chips per bit, the digital system is capable of achieving a sustained throughput of 93% of the maximum achievable bit rate as shown in Table. 5.3. For lower spreading factors, the sustained throughput decreases due to the inter-packet processing time required by the embedded CPU.

This performance may be improved through by applying the design approach discussed in Section. 3.1.1. The present transmitter implementation uses the embedded CPU to construct the header of each packet, as well as CRC calculation. Implementation of dedicated hardware for these tasks will reduce the inter-packet time by reducing the amount of work required by the embedded CPU.

6.2.2. Analogue system

Improvement of the analogue receiver sensitivity is required to enable reliable communications without the use of augmented ground coupling. As discussed in Section. 5.2.1 several approaches may be implemented to reduce the noise level at the receiver. Electromagnetic interference shielding can reduce the coupling between the sensitive receiver electronics and system noise sources, such as CPU switching and power supply switching. Improved PCB layout and the use of additional PCB layers may also be applied to reduce the coupling between system noise and the receiver amplifier. Noise sources that exist outside the IEEE 802.15.6 signal spectrum may be filtered by a band pass filter at the input of the receiver amplifier.

The transmitter filter test results shown in Section. 4.1.3 demonstrate IEEE 802.15.6 compliant performance in the region between 10MHz and 21MHz, however higher frequencies are not adequately attenuated do to the limited gain-bandwidth product of the operational amplifiers used. The use of passive electronics to implement the filter, followed by a buffer amplifier would reduce the amount of high frequency power at the amplifier input and reduce the bandwidth requirements of the output amplifier.

Despite these limitations, the analogue system demonstrates that HBC communications are feasible and that reasonable throughput and BER can be achieved using off the shelf components, allowing for straightforward implementation of IEEE 802.15.6 transceivers for research purposes.

6.3. Further work

The hardware descriptions and simulation environment presented in this thesis allow for the testing and characterisation of performance improvements to IEEE 802.15.6, as well as analysis of the nature of the communications channel using a realistic measurement environment that is free of ground connections.

It has been observed throughout the testing phase of this thesis that the performance of the transceiver is highly dependent on the ground coupling between transmitter and receiver. In practical applications the user of a HBC system will be mobile and constantly interacting with changing environments. The availability of a relatively small, battery power transceiver with channel attenuation measurement enables dynamic measurement of the HBC channel. This data may be used to provide a fading model for HBC communications, enabling further research into methods to improve throughput when transmitter-receiver ground coupling is poor.

Another area of improvement that has been identified in Section. 2.2.4 is the possibility of better throughput and BER through the use of alternative binary linear codes. For example, the $[16, 5, 8]$, $[30, 8, 14]$ and $[26, 8, 12]$ codes all offer improved throughput while providing the same or better noise immunity. The modular transceiver architecture presented in this thesis may be modified to test these codes in simulation or hardware.

In addition to implementation of the physical layer, IEEE 802.15.6 describes network architecture requirements, multiple access requirements and security requirements that form the full protocol stack. While this work is outside the scope of this thesis, it is required in order to fully implement a body area network architecture as described by IEEE 802.15.6. The security, multiple access and network architecture

components are common across the three physical layer technologies, HBC, ultra-wideband RF and narrowband RF. A common protocol stack may be implemented in software to accommodate all physical layers.

Bibliography

[1] T. Zimmerman, "Personal Area Networks: Near-field intrabody communication," *IBM Systems Journal*, vol. 35, no. 3.4, pp. 609–617, 1996.

[2] "IEEE Standard for Local and metropolitan area networks - Part 15.6: Wireless Body Area Networks," *IEEE Std 802.15.6-2012*, pp. 1–271, Feb. 2012.

[3] H. Cho, H. Lee, J. Bae, and H.-J. Yoo, "A 5.2 mW IEEE 802.15.6 HBC Standard Compatible Transceiver With Power Efficient Delay-Locked-Loop Based BPSK Demodulator," *IEEE Journal of Solid-State Circuits*, vol. 50, pp. 2549–2559, Nov. 2015.

[4] Analog Devices, Inc., "AD8367 - 500mhz, Linear-in-dB VGA with AGC Detector," 2005.

[5] Olly Bootle, "Monitor Me," Aug. 2013. http://www.bbc.co.uk/programmes/b038p1pm.

[6] D. Lewis, "802.15.6 proposed applications," 2008. https://mentor.ieee.org/802.15/dcn/08/15-08-0407-06-0006-tg6-applications-summary.doc.

[7] GlobalData, "Mobile Technology 'Modernizes' Healthcare Sector as Gadgets Support Patients With Chronic Conditions," Apr. 2012. http://healthcare.globaldata.com/media-center/press-releases/medical-devices/mobile-technology-modernizes-healthcare-sector-as-gadgets-support-patients-with-chronic-conditions.

[8] Dynastream Innovations Inc., "ANT Message Protocol and Usage," 2014. https://www.thisisant.com.

[9] Bluetooth SIG Proprietary, "Bluetooth Specification Version 4.2," 2014.

[10] "IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)," *IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006)*, pp. 1–314, Sept. 2011.

[11] M. Seyedi, B. Kibret, D. Lai, and M. Faulkner, "A Survey on Intrabody Communications for Body Area Network Applications," *Biomedical Engineering, IEEE Transactions on*, vol. 60, pp. 2067–2079, Aug. 2013.

[12] J. Bae, K. Song, H. Lee, H. Cho, and H.-J. Yoo, "A Low-Energy Crystal-Less Double-FSK Sensor Node Transceiver for Wireless Body-Area Network," *Solid-State Circuits, IEEE Journal of*, vol. 47, pp. 2678–2692, Nov. 2012.

[13] M. Wegmueller, A. Kuhn, J. Froehlich, M. Oberle, N. Felber, N. Kuster, and W. Fichtner, "An Attempt to Model the Human Body as a Communication Channel," *Biomedical Engineering, IEEE Transactions on*, vol. 54, pp. 1851–1857, Oct. 2007.

[14] J. G. Proakis and M. Salehi, *Communication systems engineering*. Upper Saddle River, N.J: Prentice Hall, 2002.

[15] I. Bouyukliev and D. B. Jaffe, "Optimal binary linear codes of dimension at most seven.," *Discrete Mathematics*, vol. 226, pp. 51 – 70, 2001.

[16] I. Bouyukhev, D. B. Jaffe, and V. Vavrek, "The smallest length of eight-dimensional binary linear codes with prescribed minimum distance," *IEEE Transactions on Information Theory*, vol. 46, pp. 1539–1544, July 2000.

[17] K. Partridge, B. Dahlquist, A. Veiseh, A. Cain, A. Foreman, J. Goldberg, and G. Borriello, "Empirical measurements of intrabody communication performance under varied physical configurations," in *Proceedings of the 14th annual ACM symposium on User interface software and technology*, pp. 183–190, ACM, 2001.

[18] K. Hachisuka, A. Nakata, T. Takeda, Y. Terauchi, K. Shiba, K. Sasaki, H. Hosaka, and K. Itao, "Development and performance analysis of an intrabody communication device," in *TRANSDUCERS, Solid-State Sensors, Actuators and Microsystems, 12th International Conference on, 2003*, vol. 2, pp. 1722–1725 vol.2, June 2003.

[19] S. J. Song, N. Cho, S. Kim, J. Yoo, S. Choi, and H. J. Yoo, "A 0.9v 2.6mw Body-Coupled Scalable PHY Transceiver for Body Sensor Applications," in *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pp. 366–609, Feb. 2007.

[20] H.-i. Park, I.-g. Lim, S. Kang, and W.-W. Kim, "Human body communication system with FSBT," in *Consumer Electronics (ISCE), 2010 IEEE 14th International Symposium on*, pp. 1–5, June 2010.

[21] Chang-Hee Hyoung, Sung-Weon Kang, Seong-Ook Park, and Youn-Tae Kim, "Transceiver for Human Body Communication Using Frequency Selective Digital Transmission.," *ETRI Journal*, vol. 34, no. 2, p. 216, 2012.

[22] Y.-T. Lin, Y.-S. Lin, C.-H. Chen, H.-C. Chen, Y.-C. Yang, and S.-S. Lu, "A 0.5-V Biomedical System-on-a-Chip for Intrabody Communication System," *Industrial Electronics, IEEE Transactions on*, vol. 58, pp. 690–699, Feb. 2011.

[23] J. F. M. Gerrits, M. H. L. Kouwenhoven, P. R. Van Der Meer, J. R. Farserotu, and J. R. Long, "Principles and limitations of ultra-wideband FM communications systems," *Eurasip Journal on Applied Signal Processing*, vol. 2005, no. 3, pp. 382–396, 2005. http://dx.doi.org/10.1155/ASP.2005.382.

[24] K. Shikada and J. Wang, "Development of human body communication transceiver based on impulse radio scheme," in *CPMT Symposium Japan, 2012 2nd IEEE*, pp. 1–4, Dec. 2012.

[25] R. Xu, W. C. Ng, H. Zhu, H. Shan, and J. Yuan, "Equation Environment Coupling and Interference on the Electric-Field Intrabody Communication Channel," *IEEE Transactions on Biomedical Engineering*, vol. 59, pp. 2051–2059, July 2012.

[26] T. Kang, I. Lim, J. Hwang, C. Hyoung, H. Park, and S. Kang, "A Method of Increasing Data Rate for Human Body Communication System for Body Area Network Applications.," *2012 IEEE Vehicular Technology Conference (VTC Fall)*, p. 1, 2012.

[27] H. Park, I. Lim, S. Kang, and W.-w. Kim, "10mbps human body communication

SoC for BAN," in *Very Large Scale Integration (VLSI-SoC), 2015 IFIP/IEEE International Conference on*, pp. 149–153, Oct. 2015.

