

Electrical Engineering
Department at HES-SO: Systèmes industriels

***Mains driven transmitter & portable receiver
for visible light communication
Indoor positioning***

Master's thesis to obtain the degree of
Master in Industrial Sciences: Electronics-ICT
offered by: **Jelle Smets** and **Rob Holvoet**

Promotor: Mr. François Corthay

Academic year: 2012 - 2013

Acknowledgements

We could not accomplish this master's thesis without the help of lots of people. Hereby we would like to thank all those people who helped us to bring this thesis work to a good end.

Special thanks to François Corthay as our promotor at HES-SO Valais/Wallis in Sion . For the good advice he gave us, and to encourage us to try different ways to do things and look for new/other solutions. His close and enthusiast accompaniment was a great help for us.

We got a great help from Olivier Walpen and Steve Gallay who gave us advice about lots of different things and did the routing and the etching of our boards, what made it possible to move on faster.

Médard Rieder helped us to design an anti-aliasing filter before the ADC. He recommended a program where you can easily design a filter with a certain cut-off frequency. He also shared his experience about the problems we had with the ADC (sampling noise, input spikes...).

We would also like to thank Oliver Gubler, Silvan Zahno and Darko Petrovic for there enthusiastic help whenever we needed it. Silvan Zahno helped us when we had problems with the Ethernet code.

Many thanks to Didier Blatter for the help, advice and information about designing the transformer circuit.

For information and advice about LEDs and photodiodes we could count on the help from Frederic Truffer.

Dominique Gabioud helped us when we had problems to send a UDP packet from the PC to the FPGA. We had to add manually the IP address and MAC address of the FPGA in the ARP table so we could send messages to the FPGA.

The HES-SO Valais/Wallis in Sion was a great school to do research and we got every support we needed both in terms of information, help, enthusiasm, advice or provision of professional equipment.

All of this wouldn't be possible if we didn't found a place in Sion to sleep, cook or relax. We got a warm welcome from the family Dayer who shared their house with us and showed us around.

Abstract (English)

Names: Rob Holvoet , Jelle Smets

Contact: robholvoet@hotmail.com, jelle.smets@hotmail.com

Title: Mains driven transmitter and portable receiver for visible light communication: indoor positioning.

The possibilities of Visible Light Communication (VLC) for indoor positioning are being studied in this work. The placement of several LEDs on the ceiling can have a dual purpose: first, the lighting in the room and second the transmission of binary sequences. The binary sequences are sent at a high frequency and are thus invisible for the human eye. A light receiving device can detect these sequences and measure their amplitude. The different amplitudes are in function of the distances. This property allows to calculate the position of the receiver in the room.

For the sending part, a stand-alone prototype is created to be connected to the mains in a room. A transformer circuit (Flyback) is used to drive the led. Two different possibilities to implement the binary sequence in the emitted light are investigated. Once by using the switch of the transformer circuit and once by placing a switch in series with the LED. An FPGA is used to generate the binary sequence.

For the receiving part, a photodiode is used with an analog circuitry to detect, filter and amplify the transmitted light sequences. An ADC is used to digitalize the received information. This makes it possible to send the data to a digital device. Software can then be used to process the data.

Key words: Visible Light Communication (VLC), indoor positioning, flyback converter, optical receiver, correlation filter.

Abstract (Français)

Noms: Rob Holvoet , Jelle Smets

Contact: robholvoet@hotmail.com, jelle.smets@hotmail.com

Titre: Émetteur alimenté par le secteur et récepteur portable pour la communication par la lumière visible: détermination du placement à l' intérieure.

Les possibilités de la communication par la lumière visible (Visible Light Communication, VLC) ont été étudiées pour le calcul de position à l'intérieur d'une pièce. La fixation de plusieurs lampes à LEDs au plafond peut avoir deux buts: premièrement, l'éclairage dans une chambre et deuxièmement la transmission de séquences binaires. Les séquences binaires sont envoyées à une haute fréquence et sont donc invisibles pour les yeux humains. Un appareil optique peut détecter les séquences et mesurer leurs amplitudes. Les amplitudes différentes sont une fonction des distances. Cette propriété permet de calculer la position du récepteur dans la pièce.

Pour l'émetteur, un prototype autonome a été développé. Il est alimenté par le secteur. Un transformateur est utilisé pour piloter la LED. Deux possibilités différentes pour implémenter la séquence binaire sur la lumière envoyée ont été examinées: la première consiste à utiliser l'interrupteur du circuit à transformateur "flyback" et l'autre à placer un interrupteur en série avec le LED. Un circuit programmable FPGA est utilisé pour générer la séquence binaire.

Pour ce qui est du récepteur, une photodiode est utilisée avec un circuit analogique pour détecter, filtrer et amplifier la séquence binaire envoyée. Un convertisseur A/D est utilisé pour numériser l'information reçue. Il est alors possible d'envoyer ces données à un appareil numérique où un programme peut servir à traiter ces données.

Mots-clés: communication par la lumière visible, détermination du placement à l' intérieure, convertisseur flyback, récepteur optique, filtre de corrélation.

Abstract (Nederlands)

Namen: Rob Holvoet , Jelle Smets

Contact: robholvoet@hotmail.com, jelle.smets@hotmail.com

Titel: Verzender op netspanning en draagbare ontvanger voor communicatie met zichtbaar licht: Indoor positiebepaling.

In deze paper zullen de mogelijkheden bestudeerd worden van communicatie met zichtbaar licht (Visible Light Communication, VLC) voor indoor positiebepaling. Het plaatsen van LEDs aan het plafond heeft hier een dubbele functie: enerzijds de verlichting van de ruimte en anderzijds het verzenden van binaire sequenties. De binaire sequenties worden verzonden op een hoge frequentie en zijn dus niet waarneembaar door het menselijke oog. Een optische ontvanger kan deze sequenties detecteren en hun amplitude bepalen. De verschillende amplitudes zijn namelijk in functie van de afstanden. Deze eigenschap laat toe om de positie te bepalen van een ontvanger in een kamer.

Als zender wordt een autonoom prototype gemaakt die onmiddellijk verbonden wordt op de netspanning. Voor het voeden van de led wordt er gebruik gemaakt van een transformator circuit (Flyback). Twee verschillende mogelijkheden worden bekeken om de binaire sequenties te implementeren in het uitgestraalde licht. De mogelijkheid om de schakelaar van de transformator circuit zelf te gebruiken wordt enerzijds onderzocht en anderzijds wordt de mogelijkheid bekeken om de schakelaar in serie te zetten met de LED. Er wordt gebruik gemaakt van een FPGA om de binaire sequenties te genereren.

Wat de ontvanger betreft wordt gebruik gemaakt van een fotodiode samen met een analoog circuit om de verzonden licht sequenties te detecteren, filteren en versterken. Een ADC wordt gebruikt om de ontvangen informatie te digitaliseren. Dit maakt het mogelijk om de data te verzenden naar een digitaal apparaat. Software kan dan gebruikt worden om de data te verwerken.

Trefwoorden: communicatie met zichtbaar licht, indoor positiebepaling, flyback converter, optische ontvanger, correlatie filter

Contents

1	Introduction	1
2	Preparation and Research	3
2.1	Choice of emitting LED and receiving photodiode	3
2.1.1	Introduction	3
2.1.2	LED	3
2.1.3	Photodiode	5
2.1.4	Conclusion	10
3	LED driver circuit	12
3.1	Introduction	12
3.2	Power Supply Topologies	13
3.3	Flyback Converter	13
3.4	LFSR	15
3.4.1	First tests	15
3.4.2	Final LFSR	15
3.5	LED Heatsink	15
3.6	Testing circuit 1	16
3.6.1	Tests	17
3.7	Testing circuit 2	18
3.7.1	Twisted pair cable	18
3.7.2	Tests	19
3.7.3	VHDL of sender FPGA	19
4	The receiver circuit	21
4.1	Introduction	21
4.2	Power supply	22
4.3	Current-voltage converter	23

4.4	Ambient light filter	24
4.5	Amplifier	28
4.6	Anti-aliasing filter	28
4.7	ADC	32
5	Connection to the PC	37
5.1	Introduction	37
5.2	Structure Ethernet communication	37
6	Processing with Python	42
6.1	Receiving the data	42
6.2	Processing the data	44
6.3	Positioning	48
6.3.1	Radius calculation	48
6.3.2	Position calculation	50
6.3.3	Data Transmission	52
7	Result	53
8	Future work	54
A	List of Main Switch Mode Power Supply (SMPS) Topologies	57
B	Calculations to determine the needed heat sink	59
C	Ambient light rejection circuit	61
D	Simulations	66
D.1	ADC	66
D.2	Sending Ethernet frames	68
E	Tests of sending circuit	70
E.1	Results Test 1 of circuit 1	70
E.2	General findings	70
E.3	Changing the duty cycle	70
E.4	Conclusion	74
E.5	Results Test 2 of circuit 1	74
F	Correlation with an n-bit register	77