[28] Y. L. Tsou, C. S. A. Gong, N. C. Cheng, Y. Lee, and C. F. Jou, "Integrated Biosensing Platform Based on a 1.74-mW -90-dBm Sensitivity Dual- Mode-Operation Receiver for IEEE 802.15.6 Human Body Communication Standard," *IEEE Sensors Journal*, vol. 15, pp. 3317–3327, June 2015.

[29] Nick Sawyer, "XAPP225: Data to Clock Phase Alignment," Feb. 2009. www.xilinx.com.

[30] K. Taylor and D. Lai, "An Empirical Measurement of Signal Attenuation and BER of IEEE 802.15.6 HBC using a phantom solution," in *Proceedings of the 10th International Conference on Body Area Networks*, Bodynets '15, (Sydney, Australia), 2015.

[31] V. Sklyarov and I. Skliarova, "Digital Hamming weight and distance analyzers for binary vectors and matrices," *Int. Journal of Innovative Computing, Information and Control*, vol. 9, no. 12, pp. 4825–4849, 2013.

[32] L. D. Paarmann, *Design and analysis of analog filters. a signal processing perspective.* SECS: 617, Boston : Kluwer Academic Publishers, 2001.

[33] Paulo Neis and Doug Stewart, "Octave Forge ellip()." http://octave.sourceforge.net/signal/index.html.

[34] A. Bolle, "Theory of twin-T RC-networks and their application to oscillators," *Radio Engineers, Journal of the British Institution of*, vol. 13, pp. 571–587, Dec. 1953.

[35] T. Lazear and A. Rosenstein, "Pole-Zero Synthesis and the General Twin-T," *Applications and Industry, IEEE Transactions on*, vol. 83, pp. 389–393, Nov. 1964.

[36] A. Barker and A. Rosenstein, "S-Plane Synthesis of the Symmetrical Twin-T Network," *Applications and Industry, IEEE Transactions on*, vol. 83, pp. 382–388, Nov. 1964.

[37] Various, "ngspice." http://ngspice.sourceforge.net/.

[38] Z. Lucev, I. Krois, and M. Cifrek, "A Capacitive Intrabody Communication Channel from 100 kHz to 100 MHz," *Instrumentation and Measurement, IEEE Transactions on*, vol. 61, pp. 3280–3289, Dec. 2012.

[39] SANYO Energy Corporation, "UPF454261 Lithium Ion Battery Data Sheet," 2012.

[40] Samsung Electronics, "256mb J-die DDR SDRAM Specification," 2008.

[41] Texas Instruments, "LM2727/LM2737 N-Channel FET Synchronous Buck Regulator Controller for Low Output Voltages," 2013.

[42] USB Implementers Forum, Inc., "Universal Serial Bus Class Definitions for Communications Devices," 2010.

[43] Bluetooth SIG Proprietary, "RFCOMM with TS 07.10," 2012.

A. Hardware description code

Listing A.1 gives a graphical view of the HDL hierarchy, as well as displaying the purpose of each HDL module. Modules surrounded by angled brackets denote IP cores that were not developed as part of this project. All other modules are original work and are licensed using the GNU public license (GPL).

Listing A.1: FPGA SoC HDL Hierarchy

```
toplevel.vhd
    ->  cpu_clk                                 <pll_mem.vhd>
    ->  serial_clk                              <pll_serial.vhd>
    ->  Embedded CPU                            <mcs_0.xco>
    ->  CPU/DRAM interface                      (mem_interface.vhd)
    ->  DDR Controller                          (ddr.vhd)
    ->  HBC TX System                           (hbc_tx.vhd)
        ->  CPU/Modulator bus arbitration       (fifo_bus_arbitrator.vhd)
        ->  CPU/TX FIFO interface               (tx_fifo_interface.vhd)
        ->  TX FIFO                             <fifo_tx.xco>
        ->  Modulator                           (modulator.vhd)
            ->  Walsh encoder                   (walsh_encode_lut.vhd)
        ->  TX shift register                   (parallel_to_serial.vhd)
    ->  HBC RX System                           (hbc_rx.vhd)
        ->  CPU/RX FIFO interface               (rx_fifo_interface.vhd)
        ->  RX FIFO                             <fifo_rx.xco>
        ->  Input data to clock synchronisation (data_synchroniser.vhd)
        ->  RX State machine and control        (serial_to_parallel.vhd)
            ->  Phase alignement                (phase_align.vhd)
                ->  Preamble detector           (hamming_lut.vhd)
            ->  SFD detector                    (hamming_lut.vhd)
            ->  Walsh decoder                   (walsh_decoder.vhd)
                ->  Walsh encoder               (walsh_encode_lut.vhd)
                ->  Hamming distance logic      (hamming_lut.vhd)
    ->  Scrambler                               (scrambler.vhd)
    ->  SPI/Host interface                      (spi_interface.vhd)
        ->  Data interface                      <spi_slave_core.vhd>
        ->  Command interface                   <spi_slave_core.vhd>
    ->  PSoC memory programming interface       (psoc_interface.vhd)
    ->  Flash memory interface                  (flash_interface.vhd)
        ->  Flash SPI core                      <spi_master_core.vhd>
    ->  Timing constraints file                 (timing.ucf)
    ->  Pin map constraints file                (pin_map_spartan.vhd)
```