F.1	n = 6	77
F.2	n = 7	80
F.3	n = 8	81
F.4	n = 9	82
F.5	n = 10	85
F.6	n = 11	86
F.7	n = 12	87
F.8	n = 13	87
F.9	n = 14	88
F.10	n = 15	88
F.11	Conclusion	89
G	Data transmission	90
G.1	One sequence per bit	90
G.2	Two sequences per bit	91
G.3	Five sequences per bit	92
G.4	Conclusion	92
H	Schematics	93
H.1	Schematic 1	93
H.1.1	Rectifier	96
H.1.2	Snubber	97
H.1.3	Mosfet and mosfet driver	98
H.1.4	Power switch chip and analog switch	99
H.1.5	Feedback circuit	100
H.2	Schematic 2	101
H.2.1	Ambient light rejection circuit	105
H.2.2	High pass filter	105
H.2.3	Amplifier	105
H.2.4	Voltage reference	106
H.2.5	LC filter	107
H.2.6	Voltage regulator	107
H.2.7	Separation between analog and digital ground	107
H.2.8	Connection to the FPGA board	107
H.3	Schematic 3	108
I	Code	113

I.1	VHDL Code	113
I.1.1	ADC	113
I.1.2	Ethernet	117
I.1.3	Sequence generation for LEDs	125
I.2	Python Code	133
I.2.1	Ethernet sender	133
I.2.2	Main script	133
I.2.3	Functions	137
I.2.4	Functions testing	150

List of Figures

2.1	Two approaches for generating white emission from LEDs.[11]	3
2.2	Different kinds of white light.[9]	4
2.3	Typical Color Spectrum.[3]	5
2.4	Spectrum of warm-white LED in comparison with the range of the photodiode	6
2.5	Photodiode BPW21R	6
2.6	Short Circuit Current vs. Illuminance	8
2.7	Relative Spectral Sensitivity vs. Wavelength	8
2.8	Photodiode S1223	9
2.9	Sensitivity vs. Wavelength	10
2.10	Model of photodiode	10
2.11	Main characteristics of the BXRA-30E0740-A-00 [3]	11
2.12	Other characteristics of the BXRA-30E0740-A-00 [3]	11
3.1	LED DRIVER	12
3.2	VLC Circuit	13
3.3	Flyback Converter Circuit[7]	14
3.4	3-bit LFSR	15
3.5	Block diagram 1	17
3.6	Block diagram 2	18
3.7	Result twisted pair cable (6m)	19
3.8	Signal from on-board FPGA (yellow) and current trough LED (green)	20
3.9	Block diagrams	20
4.1	Block diagram receiver	21
4.2	Step-up converter (1.2V to 3.3V)	22
4.3	Step-up converter (1.2V to 5V)	23
4.4	Current-voltage converter	24
4.5	RC high pass filter	25

4.6	Output receiver with RC-filter	26
4.7	Ambient light filter circuit with capacitor	26
4.8	Output receiver with condensator	27
4.9	Receiver circuit	27
4.10	Creating a virtual ground for single supply operation	28
4.11	LC filter with $f_{cut-off} = 500$ kHz and $Z_{in} = Z_{out} = 50 \Omega$	29
4.12	LC filter with $f_{cut-off} = 500$ kHz and $Z_{in} = Z_{out} = 50 \Omega$ (PCB)	29
4.13	Spectrum LC filter measured with network analyzer (50 Ω input and output impedance)	30
4.14	Simulation spectrum LC filter with $Z_{in} = 1 \Omega$ and $Z_{out} = 300 \Omega$	30
4.15	Signal before and after filter at 2,3 m (square wave)	31
4.16	Signal before and after filter at 2,3 m (LFSR)	31
4.17	Simulation spectrum LC filter with $Z_{in} = 1 \Omega$ and $Z_{out} = 300 \Omega$ (redesign)	32
4.18	Signal before and after filter at 2,3 m (LFSR) with new LC filter	32
4.19	Timing diagram 12-bit ADC	33
4.20	Signals \overline{CS} and SDATA	34
4.21	FPGA board	34
4.22	THD (dB) vs. Source impedance (Ω)	35
4.23	Input signal ADC when driven by FPGA (LFSR at 2,3m)	35
4.24	Input signal ADC when driven by FPGA measured with probe with ground wire.	36
4.25	PCB receiver	36
5.1	Structure to send and receive packets over Ethernet	38
5.2	Structure to write the samples in RAM and read from RAM	39
5.3	Received UDP packets in Wireshark	40
5.4	Construction of receiver and connection to the PC	41
6.1	Network adapters on PC	42
6.2	ARP table	43
6.3	Sending a UDP packet to FPGA	44
6.4	Block diagram of the <code>main</code> code	44
6.5	Graph received samples (bit sequence of 1023 bits)	45
6.6	Graph received samples after correlation filter (bit sequence of 1023 bits)	47
6.7	Auto-correlation for a 10-bit sequence	47
6.8	Comparison between received signal and ideal signal	48
6.9	Fitting correlation value vs. radius for first LED	50