A.1. Hardware description modules

Complete listings of the core transceiver modules follow. Many other modules are required for system integration and are included in the repository `hardware_desc.tar.xz`. This repository has been embedded into this document.

```
#include <preprocessor/constants.vhh>

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library transceiver;
use transceiver.bits.all;

entity scrambler is
    port (
        cpu_clk, reset : in std_logic;
        reseed, seed_val, seed_clk : in std_logic;
        io_addr : in vec8_t;
        io_d_out : out vec32_t;
        io_addr_strobe : in std_logic;
        io_read_strobe : in std_logic;
        io_ready : out std_logic);
end entity scrambler;

architecture scrambler_arch of scrambler is

    constant SCRAMBLER_SEED_0 : vec32_t := X"69540152";
    constant SCRAMBLER_SEED_1 : vec32_t := X"8A5F621F";

    constant SCRAMBLER_ADDR  : vec8_t := HEX(SCRAMBLER_ADDR);

    signal io_addr_reg : vec8_t;
    signal enabled : std_logic;
    signal do_ack : std_logic;

    signal scram_reg : vec32_t;

    signal inc_scram : std_logic;
    signal scram_update : std_logic;

begin

    scram_update <= seed_clk or inc_scram;

    update_scram : process (scram_update)
        variable scram_reg_i : vec32_t;
        variable tmp_bit : std_logic;
    begin

        if scram_update'event and scram_update = '1' then
            if reseed = '1' then
                if seed_val = '1' then
                    scram_reg <= SCRAMBLER_SEED_1;
                else
                    scram_reg <= SCRAMBLER_SEED_0;
                end if;
            else
                scram_reg_i := scram_reg;
                for i in 31 downto 0 loop
                    -- Polynomial is z^32 + z^31 + z^11 + 1
```

```
                    tmp_bit :=  scram_reg_i(32 - 11) xor
                                scram_reg_i(32 - 31) xor
                                scram_reg_i(32 - 32);
                    scram_reg_i := shift_right(scram_reg_i, 1);
                    scram_reg_i(31) := tmp_bit;
                end loop;
                scram_reg <= scram_reg_i;
            end if;
        end if;
    end process update_scram;

    -- Get IO data
    io_proc : process(cpu_clk, reset) begin
        if reset = '1' then
            do_ack <= '0';
        -- Read IO bus on falling edge
        elsif cpu_clk'event and cpu_clk = '0' then
            if io_read_strobe = '1' then
                do_ack <= '1';
            else
                do_ack <= '0';
            end if;
        end if;
    end process io_proc;

    -- ACK process
    ack_proc : process(cpu_clk, reset) begin
        if reset = '1' then
            io_ready <= '0';
            io_d_out <= (others => 'Z');
            inc_scram <= '0';
        elsif cpu_clk'event and cpu_clk = '0' then
            if enabled = '1' then
                if do_ack = '1' then
                    io_ready <= '1';
                    io_d_out <= scram_reg;
                    inc_scram <= '1';
                else
                    io_ready <= '0';
                    io_d_out <= (others => 'Z');
                    inc_scram <= '0';
                end if;
            end if;
        end if;
    end process ack_proc;

    -- Get address from IO bus
    get_io_addr : process(cpu_clk, reset) begin
        if reset = '1' then
            io_addr_reg <= (others => '0');
        elsif cpu_clk'event and cpu_clk = '0' then
            if io_addr_strobe = '1' then
                io_addr_reg <= io_addr;
            end if;
        end if;
    end process get_io_addr;

    -- Assert enabled
    with io_addr_reg (7 downto 0) select enabled
        <=  '1' when SCRAMBLER_ADDR,
            '0' when others;

end architecture scrambler_arch;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library transceiver;
use transceiver.numeric.all;

library unisim;
use unisim.vcomponents.all;

entity walsh_encode_lut is
    port (
        symbol : in walsh_sym_t;
        walsh : out walsh_code_t);
end entity walsh_encode_lut;

architecture walsh_encode_lut_arch of walsh_encode_lut is

begin

    walsh_low_lut : RAM16X8S
        generic map (
            INIT_00 => X"9669",
            INIT_01 => X"3CC3",
            INIT_02 => X"5AA5",
            INIT_03 => X"F00F",
            INIT_04 => X"6699",
            INIT_05 => X"CC33",
            INIT_06 => X"AA55",
            INIT_07 => X"00FF")
        port map (
            O => walsh(7 downto 0),
            A0 => symbol(0),
            A1 => symbol(1),
            A2 => symbol(2),
            A3 => symbol(3),
            D => X"00",
            WCLK => '0',
            WE => '0');

    walsh_high_lut : RAM16X8S
        generic map (
            INIT_00 => X"6969",
            INIT_01 => X"C3C3",
            INIT_02 => X"A5A5",
            INIT_03 => X"0F0F",
            INIT_04 => X"9999",
            INIT_05 => X"3333",
            INIT_06 => X"5555",
            INIT_07 => X"FFFF")
        port map (
            O => walsh(15 downto 8),
            A0 => symbol(0),
            A1 => symbol(1),
            A2 => symbol(2),
            A3 => symbol(3),
```

```
            D => X"00",
            WCLK => '0',
            WE => '0');

end architecture walsh_encode_lut_arch;
```

```
#include <preprocessor/constants.vhh>

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library transceiver;
use transceiver.numeric.all;
use transceiver.bits.all;

entity modulator is
    port (
        clk, reset  : in std_logic;
        io_addr     : in vec8_t;
        io_d_in     : in vec32_t;
        io_addr_strobe : in std_logic;
        io_write_strobe : in std_logic;
        io_ready    : out std_logic;
        bus_master  : out std_logic;
        sub_addr_out: out vec8_t;
        sub_d_out   : out vec32_t;
        sub_addr_strobe : out std_logic;
        sub_write_strobe : out std_logic;
        sub_io_ready : in std_logic;
        fifo_almost_full : in std_logic);
end modulator;

architecture modulator_arch of modulator is

    -- FIFO_ADDR must be word aligned
    constant MODULATOR_ADDR : vec8_t := HEX(MODULATOR_ADDR);
    constant MODULATOR_SF_ADDR : vec8_t := HEX(MODULATOR_SF_ADDR);
    constant FIFO_ADDR : vec8_t := X"00";

    constant BIT_MAP_32_1 : vec32_t := X"AAAAAAAA";
    constant BIT_MAP_32_0 : vec32_t := X"55555555";

    constant DOUBLE_WRITE : vec8_t := X"00";

    constant SYM_LIMIT : integer := 32 / WALSH_SYM_SIZE;

    signal io_addr_reg : vec8_t;
    signal io_d_in_r : vec32_t;
    signal sf : vec8_t;

    signal enabled : std_logic;
    signal set_sf_op : std_logic;
    signal got_write : std_logic := '0';
    signal do_ack : std_logic;
    signal walsh_load : std_logic;
    signal walsh_shift : std_logic;
    signal symbol_load : std_logic;
    signal symbol_shift : std_logic;
    signal io_ready_s, io_ready_d : std_logic;

    type state_type is (st_idle,
```

```
                        st_load,
                        st_fifo_write,
                        st_fifo_ack,
                        st_shift,
                        st_bus_ack);
    signal state, state_i : state_type;

    constant BIT_COUNTER_BITS : natural := bits_for_val(WALSH_CODE_SIZE - 1);
    constant SYM_COUNTER_BITS : natural := bits_for_val(SYM_LIMIT - 1);
    signal symbol_count, symbol_count_i : unsigned(SYM_COUNTER_BITS-1 downto 0);
    signal bit_count, bit_count_i : unsigned(BIT_COUNTER_BITS-1 downto 0);
    signal repeat, repeat_i : std_logic;

    signal walsh_data, walsh_data_r : walsh_code_t;
    signal walsh_bit : std_logic;
    signal symbol : walsh_sym_t;
    signal symbol_r : vec32_t;

begin
    io_ready <= io_ready_s or io_ready_d;

    walsh_bit <= walsh_data_r(0);
    symbol <= symbol_r(WALSH_SYM_SIZE - 1 downto 0);

    walsh_encoder : entity work.walsh_encode_lut
        port map (
            symbol => symbol,
            walsh => walsh_data);

    walsh_shifter : process (clk) begin
        if clk'event and clk = '1' then
            if walsh_load = '1' then
                walsh_data_r <= walsh_data;
            elsif walsh_shift = '1' then
                walsh_data_r <= shift_right(walsh_data_r, 1);
            end if;
        end if;
    end process walsh_shifter;

    symbol_shifter : process (clk) begin
        if clk'event and clk = '1' then
            if symbol_load = '1' then
                symbol_r <= io_d_in_r;
            elsif symbol_shift = '1' then
                symbol_r <= shift_right(symbol_r, WALSH_SYM_SIZE);
            end if;
        end if;
    end process symbol_shifter;

    output_decode : process (state, walsh_bit, sf, repeat) begin
        sub_addr_out <= FIFO_ADDR;
        sub_d_out <= (others => 'Z');
        sub_write_strobe <= '0';
        sub_addr_strobe <= '0';
        bus_master <= '1';
        io_ready_s <= '0';
        walsh_shift <= '0';
        walsh_load <= '0';
        symbol_load <= '0';
        symbol_shift <= '0';

        case (state) is
            when st_idle =>
                bus_master <= '0';
                symbol_load <= '1';
            when st_load =>
                walsh_load <= '1';
            when st_fifo_write =>
                sub_write_strobe <= '1';
                sub_addr_strobe <= '1';
                if walsh_bit = '1' then
                    sub_d_out <= BIT_MAP_32_1;
                else
```

```
                    sub_d_out <= BIT_MAP_32_0;
                end if;
            when st_fifo_ack =>
                if sf = DOUBLE_WRITE then
                    walsh_shift <= repeat;
                else
                    walsh_shift <= '1';
                end if;
            when st_shift =>
                symbol_shift <= '1';
            when st_bus_ack =>
                io_ready_s <= '1';
        end case;
    end process;

    next_state_decode : process (state, got_write, sub_io_ready, enabled,
                                 set_sf_op, sf, repeat, bit_count,
                                 symbol_count, fifo_almost_full) begin
        state_i <= state;
        bit_count_i <= bit_count;
        symbol_count_i <= symbol_count;
        repeat_i <= repeat;

        case (state) is
            when st_idle =>
                if got_write = '1' and enabled = '1' and set_sf_op = '0' then
                    state_i <= st_load;
                end if;
                bit_count_i <= to_unsigned(0, bit_count_i'length);
                symbol_count_i <= to_unsigned(0, symbol_count_i'length);
            when st_load =>
                if fifo_almost_full = '0' then
                    state_i <= st_fifo_write;
                end if;
            when st_fifo_write =>
                state_i <= st_fifo_ack;
            when st_fifo_ack =>
                -- Also do counter updating here
                if sub_io_ready = '1' then
                    if sf = DOUBLE_WRITE and repeat = '0' then
                        state_i <= st_fifo_write;
                        repeat_i <= '1';
                    elsif bit_count /= WALSH_CODE_SIZE - 1 then
                        state_i <= st_fifo_write;
                        bit_count_i <= bit_count + 1;
                        repeat_i <= '0';
                    elsif symbol_count /= SYM_LIMIT - 1 then
                        state_i <= st_shift;
                        repeat_i <= '0';
                    else
                        state_i <= st_bus_ack;
                        repeat_i <= '0';
                    end if;
                end if;
            when st_shift =>
                state_i <= st_load;
                symbol_count_i <= symbol_count + 1;
                bit_count_i <= to_unsigned(0, bit_count_i'length);
            when st_bus_ack =>
                state_i <= st_idle;
        end case;
    end process;

    sync_proc : process (clk, reset) begin
        if reset = '1' then
            state <= st_idle;
            bit_count <= to_unsigned(0, bit_count'length);
            symbol_count <= to_unsigned(0, symbol_count'length);
            repeat <= '0';
        elsif clk'event and clk = '1' then
            state <= state_i;
            symbol_count <= symbol_count_i;
            bit_count <= bit_count_i;
```

```
            repeat <= repeat_i;
        end if;
    end process;

    sf_proc : process (clk, reset) begin
        if reset = '1' then
            sf <= (others => '0');
            io_ready_d <= '0';
        elsif clk'event and clk = '0' then
            if io_ready_d = '1' then
                io_ready_d <= '0';
            elsif set_sf_op = '1' and do_ack = '1' then
                sf <= io_d_in_r(7 downto 0);
                io_ready_d <= '1';
            end if;
        end if;
    end process sf_proc;

    -- Get IO data
    io_proc : process(clk, reset, io_ready_s) begin
        if reset = '1' or io_ready_s = '1' then
            io_d_in_r <= (others => '0');
            got_write <= '0';
            do_ack <= '0';
        -- Read IO bus on falling edge
        elsif clk'event and clk = '0' then
            if io_write_strobe = '1' then
                io_d_in_r <= io_d_in;
                got_write <= '1';
                do_ack <= '1';
            else
                do_ack <= '0';
            end if;
        end if;
    end process io_proc;

    -- Get address from IO bus
    get_io_addr : process(clk, reset) begin
        if reset = '1' then
            io_addr_reg <= (others => '0');
        elsif clk'event and clk = '0' then
            if io_addr_strobe = '1' then
                io_addr_reg <= io_addr;
            end if;
        end if;
    end process get_io_addr;

    -- Assert enabled
    with io_addr_reg (7 downto 0) select enabled
        <=  '1' when MODULATOR_ADDR,
            '1' when MODULATOR_SF_ADDR,
            '0' when others;

    with io_addr_reg (7 downto 0) select set_sf_op
        <=  '1' when MODULATOR_SF_ADDR,
            '0' when others;

end modulator_arch;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library transceiver;
use transceiver.bits.all;

entity parallel_to_serial is
        port (
                clk, reset : in std_logic;
                trigger : in std_logic;
                trig_clk : in std_logic;
                fifo_d_in : in vec32_t;
                fifo_rden : out std_logic;
                fifo_empty : in std_logic;
                data_active : out std_logic;
                data_out : out std_logic);
end parallel_to_serial;

architecture parallel_to_serial_arch of parallel_to_serial is

    type state_type is (st_reset,
                        st_rd_fifo_1,
                        st_rd_fifo_2,
                        st_output_data,
                        st_chain_1,
                        st_chain_2,
                        st_chain_3,
                        st_finish_1,
                        st_finish_2);
    signal state, next_state : state_type;
    signal data_out_i : std_logic;
    signal data_active_i : std_logic;

    signal tmp_data, tmp_data_i : vec32_t := (others => '0');

    type cur_bit_type is range 0 to 31;
    signal cur_bit, cur_bit_i : cur_bit_type := 0;

    signal enabled, en_reset : std_logic;

begin

    trigger_proc : process(trig_clk, reset, en_reset) begin
        if reset = '1' or en_reset = '1' then
            enabled <= '0';
        elsif trig_clk'event and trig_clk = '1' then
            if trigger = '1' then
                enabled <= '1';
            end if;
        end if;
    end process trigger_proc;

    output_decode : process (state, tmp_data) begin
        if (state = st_rd_fifo_1) or (state = st_chain_2) then
            fifo_rden <= '1';
```

```
        else
            fifo_rden <= '0';
        end if;

        if (state = st_output_data) or (state = st_chain_1) or
            (state = st_chain_2) or (state = st_chain_3) or
            (state = st_finish_1) or (state = st_finish_2) then
            data_out_i <= tmp_data(31);
            data_active_i <= '1';
        else
            data_out_i <= '0';
            data_active_i <= '0';
        end if;
    end process;

    -- enabled is asynchronous
    -- synthesise fsm with -safe_implementation
    next_state_decode : process (state, fifo_empty, cur_bit,
                                 fifo_d_in, tmp_data, enabled) begin
        next_state <= state;
        cur_bit_i <= cur_bit;
        tmp_data_i <= tmp_data;
        en_reset <= '0';

        case (state) is
            when st_reset =>
                if fifo_empty = '0' and enabled = '1' then
                    next_state <= st_rd_fifo_1;
                end if;
            when st_rd_fifo_1 =>
                next_state <= st_rd_fifo_2;
            when st_rd_fifo_2 =>
                tmp_data_i <= fifo_d_in;
                next_state <= st_output_data;
            when st_output_data =>
                tmp_data_i <= shift_left(tmp_data, 1);
                if cur_bit = 31 - 3 then
                    cur_bit_i <= 0;
                    next_state <= st_chain_1;
                else
                    cur_bit_i <= cur_bit + 1;
                end if;
            when st_chain_1 =>
                tmp_data_i <= shift_left(tmp_data, 1);
                if fifo_empty = '0' then
                    next_state <= st_chain_2;
                else
                    next_state <= st_finish_1;
                end if;
            when st_chain_2 =>
                tmp_data_i <= shift_left(tmp_data, 1);
                next_state <= st_chain_3;
            when st_chain_3 =>
                tmp_data_i <= fifo_d_in;
                next_state <= st_output_data;
            when st_finish_1 =>
                tmp_data_i <= shift_left(tmp_data, 1);
                next_state <= st_finish_2;
            when st_finish_2 =>
                en_reset <= '1';
                next_state <= st_reset;
        end case;
    end process;

    sync_proc : process (clk, reset) begin
        if reset = '1' then
            state <= st_reset;
            data_out <= '0';
            data_active <= '0';
            cur_bit <= 0;
            tmp_data <= (others => '0');
        elsif clk'event and clk = '1' then
            state <= next_state;
```

```
            data_active <= data_active_i;
            data_out <= data_out_i;
            cur_bit <= cur_bit_i;
            tmp_data <= tmp_data_i;
        end if;
    end process;

end parallel_to_serial_arch;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library transceiver;
use transceiver.bits.all;

    -- Double shifting may help where the clock frequencies are significantly
    -- different, and the data contains long runs of 0 or 1
#define DOUBLE_SHIFT 0
    -- Glitches can cause case a delay shift if they happen to trigger the right
    -- sequence of use_0, use_90, use_180, use_270 in order (or reverse order)
    -- A possible work around is to require n consecutive counts of each
    -- phase_hold.
    -- Should be 2^n
#define SLOW_SHIFT 32

entity data_synchroniser is
        port (
                serial_clk, reset : in std_logic;
                serial_clk_90 : in std_logic;
                data_in : in std_logic;
                data_out : out std_logic);
end data_synchroniser;

architecture data_synchroniser_arch of data_synchroniser is

    -- Larger shift register allows for greater clock drift.
    constant WRAP_REG_SIZE : natural := 64;

    signal serial_clk_180 : std_logic;
    signal serial_clk_270 : std_logic;

    signal d_0      : std_logic_vector (1 downto 0);
    signal d_90     : std_logic_vector (1 downto 0);
    signal d_180    : std_logic_vector (1 downto 0);
    signal d_270    : std_logic_vector (1 downto 0);

    signal d_0_wrap     : std_logic_vector (WRAP_REG_SIZE-1 downto 0);
    signal d_90_wrap    : std_logic_vector (WRAP_REG_SIZE-1 downto 0);
    signal d_180_wrap   : std_logic_vector (WRAP_REG_SIZE-1 downto 0);
    signal d_270_wrap   : std_logic_vector (WRAP_REG_SIZE-1 downto 0);

    signal d_0_n, d_0_p : std_logic;
    signal d_90_n, d_90_p : std_logic;
    signal d_180_n, d_180_p : std_logic;
    signal d_270_n, d_270_p : std_logic;

    signal use_0, use_0_hold : std_logic;
    signal use_90, use_90_hold : std_logic;
    signal use_180, use_180_hold : std_logic;
    signal use_270, use_270_hold : std_logic;

    -- Only used for double shifting, optimised away otherwise
    signal previous_shift : std_logic;
```

```
    signal delay_time : unsigned (bits_for_val(WRAP_REG_SIZE-1)-1 downto 0);

    signal delay_d_0 : std_logic_vector (1 downto 0);
    signal delay_d_90 : std_logic_vector (1 downto 0);
    signal delay_d_180 : std_logic_vector (1 downto 0);
    signal delay_d_270 : std_logic_vector (1 downto 0);

#if SLOW_SHIFT
    constant SLOW_SHIFT_BITS : natural := bits_for_val(SLOW_SHIFT-1);
    signal use_0_valid, use_90_valid, use_180_valid, use_270_valid : std_logic;
    signal use_0_valid_sum : unsigned (SLOW_SHIFT_BITS-1 downto 0);
    signal use_90_valid_sum : unsigned (SLOW_SHIFT_BITS-1 downto 0);
    signal use_180_valid_sum : unsigned (SLOW_SHIFT_BITS-1 downto 0);
    signal use_270_valid_sum : unsigned (SLOW_SHIFT_BITS-1 downto 0);
    signal reset_valid : std_logic;
#endif

begin

    serial_clk_180 <= not serial_clk;
    serial_clk_270 <= not serial_clk_90;