6.10	intersection points of 2 circles	50
6.11	Position calculation (graphical view)	52
7.1	Graphical view of the position of the receiver in the room	53
B.1	Thermal circuit[4]	59
C.1	Ambient light rejection circuit	61
C.2	Closed-loop system	62
C.3	Feedback circuit	62
C.4	Closed-loop system with addition	63
C.5	Bode plot $H_{tot}(s)$	64
D.1	Testbench ADC	66
D.2	Simulation ADC	67
D.3	Testbench Ethernet	68
D.4	Simulation sending Ethernet frames	69
E.1	Rectified voltage(Yellow) and input current(green)	71
E.2	Primary current(Yellow) and secondary current(green)	71
E.3	Primary current(green) and rectified input voltage(Yellow)	72
E.4	Primary current(Yellow) and secondary current(green)	72
E.5	Voltage over the LED (Yellow) and current through the LED(green)	73
E.6	Voltage over the LED (Yellow) and current through the LED(green)	73
E.7	extra circuitry on breadboard	75
E.8	Current through LED(Yellow) and voltage over the LED (17.8V to 30.7V)(green)	75
E.9	Current through LED(Yellow) and signal from FPGA (green)	76
E.10	Current through LED(Yellow) and signal at gate(green))	76
F.1	Graph received signal after correlation filter (6-register LFSR), one LED burning	78
F.2	Graph received signal after correlation filter 1 (6-register LFSR), 5 LEDs burning	78
F.3	Graph received signal after correlation filter 2 (6-register LFSR), 5 LEDs burning	79
F.4	Graph received signal after correlation filter 3 (6-register LFSR), 5 LEDs burning	79

F.5	Graph received signal after correlation filter 1 (7-register LFSR bit LFSR), 5 LEDs burning	80
F.6	Graph received signal after correlation filter 2 (7-register LFSR), 5 LEDs burning	80
F.7	Graph received signal after correlation filter 3 (7-register LFSR), 5 LEDs burning	81
F.8	Graph received signal after correlation filter 1 (8-register LFSR bit LFSR), 5 LEDs burning	81
F.9	Graph received signal after correlation filter 2 (8-register LFSR), 5 LEDs burning	82
F.10	Graph received signal after correlation filter 3 (8-register LFSR), 5 LEDs burning	82
F.11	Graph received signal after correlation filter 1 (9-register LFSR), 5 LEDs burning	83
F.12	Graph received signal after correlation filter 2 (9-register LFSR), 5 LEDs burning	83
F.13	Graph received signal after correlation filter 3 (9-register LFSR), 5 LEDs burning	84
F.14	Graph received signal after correlation filter (10-register LFSR), one LED burning	85
F.15	Graph received signal after correlation filter 1 (10-register LFSR), 5 LEDs burning	85
F.16	Graph received signal after correlation filter 2 (10-register LFSR), 5 LEDs burning	86
F.17	Graph received signal after correlation filter 3 (10-register LFSR), 5 LEDs burning	86
F.18	Graph received signal after correlation filter (11-register LFSR), one LED burning	87
F.19	Graph received signal after correlation filter (12-register LFSR), one LED burning	87
F.20	Graph received signal after correlation filter (13-register LFSR), one LED burning	88
F.21	Graph received signal after correlation filter (14-register LFSR), one LED burning	88
F.22	Graph received signal after correlation filter (15-register LFSR), one LED burning	89
G.1	Graph received signal after correlation filter (10-bit LFSR), one LED burning	91
G.2	Graph received signal after correlation filter (10-bit LFSR), 5 LEDs burning	91

G.3	Graph received signal after correlation filter (10-bit LFSR), one LED burning	92
H.1	PCB sender schematic 1, without unneeded parts	93
H.2	Bridge rectifier and reservoir capacitors	96
H.3	Snubber	97
H.4	Mosfet with mosfet driver	98
H.5	Power switch chip	99
H.6	Feedback circuit for powerswitch	100
H.7	PCB receiver	101
H.8	Comparison output signal with different cut-off frequency of HPF	105
H.9	PCB receiver with some modifications	106
H.10	PCB sender schematic 2, with FPGA mounted	108

List of Tables

2.1	Characteristics of BPW21R photodiode	6
2.2	Characteristics of S1223 photodiode	9
3.1	Comparative table between different switch uses	17
6.1	Measurements radius versus correlation value with each LED	49