    -- Oversample input signal using 4 phases of serial_clk
    sample_0 : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            d_0(0) <= data_in;
            -- Resample onto serial_clk
            d_0(1) <= d_0(0);
            d_90(1) <= d_90(0);
            d_180(1) <= d_180(0);
        end if;
    end process sample_0;

    sample_90 : process(serial_clk_90) begin
        if serial_clk_90'event and serial_clk_90 = '1' then
            d_90(0) <= data_in;
            -- Special case for d_270, allow half a clock cycle before
            -- resampling.
            d_270(1) <= d_270(0);
        end if;
    end process sample_90;

    sample_180 : process(serial_clk_180) begin
        if serial_clk_180'event and serial_clk_180 = '1' then
            d_180(0) <= data_in;
        end if;
    end process sample_180;

    sample_270 : process(serial_clk_270) begin
        if serial_clk_270'event and serial_clk_270 = '1' then
            d_270(0) <= data_in;
        end if;
    end process sample_270;

    -- Shift register to delay incoming data. This allows us to select either a
    -- shorter or longer delay whenever we wrap from 0 to 270 or vice versa.
    wrap_delay : process (serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            d_0_wrap <= concat_bit(d_0_wrap, d_0(1));
            d_90_wrap <= concat_bit(d_90_wrap, d_90(1));
            d_180_wrap <= concat_bit(d_180_wrap, d_180(1));
            d_270_wrap <= concat_bit(d_270_wrap, d_270(1));
        end if;
    end process wrap_delay;

    -- May need to optimise this using SRL16
    delay_d_0(0) <=d_0_wrap(to_integer(delay_time));
    delay_d_90(0) <=d_90_wrap(to_integer(delay_time));
    delay_d_180(0) <=d_180_wrap(to_integer(delay_time));
    delay_d_270(0) <=d_270_wrap(to_integer(delay_time));

    edge_store : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
```

```
            delay_d_0(1) <= delay_d_0(0);
            delay_d_90(1) <= delay_d_90(0);
            delay_d_180(1) <= delay_d_180(0);
            delay_d_270(1) <= delay_d_270(0);
        end if;
    end process edge_store;

    edge_detect : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            d_0_p <=
                    (delay_d_0(1)   xor delay_d_0(0))   and delay_d_0(0);
            d_90_p <=
                    (delay_d_90(1)  xor delay_d_90(0))  and delay_d_90(0);
            d_180_p <=
                    (delay_d_180(1) xor delay_d_180(0)) and delay_d_180(0);
            d_270_p <=
                    (delay_d_270(1) xor delay_d_270(0)) and delay_d_270(0);

            d_0_n <=
                    (delay_d_0(1)   xor delay_d_0(0))   and not delay_d_0(0);
            d_90_n <=
                    (delay_d_90(1)  xor delay_d_90(0))  and not delay_d_90(0);
            d_180_n <=
                    (delay_d_180(1) xor delay_d_180(0)) and not delay_d_180(0);
            d_270_n <=
                    (delay_d_270(1) xor delay_d_270(0)) and not delay_d_270(0);
        end if;
    end process edge_detect;

    select_phase : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            use_0   <= (d_0_p and     d_90_p and not d_180_p and not d_270_p) or
                       (d_0_n and     d_90_n and not d_180_n and not d_270_n);
            use_90  <= (d_0_p and     d_90_p and     d_180_p and not d_270_p) or
                       (d_0_n and     d_90_n and     d_180_n and not d_270_n);
            use_180 <= (d_0_p and     d_90_p and     d_180_p and     d_270_p) or
                       (d_0_n and     d_90_n and     d_180_n and     d_270_n);
            use_270 <= (d_0_p and not d_90_p and not d_180_p and not d_270_p) or
                       (d_0_n and not d_90_n and not d_180_n and not d_270_n);
        end if;
    end process select_phase;

    hold_phase : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if reset = '1' then
                use_0_hold <= '0';
                use_90_hold <= '0';
                use_180_hold <= '0';
                use_270_hold <= '0';
#if SLOW_SHIFT
                reset_valid <= '0';
            elsif reset_valid = '1' then
                reset_valid <= '0';
#endif
#if SLOW_SHIFT
            elsif (use_0_valid or use_90_valid or
                   use_180_valid or use_270_valid) = '1' then
#elif DOUBLE_SHIFT
            elsif (use_0 or use_90 or use_180 or use_270) = '1' then
#else
            -- Don't allow double shifts
            elsif ((use_0 and not use_180_hold) or
                (use_90 and not use_270_hold) or
                (use_180 and not use_0_hold) or
                (use_270 and not use_90_hold)) = '1' then
#endif
#if SLOW_SHIFT
                use_0_hold <= use_0_valid;
                use_90_hold <= use_90_valid;
                use_180_hold <= use_180_valid;
                use_270_hold <= use_270_valid;
                reset_valid <= '1';
#else
```

```
                use_0_hold <= use_0;
                use_90_hold <= use_90;
                use_180_hold <= use_180;
                use_270_hold <= use_270;
#endif
            end if;
        end if;
    end process hold_phase;


#if SLOW_SHIFT
    phase_valid : process (serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if (reset_valid or reset) = '1' then
                use_0_valid <= '0';
                use_90_valid <= '0';
                use_180_valid <= '0';
                use_270_valid <= '0';
                use_0_valid_sum <= (others => '0');
                use_90_valid_sum <= (others => '0');
                use_180_valid_sum <= (others => '0');
                use_270_valid_sum <= (others => '0');
            else
                if use_0 = '1' then
                    use_0_valid_sum <= use_0_valid_sum + 1;
                elsif use_90 = '1' then
                    use_90_valid_sum <= use_90_valid_sum + 1;
                elsif use_180 = '1' then
                    use_180_valid_sum <= use_180_valid_sum + 1;
                elsif use_270 = '1' then
                    use_270_valid_sum <= use_270_valid_sum + 1;
                end if;

                if use_0_valid_sum = SLOW_SHIFT-1 then
                    use_0_valid <= '1';
                elsif use_90_valid_sum = SLOW_SHIFT-1 then
                    use_90_valid <= '1';
                elsif use_180_valid_sum = SLOW_SHIFT-1 then
                    use_180_valid <= '1';
                elsif use_270_valid_sum = SLOW_SHIFT-1 then
                    use_270_valid <= '1';
                end if;
            end if;
        end if;
    end process phase_valid;
#endif

    wrap_detect : process(serial_clk, reset) begin
        if serial_clk'event and serial_clk = '1' then
            if reset = '1' then
                -- On packet reset, initialise the delay to the centre of
                -- the array.
                delay_time <= to_unsigned(WRAP_REG_SIZE/2, delay_time'length);
#if SLOW_SHIFT
            elsif (use_270_hold and use_0_valid) = '1' then
#else
            elsif (use_270_hold and use_0) = '1' then
#endif
                previous_shift <= '0';
                delay_time <= delay_time - 1;
#if SLOW_SHIFT
            elsif (use_0_hold and use_270_valid) = '1' then
#else
            elsif (use_0_hold and use_270) = '1' then
#endif
                previous_shift <= '1';
                delay_time <= delay_time + 1;
#if DOUBLE_SHIFT && !SLOW_SHIFT
            elsif ( (use_0_hold and use_90) or
                    (use_90_hold and use_180) or
                    (use_180_hold and use_270) ) = '1' then
                previous_shift <= '0';
            elsif ( (use_270_hold and use_180) or
```

```
                    (use_180_hold and use_90) or
                    (use_90_hold and use_0) ) = '1' then
                previous_shift <= '1';
            -- For double shifts, it's not so easy. We rely on the previous
            -- shift direction.
            elsif ( (use_0_hold and use_180) or
                    (use_180_hold and use_0) or
                    (use_90_hold and use_270) or
                    (use_270_hold and use_90) ) = '1' then
                if previous_shift = '0' then
                    delay_time <= delay_time - 1;
                else
                    delay_time <= delay_time + 1;
                end if;
#endif
            end if;
        end if;
    end process wrap_detect;

    data_out <= (delay_d_0(0)   and use_0_hold) or
                (delay_d_90(0)  and use_90_hold) or
                (delay_d_180(0) and use_180_hold) or
                (delay_d_270(0) and use_270_hold);

end data_synchroniser_arch;
```

```
-- Copyright (C) 2016 Kim Taylor
--
-- This file is part of hbc_mac.
--
-- hbc_mac is free software: you can redistribute it and/or modify
-- it under the terms of the GNU General Public License as published by
-- the Free Software Foundation, either version 3 of the License, or
-- (at your option) any later version.
--
-- hbc_mac is distributed in the hope that it will be useful,
-- but WITHOUT ANY WARRANTY; without even the implied warranty of
-- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
-- GNU General Public License for more details.
--
-- You should have received a copy of the GNU General Public License
-- along with hbc_mac.  If not, see <http://www.gnu.org/licenses/>.

-- Hamming weight calculator based on the LUT method proposed in:
-- "Digital Hamming Weight and Distance Analyzers for Binary Vectors and
-- Matrices" – International Journal of Innovative Computing, Information and
-- Control – Volume 9, Number 12, December 2013
-- Valery Sklyarov and Iouliia Skilarova

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity hamming_lut is
    port (
        val : in std_logic_vector (63 downto 0);
        weight : out std_logic_vector (6 downto 0));
end entity hamming_lut;

----------------- 36 bit LUT -------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library unisim;
use unisim.vcomponents.all;

entity hamming_lut_36 is
    port (
        val : in std_logic_vector (35 downto 0);
        weight : out std_logic_vector (5 downto 0));
end entity hamming_lut_36;

----------------- 16 bit LUT -------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library unisim;
use unisim.vcomponents.all;

entity hamming_lut_16 is
    port (
        val : in std_logic_vector (15 downto 0);
        weight : out std_logic_vector (4 downto 0));
end entity hamming_lut_16;

----------------- Architectures ----------------------------
architecture toplevel of hamming_lut is

    signal sum : unsigned(6 downto 0);
    signal weight_0, weight_1 : std_logic_vector(5 downto 0);
    constant zeros : std_logic_vector(7 downto 0) := X"00";
    signal tmp_val : std_logic_vector (35 downto 0);

begin

    weight <= std_logic_vector(unsigned('0' & weight_0) + unsigned(weight_1));
```

```
    tmp_val <= zeros & val(63 downto 36);

    hammint_lut_36_0 : entity work.hamming_lut_36
        port map (val => val(35 downto 0), weight => weight_0);

    hammint_lut_36_1 : entity work.hamming_lut_36
        port map (val => tmp_val, weight => weight_1);

end architecture toplevel;

----------------- 36 bit LUT -------------------------------
architecture hamming_lut_arch of hamming_lut_36 is
#define SIGNAL_HAMMING_LUT(layer, num) \
    signal tmp_##layer##_##num : std_logic_vector (5 downto 0);             \
    signal tmp_out_##layer##_##num : std_logic_vector (2 downto 0);

    SIGNAL_HAMMING_LUT(0, 0)
    SIGNAL_HAMMING_LUT(0, 1)
    SIGNAL_HAMMING_LUT(0, 2)
    SIGNAL_HAMMING_LUT(0, 3)
    SIGNAL_HAMMING_LUT(0, 4)
    SIGNAL_HAMMING_LUT(0, 5)
    SIGNAL_HAMMING_LUT(1, 0)
    SIGNAL_HAMMING_LUT(1, 1)
    SIGNAL_HAMMING_LUT(1, 2)

    signal sum_a : unsigned (4 downto 0);
    signal sum_b : unsigned (5 downto 0);

begin

#define HAMMING_LUT(layer, num) \
    L##layer##_##num##_LUT_0 : LUT6                                         \
        generic map (INIT => X"6996966996696996")                           \
        port map (                                                          \
            I0 => tmp_##layer##_##num##(0), I1 => tmp_##layer##_##num##(1), \
            I2 => tmp_##layer##_##num##(2), I3 => tmp_##layer##_##num##(3), \
            I4 => tmp_##layer##_##num##(4), I5 => tmp_##layer##_##num##(5), \
            O => tmp_out_##layer##_##num##(0));                             \
    L##layer##_##num##_LUT_1 : LUT6                                         \
        generic map (INIT => X"8117177E177E7EE8")                           \
        port map (                                                          \
            I0 => tmp_##layer##_##num##(0), I1 => tmp_##layer##_##num##(1), \
            I2 => tmp_##layer##_##num##(2), I3 => tmp_##layer##_##num##(3), \
            I4 => tmp_##layer##_##num##(4), I5 => tmp_##layer##_##num##(5), \
            O => tmp_out_##layer##_##num##(1));                             \
    L##layer##_##num##_LUT_2 : LUT6                                         \
        generic map (INIT => X"FEE8E880E8808000")                           \
        port map (                                                          \
            I0 => tmp_##layer##_##num##(0), I1 => tmp_##layer##_##num##(1), \
            I2 => tmp_##layer##_##num##(2), I3 => tmp_##layer##_##num##(3), \
            I4 => tmp_##layer##_##num##(4), I5 => tmp_##layer##_##num##(5), \
            O => tmp_out_##layer##_##num##(2));

    HAMMING_LUT(0, 0)
    HAMMING_LUT(0, 1)
    HAMMING_LUT(0, 2)
    HAMMING_LUT(0, 3)
    HAMMING_LUT(0, 4)
    HAMMING_LUT(0, 5)

    HAMMING_LUT(1, 0)
    HAMMING_LUT(1, 1)
    HAMMING_LUT(1, 2)

    tmp_0_0 <= val(5 downto 0);
    tmp_0_1 <= val(11 downto 6);
    tmp_0_2 <= val(17 downto 12);
    tmp_0_3 <= val(23 downto 18);
    tmp_0_4 <= val(29 downto 24);
    tmp_0_5 <= val(35 downto 30);

    -- MSB of weights W_n
```

```
    tmp_1_2(0) <= tmp_out_0_0(2);
    tmp_1_2(1) <= tmp_out_0_1(2);
    tmp_1_2(2) <= tmp_out_0_2(2);
    tmp_1_2(3) <= tmp_out_0_3(2);
    tmp_1_2(4) <= tmp_out_0_4(2);
    tmp_1_2(5) <= tmp_out_0_5(2);

    tmp_1_1(0) <= tmp_out_0_0(1);
    tmp_1_1(1) <= tmp_out_0_1(1);
    tmp_1_1(2) <= tmp_out_0_2(1);
    tmp_1_1(3) <= tmp_out_0_3(1);
    tmp_1_1(4) <= tmp_out_0_4(1);
    tmp_1_1(5) <= tmp_out_0_5(1);

    -- LSB of weights W_n
    tmp_1_0(0) <= tmp_out_0_0(0);
    tmp_1_0(1) <= tmp_out_0_1(0);
    tmp_1_0(2) <= tmp_out_0_2(0);
    tmp_1_0(3) <= tmp_out_0_3(0);
    tmp_1_0(4) <= tmp_out_0_4(0);
    tmp_1_0(5) <= tmp_out_0_5(0);

    sum_a <= unsigned('0' & tmp_out_1_2 & '0') + unsigned(tmp_out_1_1);
    sum_b <= (sum_a & '0' ) + unsigned(tmp_out_1_0);

    weight <= std_logic_vector(sum_b);

end architecture hamming_lut_arch;

----------------- 16 bit LUT --------------------------------
architecture hamming_lut_arch of hamming_lut_16 is

    SIGNAL_HAMMING_LUT(0, 0)
    SIGNAL_HAMMING_LUT(0, 1)
    SIGNAL_HAMMING_LUT(0, 2)

    signal sum : unsigned (4 downto 0);

begin

    HAMMING_LUT(0, 0)
    HAMMING_LUT(0, 1)
    HAMMING_LUT(0, 2)

    tmp_0_0 <= val(5 downto 0);
    tmp_0_1 <= val(11 downto 6);
    tmp_0_2 <= "00" & val(15 downto 12);

    sum <=  unsigned("00" & tmp_out_0_0) +
            unsigned("00" & tmp_out_0_1) +
            unsigned("00" & tmp_out_0_2);

    weight <= std_logic_vector(sum);

end architecture hamming_lut_arch;
```

```
#include <preprocessor/constants.vhh>

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library transceiver;
use transceiver.numeric.all;
use transceiver.bits.all;

entity phase_align is
    port (
        pkt_reset : in std_logic;
        serial_clk : in std_logic;
        data_in : in std_logic;
        allow_re_align : in std_logic;
        data_in_sync : out std_logic;
        phase_change : out std_logic;
        comma_found_out : out std_logic;
        re_align : out std_logic);
end phase_align;

architecture phase_align_arch of phase_align is

    constant COMMA : std_logic_vector (COMMA_SIZE-1 downto 0) := HEX(PREAMBLE);
    constant COMMA_WEIGHT_BITS : natural := bits_for_val(COMMA_SIZE);

    constant EARLY_SYMBOL_SIZE : natural := 8;
    constant EARLY_SYMBOL_INDEX_BITS : natural :=
                            bits_for_val(EARLY_SYMBOL_SIZE-1);

    function wrap (val : natural) return natural is begin
        if (val = EARLY_SYMBOL_SIZE-1) then
            return 0;
        else
            return val + 1;
        end if;
    end function wrap;

    subtype demod_reg_t is std_logic_vector (COMMA_SIZE-1 downto 0);
    subtype phase_index_t is unsigned (EARLY_SYMBOL_INDEX_BITS-1 downto 0);
    subtype comma_weight_t is std_logic_vector (COMMA_WEIGHT_BITS-1 downto 0);

    type demod_reg_array_t is array (EARLY_SYMBOL_SIZE-1 downto 0) of
                            demod_reg_t;
    type phase_sum_array_t is array (EARLY_SYMBOL_SIZE-1 downto 0) of
                            phase_index_t;
    type comma_weight_array_t is array (EARLY_SYMBOL_SIZE-1 downto 0) of
                            comma_weight_t;


    signal s2p_align_index : phase_index_t := (others => '0');

    signal data_in_r : std_logic;

    signal expected_phase : std_logic;
```

```
    signal current_phase : std_logic_vector(EARLY_SYMBOL_SIZE-1 downto 0);
    signal max_phase_sum : phase_index_t;
    signal best_phase, best_phase_r : phase_index_t;
    signal phase_sum : phase_sum_array_t;
    signal demod_regs : demod_reg_array_t;

    signal comma_weight :  comma_weight_array_t;
    signal max_comma_weight : comma_weight_t;
    signal comma_found : std_logic_vector(EARLY_SYMBOL_SIZE-1 downto 0);
    signal data_inverted : std_logic_vector(EARLY_SYMBOL_SIZE-1 downto 0);
    signal comma_xnor : demod_reg_array_t;
    signal invert_data, invert_data_r : std_logic;

begin

    data_sync : process (serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            data_in_r <= data_in;
        end if;
    end process data_sync;

    -- Need to delay the data stream, as the upstream re-align process will
    -- require one clock cycle.
    sync_data_proc : process (serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            data_in_sync <= data_in_r xor invert_data_r xor best_phase_r(0);
        end if;
    end process sync_data_proc;

    -- If we later choose another phase, don't alter upstream's data feed
    allow_re_align_proc : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if allow_re_align = '1' then
                invert_data_r <= invert_data;
                best_phase_r <= best_phase;
            end if;
        end if;
    end process allow_re_align_proc;

    phase_change <= bool_to_bit(data_in = data_in_r);

    early_compare_phase : process (serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if s2p_align_index = EARLY_SYMBOL_SIZE-1 then
                phase_sum(0) <= (others => '0');
                expected_phase <= not invert_data;
            else
                expected_phase <= not expected_phase;
                if (data_in_r = expected_phase) then
                    phase_sum(0) <= phase_sum(0) + 1;
                end if;
            end if;
            s2p_align_index <= s2p_align_index + 1;
        end if;
    end process early_compare_phase;

    early_compare_phase_gen : for i in 0 to EARLY_SYMBOL_SIZE-2 generate
        early_compare_phase : process (serial_clk) begin
            if serial_clk'event and serial_clk = '1' then
                if s2p_align_index = i then
                    phase_sum(wrap(i)) <= (others => '0');
                elsif (data_in_r = expected_phase) then
                    phase_sum(wrap(i)) <= phase_sum(wrap(i)) + 1;
                end if;
            end if;
        end process early_compare_phase;
    end generate early_compare_phase_gen;

    early_detect_phase_gen : for i in 0 to EARLY_SYMBOL_SIZE-1 generate
        current_phase(i) <= bool_to_bit(phase_sum(i) > EARLY_SYMBOL_SIZE/2);
    end generate early_detect_phase_gen;

    early_demodulate_gen : for i in 0 to EARLY_SYMBOL_SIZE-1 generate
```

```
        early_demodulate : process (serial_clk) begin
            if serial_clk'event and serial_clk = '1' then
                if s2p_align_index = i then
                    demod_regs(wrap(i)) <=
                        concat_bit(demod_regs(wrap(i)), current_phase(wrap(i)));
                end if;
            end if;
        end process early_demodulate;
    end generate early_demodulate_gen;

    -- FIXME: This probably generates too much logic. Probably a multiplexer
    -- could be used to pass data to one hamming_lut at the correct times.
    comma_gen : for i in 0 to EARLY_SYMBOL_SIZE-1 generate
        comma_xnor(i) <= demod_regs(i) xnor COMMA;
        comma_distance_lut : entity work.hamming_lut
            port map (val => comma_xnor(i), weight => comma_weight(i));
        comma_found(i) <= weight_threshold(comma_weight(i));
        data_inverted(i) <= weight_inverted(comma_weight(i));
    end generate comma_gen;

    check_inverted : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if pkt_reset = '1' then
                invert_data <= '0';
            elsif data_inverted /= zeros(data_inverted'length) then
                invert_data <= '1';
            end if;
        end if;
    end process check_inverted;

    -- Mod 8 counters can't actually show a perfect phase match
    choose_phase : process (serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if pkt_reset = '1' then
                max_phase_sum <= (others => '0');
                -- Need to initialise this to 1 so that weight_comp can do a
                -- three bit comparison, but also cover 64
                max_comma_weight <= std_logic_vector(
                                to_unsigned(1, max_comma_weight'length));
                best_phase <= (others => '0');
            else
                for i in 0 to EARLY_SYMBOL_SIZE-1 loop
                    if  (s2p_align_index = i) and
                        (comma_found(wrap(i)) = '1') and
                        weight_comp(comma_weight(wrap(i)), max_comma_weight)
                    then
                        max_comma_weight <= comma_weight(wrap(i));
                        if phase_sum(wrap(i)) > max_phase_sum then
                            max_phase_sum <= phase_sum(wrap(i));
                            best_phase <= to_unsigned(
                                            wrap(i), best_phase'length);
                        end if;
                    end if;
                end loop;
            end if;
        end if;
    end process choose_phase;

    comma_found_out <= comma_found(to_integer(best_phase_r));

    re_align <= bool_to_bit(s2p_align_index = best_phase_r);

end phase_align_arch;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library transceiver;
use transceiver.numeric.all;
use transceiver.bits.all;

entity walsh_decoder is
    port (
        clk : in std_logic;
        data : in walsh_code_t;
        sym : out walsh_sym_t);
end entity walsh_decoder;

architecture walsh_decoder_arch of walsh_decoder is

    signal reset : std_logic := '1';
    subtype distance_t is std_logic_vector (
                    bits_for_val(WALSH_CODE_SIZE)-1 downto 0);

    signal cur_walsh : walsh_code_t;
    signal cur_distance : distance_t;
    signal walsh_xnor : walsh_code_t;

    signal sym_counter : walsh_sym_t := (others => '0');
    signal max_index : walsh_sym_t;
    signal max : distance_t;

begin

    walsh_encoder : entity work.walsh_encode_lut
        port map (symbol => sym_counter, walsh => cur_walsh);

    walsh_xnor <= cur_walsh xnor data;

    walsh_distance : entity work.hamming_lut_16
        port map (val => walsh_xnor, weight => cur_distance);

    loop_proc : process (clk) begin
        if clk'event and clk = '1' then
            if reset = '1' then
                reset <= '0';
                max <= (others => '0');
                sym <= max_index;
            else
                sym_counter <= std_logic_vector(unsigned(sym_counter) + 1);
                if cur_distance > max then
                    max <= cur_distance;
                    max_index <= sym_counter;
                end if;
                if unsigned(sym_counter) = WALSH_CODE_SIZE-1 then
                    reset <= '1';
                end if;
            end if;
        end if;
```

```
    end process loop_proc;
end architecture walsh_decoder_arch;
```

```
#include <preprocessor/constants.vhh>

#define DEBUG 0

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library transceiver;
use transceiver.numeric.all;
use transceiver.bits.all;

entity serial_to_parallel is
        port (
                serial_clk, reset_i : in std_logic;
                fifo_d_out : out vec32_t;
                fifo_wren : out std_logic;
                fifo_full : in std_logic;
                enable : in std_logic;
                data_in : in std_logic;
                pkt_active : out std_logic;
                pkt_reset : out std_logic;
                pkt_ready : out std_logic;
                pkt_ack : in std_logic);
end serial_to_parallel;

architecture serial_to_parallel_arch of serial_to_parallel is

#define _COUNT_OFFSET(val) COMMA_SIZE + INT(val)
#define COUNT_OFFSET(val) _COUNT_OFFSET(val)

    constant RESET_DELAY : natural := 16;
    constant MAX_SYMBOL_SIZE : natural := 64;
    constant SYMBOL_INDEX_BITS : natural := bits_for_val(MAX_SYMBOL_SIZE-1);
    constant RATE_SELECT_BITS : natural := bits_for_val(MAX_SYMBOL_SIZE);
    constant COMMA_WEIGHT_BITS : natural := bits_for_val(COMMA_SIZE);
    constant SFD : std_logic_vector (COMMA_SIZE-1 downto 0) := HEX(SFD);
    constant NIBBLE_SIZE : natural := 8;
    constant WORD_SIZE : natural := 32;
    constant PKT_END_THRESH : natural := 8;

    signal using_ri : std_logic;
    signal rate_found : std_logic;
    signal ri_rate : unsigned(RATE_SELECT_BITS-1 downto 0);
    signal r_sf : unsigned(RATE_SELECT_BITS-1 downto 0);

    signal sym_reset_i : std_logic;
    signal sym_reset : std_logic;
    signal reset_shift_r : std_logic_vector(RESET_DELAY-1 downto 0);
    signal reset : std_logic;
    attribute equivalent_register_removal : string;
    attribute max_fanout : string;
    attribute shreg_extract : string;
    attribute equivalent_register_removal of reset : signal is "no";
    attribute max_fanout of reset : signal is "10";
    attribute shreg_extract of reset : signal is "no";
```

```
    signal data_in_en : std_logic;
    signal data_in_sync : std_logic;
    signal re_align : std_logic;
    signal expected_phase : std_logic;
    signal phase_sum : unsigned (SYMBOL_INDEX_BITS-1 downto 0);
    signal current_phase : std_logic;
    signal current_phase_latch : std_logic;

    signal s2p_index : unsigned (SYMBOL_INDEX_BITS-1 downto 0);
    signal allow_re_align : std_logic;
    signal latch_sfd : std_logic;

    signal walsh_count : unsigned (bits_for_val(WALSH_CODE_SIZE-1)-1 downto 0);
    signal walsh_msb : std_logic_vector (1 downto 0);
    signal walsh_clk : std_logic;
    signal walsh_detect_i : std_logic;
    signal walsh_detect : std_logic;
    signal walsh_reg : std_logic_vector (WALSH_CODE_SIZE-1 downto 0);
    signal nibble_count : unsigned (bits_for_val(NIBBLE_SIZE-1)-1 downto 0);
    signal nibble_ready : std_logic;
    signal nibble_ready_prev : std_logic;
    signal ignore_nibble : std_logic;
    signal fifo_data_valid : std_logic;
    signal header_found : std_logic;
    signal packet_length : unsigned (
                    bits_for_val(MAX_PACKET_WORDS)-1 downto 0);
    signal decoded_sym : walsh_sym_t;
    signal decoded_word : std_logic_vector (WORD_SIZE-1 downto 0);

    signal phase_change : std_logic;
    signal phase_changes : std_logic_vector(PKT_END_THRESH-1 downto 0);
    signal chk_pkt_end : std_logic;
    signal pkt_end_phase, pkt_end_bytes, pkt_end : std_logic;

    signal sfd_weight : std_logic_vector (COMMA_WEIGHT_BITS-1 downto 0);
    signal sfd_xnor : std_logic_vector (COMMA_SIZE-1 downto 0);
    signal demod_reg : std_logic_vector (COMMA_SIZE-1 downto 0);
    signal comma_found : std_logic;
    signal sfd_found : std_logic;
    signal sfd_found_i : std_logic;
    signal sfd_finished : std_logic;
    signal ri_count : unsigned (
                    bits_for_val(COUNT_OFFSET(RI_OFFSET_MAX))-1 downto 0);

    type state_type is (
                        st_align_1,     -- Wait for alignment from phase_align
                        st_align_2,     -- Wait for alignment from phase_align
                        st_preamble,    -- Load in phase changes until
                                        -- successful comma detect.
                        st_sfd,         -- Detect SFD and padding (data rate).
                        st_demodulate,  -- Map to nearest Walsh code before
                                        -- pushing to FIFO.
                        st_pkt_end);
    signal state, state_i : state_type;

#define EXPORT_DATA_STREAM 0
#if EXPORT_DATA_STREAM
    signal stream_data : std_logic_vector (7 downto 0);

    type output_ft is file of std_logic_vector;
    file output_file : output_ft open WRITE_MODE is "bit_data";

    procedure write_output(val: std_logic_vector(7 downto 0)) is begin
        write(output_file, val);
    end procedure write_output;
#endif

begin

    reset_sync_proc : process (serial_clk, reset_i) begin
        if reset_i = '1' then
            reset_shift_r <= (others => '1');
```

```
        elsif serial_clk'event and serial_clk = '1' then
            reset_shift_r <= shift_left(reset_shift_r, 1);
        end if;
    end process reset_sync_proc;

    reset <= reset_shift_r(RESET_DELAY-1);

    fifo_control : process(state) begin
        using_ri <= '1';
        allow_re_align <= '0';
        walsh_detect_i <= '0';
        chk_pkt_end <= '0';
        sym_reset_i <= '0';
        latch_sfd <= '0';
        pkt_active <= '1';
        case(state) is
            when st_align_1 =>
                pkt_active <= '0';
                allow_re_align <= '1';
            when st_align_2 =>
                allow_re_align <= '1';
            when st_preamble =>
                allow_re_align <= '1';
            when st_sfd =>
                latch_sfd <= '1';
            when st_demodulate =>
                walsh_detect_i <= '1';
                chk_pkt_end <= '1';
            when st_pkt_end =>
                pkt_active <= '0';
                sym_reset_i <= '1';
        end case;
    end process fifo_control;

    next_state : process(state, comma_found, sfd_finished, pkt_end,
                         nibble_count) begin
        state_i <= state;
        case (state) is
            when st_align_1 =>
                if comma_found = '1' then
                    state_i <= st_align_2;
                end if;
            when st_align_2 =>
                if comma_found = '0' then
                    state_i <= st_preamble;
                end if;
            when st_preamble =>
                if comma_found = '1' then
                    state_i <= st_sfd;
                end if;
            when st_sfd =>
                if sfd_finished = '1' then
                    state_i <= st_demodulate;
                end if;
            when st_demodulate =>
                if pkt_end = '1' and nibble_count = 2 then
                    state_i <= st_pkt_end;
                end if;
            when st_pkt_end =>
                state_i <= st_align_1;
        end case;
    end process next_state;

    sync_proc : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if reset = '1' then
                state <= st_align_1;
            else
                state <= state_i;
            end if;
        end if;
    end process sync_proc;
```

```
#if EXPORT_DATA_STREAM
    process (serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            stream_data <= concat_bit(stream_data, data_in_sync);
        end if;
    end process;

    process (serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if (re_align = '1') and (state = st_demodulate) then
                write_output(stream_data);
            end if;
        end if;
    end process;
#endif

    data_in_en <= data_in and enable;

    -- Demodulator:
    phase_aligner : entity work.phase_align
        port map (
            pkt_reset => sym_reset,
            serial_clk => serial_clk,
            data_in => data_in_en,
            allow_re_align => allow_re_align,
            data_in_sync => data_in_sync,
            phase_change => phase_change,
            comma_found_out => comma_found,
            re_align => re_align);

    sym_reset <= reset or sym_reset_i;
    pkt_reset <= sym_reset;

    re_align_proc : process (serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if (allow_re_align = '1') and (re_align = '1') then
                phase_sum <= (others => '0');
                s2p_index <= (others => '0');
                expected_phase <= '1';
            elsif s2p_index = r_sf-1 then
                phase_sum <= (others => '0');
                s2p_index <= (others => '0');
                expected_phase <= '1';
            else
                if (data_in_sync = expected_phase) then
                    phase_sum <= phase_sum + 1;
                end if;
                s2p_index <= s2p_index + 1;
                expected_phase <= not expected_phase;
            end if;
        end if;
    end process re_align_proc;

    current_phase <= bool_to_bit(phase_sum >= r_sf/2);

    packet_ack : process (reset, pkt_ack, serial_clk) begin
        if pkt_ack = '1' or reset = '1'then
            pkt_ready <= '0';
        elsif serial_clk'event and serial_clk = '1' then
            if sym_reset_i = '1' then
                pkt_ready <= '1';
            end if;
        end if;
    end process packet_ack;

    demodulate : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if sym_reset = '1' then
                walsh_count <= (others => '1');
                walsh_detect <= '0';
            else
```

```
                if s2p_index = r_sf-1 then
                    current_phase_latch <= not current_phase;
                    demod_reg <= concat_bit(demod_reg, current_phase);
                    if walsh_detect_i = '1' then
                        if walsh_count = WALSH_CODE_SIZE-1 then
                            walsh_detect <= '1';
                            walsh_count <= (others => '0');
                        else
                            walsh_count <= walsh_count + 1;
                        end if;
                    end if;
                end if;
            end if;
        end if;
    end process demodulate;

    sfd_xnor <= demod_reg xnor SFD;

    sfd_distance_lut : entity work.hamming_lut
        port map (val => sfd_xnor, weight => sfd_weight);

    sfd_found_i <= weight_threshold(sfd_weight);

    latch_sfd_proc : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if sym_reset = '1' then
                sfd_found <= '0';
            else
                if sfd_found_i = '1' and latch_sfd = '1' then
                    sfd_found <= '1';
                end if;
            end if;
        end if;
    end process latch_sfd_proc;

    consume_ri_chips : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if sym_reset = '1' then
                sfd_finished <= '0';
            else
                if sfd_found = '1' then
                    if ri_count = COUNT_OFFSET(RI_OFFSET_MAX) then
                        sfd_finished <= '1';
                    end if;
                end if;
            end if;
        end if;
    end process consume_ri_chips;

    -- Count the number of chips in between the last COMMA and the SFD.
    -- This determines whether the packet is using RI or DRF mode.
    count_ri : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if sym_reset = '1' then
                ri_count <= (others => '0');
            else
                if comma_found = '1' then
                    ri_count <= (others => '0');
                elsif s2p_index(2 downto 0) = INT(SF_8)-1 then
                    ri_count <= ri_count + 1;
                end if;
            end if;
        end if;
    end process count_ri;

    set_rate : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if sym_reset = '1' then
                r_sf <= to_unsigned(INT(SF_8), r_sf'length);
            else
                if rate_found = '1' and sfd_finished = '1' then
                    r_sf <= ri_rate;
                end if;
```

```
            end if;
        end if;
    end process set_rate;

    choose_rate : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if sym_reset = '1' then
                ri_rate <= to_unsigned(INT(SF_8), ri_rate'length);
                rate_found <= '0';
            else
                if using_ri = '1' then
                    if sfd_found = '1' and rate_found = '0' then
                        if ri_count = COUNT_OFFSET(RI_OFFSET_8) then
                            ri_rate <= to_unsigned(INT(SF_8), ri_rate'length);
                        elsif ri_count = COUNT_OFFSET(RI_OFFSET_16) then
                            ri_rate <= to_unsigned(INT(SF_16), ri_rate'length);
                        elsif ri_count = COUNT_OFFSET(RI_OFFSET_32) then
                            ri_rate <= to_unsigned(INT(SF_32), ri_rate'length);
                        elsif ri_count = COUNT_OFFSET(RI_OFFSET_64) then
                            ri_rate <= to_unsigned(INT(SF_64), ri_rate'length);
                        end if;
                        rate_found <= '1';
                    end if;
                end if;
            end if;
        end if;
    end process choose_rate;

    store_walsh : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if sym_reset = '1' then
                walsh_reg <= (others => '0');
            else
                if walsh_detect = '1' then
                    if walsh_count = WALSH_CODE_SIZE-1 then
                        walsh_reg <= bit_swap(
                                        demod_reg(WALSH_CODE_SIZE-1 downto 0));
                    end if;
                end if;
            end if;
        end if;
    end process store_walsh;

    walsh_decoder : entity work.walsh_decoder
        port map (clk => serial_clk, data => walsh_reg, sym => decoded_sym);

    walsh_clk_proc : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if sym_reset = '1' then
                walsh_clk <= '0';
                walsh_msb <= (others => '0');
            else
                walsh_msb(1) <= walsh_msb(0);
                walsh_msb(0) <= walsh_count(walsh_count'length-1);
                -- Detect rising edge
                if walsh_msb(1) = '0' and walsh_msb(0) = '1' then
                    walsh_clk <= '1';
                else
                    walsh_clk <= '0';
                end if;
            end if;
        end if;
    end process walsh_clk_proc;

    decode_word : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if sym_reset = '1' then
                nibble_count <= (others => '0');
                decoded_word <= (others => '0');
            else
                if walsh_detect = '1' and walsh_clk = '1' then
                    if nibble_count = NIBBLE_SIZE-1 then
                        nibble_count <= (others => '0');
```

```
                    else
                        nibble_count <= nibble_count + 1;
                    end if;
                    decoded_word <= decoded_sym &
                                    decoded_word(WORD_SIZE-1 downto WALSH_SYM_SIZE);
                end if;
            end if;
        end if;
    end process decode_word;

    nibble_ready <=     bool_to_bit(nibble_count = 1) and
                        bool_to_bit(walsh_count = 0);

    detect_pkt_end : process(serial_clk) begin
        if serial_clk'event and serial_clk = '1' then
            if sym_reset = '1' then
                pkt_end_phase <= '0';
                phase_changes <= (others => '0');
            else
                -- Detect packet ending by receiving 8 by phase_change = '1'
                phase_changes <= concat_bit(phase_changes, phase_change);
                if chk_pkt_end = '1' then
                    if phase_changes = ones(phase_changes'length) then
                        pkt_end_phase <= '1';
                    end if;
                end if;
            end if;
        end if;
    end process detect_pkt_end;

    detect_bytes_read : process(serial_clk) begin
        if serial_clk'event and serial_clk = '0' then
            if sym_reset = '1' then
                pkt_end_bytes <= '0';
            elsif packet_length = 0 then
                pkt_end_bytes <= '1';
            end if;
        end if;
    end process detect_bytes_read;

    pkt_end <= pkt_end_phase or pkt_end_bytes;

    fifo_d_out <= decoded_word;

    push_fifo : process(serial_clk) begin
        if serial_clk'event and serial_clk = '0' then
            if sym_reset = '1' then
                nibble_ready_prev <= '0';
                fifo_data_valid <= '0';
                ignore_nibble <= '1';
            else
                nibble_ready_prev <= nibble_ready;
                if nibble_ready = '1' then
                    if nibble_ready_prev = '1' then
                        fifo_data_valid <= '0';
                    else
                        fifo_data_valid <= not ignore_nibble;
                    end if;
                else
                    if nibble_ready_prev = '1' then
                        ignore_nibble <= '0';
                    end if;
                end if;
            end if;
        end if;
    end process push_fifo;

    fifo_wren <= fifo_data_valid;

    -- Read packet size. Don't worry about error checking, the worst case
    -- scenario is that a MTU packet will be written out.
    header_latch : process(serial_clk) begin
        if serial_clk'event and serial_clk = '0' then
```

```
            if sym_reset = '1' then
                packet_length <= to_unsigned(MAX_PACKET_WORDS,
                                             packet_length'length);
                header_found <= '0';
            else
                if fifo_data_valid  = '1' then
                    if header_found = '0' then
                        header_found <= '1';
                        packet_length <=
                            words_from_size(decoded_word(23 downto 16));
                    else
                        packet_length <= packet_length - 1;
                    end if;
                end if;
            end if;
        end if;
    end process header_latch;

end serial_to_parallel_arch;
```