List of symbols and acronyms

Symbols

$f_{cut-off}$ Cut-off frequency

Acronyms

ADC	Analog-to-digital converter
ARP	Address Resolution Protocol
ESR	Equivalent Series Resistance
FIFO	First in, first out
FPGA	Field-programmable gate array
LED	Light Emitting Diode
LFSR	Linear Feedback Shift Register
LSB	Least Significant Bit
MII	Media Independent Interface
MSB	Most Significant Bit
RAM	Random Accessible Memory
THD	Total Harmonic Distortion
UDP	User Datagram Protocol
VLC	Visible Light Communication
PD	Photodiode

Chapter 1

Introduction

LEDs are semiconductor devices that emit light when biased in the forward direction of the p-n junction. LEDs present many advantages over traditional light sources, including lower energy consumption, longer lifetime, improved robustness and smaller size. A significant attribute of LEDs is their ability to switch on and off thousands of times per second. No other lighting technology has this capability. This switching occurs at ultra-high speeds, so far beyond what the human eye can detect, that the light appears to be constantly on. These embedded signals are emitted from the LEDs in the form of binary code; 'off' equals zero and 'on' equals one. This is what they call visible light communication (VLC). When VLC equipment and devices are placed throughout a building or geographical area, a comprehensive wireless communication network can be created.

This thesis talks about a stand alone transmitter and a receiver that can be connected to a computer. The transmitter can be built in the ceiling and only needs 230 AC voltage to transmit digital information. The receiver will be a device to plug in a digital port of the computer. It receives information from the emitted light and sends it to the PC. The transmitted information will give the receiver the possibility to determine his current position in a room by looking at the amplitude of the signals he receives from multiple LEDs. It can also be used to transmit other digital information/files.

The structure of this report is as follows:

In chapter 2 the choice of the used LED and photodiode for the transmission is explained. Then we split the hardware research for the transmitting circuitry (chapter 3) and the receiving circuitry (chapter 4).

Chapter 3 contains information starting with some research about the chosen components and topologies. Further on it talks about the first circuitry and tests until the final board and his operation.

Chapter 4 talks about the different parts of the receiver. First, we will tell how to supply the circuit. Afterwards, we will explain the receiver circuit starting from the photodiode until the ADC used to digitalize the analog signal.

In chapter 5 we have a look how the connection with the PC is made. This contains the VHDL code used to process the data from the receiver circuitry in the FPGA and send it to the PC. At the side of the PC, we use some software to obtain that information. The processing in software is explained in chapter 6.

The final result of our thesis is presented in chapter 7. The things we couldn't do due to the time limit and the ideas we have in mind for future work are given in chapter 8.

Source code, schematics, calculations, documentation and extensive test results are found in the appendices.

Chapter 2

Preparation and Research

2.1 Choice of emitting LED and receiving photodiode

2.1.1 Introduction

To start this project, we first had to look for the proper LED and photodiode that we would use. We need those specifications to know what spectrum we will use to transceive, what electrical circuit we need to control the LED and to process the received data through the photodiode.

2.1.2 LED

As we are researching an application for indoor use and we want to use LEDs to illuminate a room and to transmit data, we will work with white light. Two approaches are generally used to generate white light with LEDs. The first approach is to combine light from, e.g.,

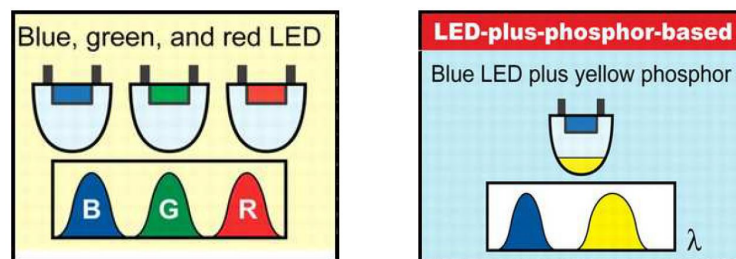


Figure 2.1: Two approaches for generating white emission from LEDs.[11]

red, green and blue (RGB) LEDs [11]. Typically, these triplet devices consist of a single package with three emitters and combining optics, and they are often used in application where variable color emission is required. These devices are attractive for VLC as they

offer the opportunity for transmitting different data on each LED. The other technique is to use a single blue LED which is coated with, or sometimes embedded in, a layer phosphor that emits red-shifted light upon absorbing a portion of blue light emitted by the LED. The red-shifted emission mixes additively with the nonabsorbed blue component to create the required white color (see Figure 2.1). At present, the later approach is often favored due to the lower complexity and cost. Nevertheless, in single-chip devices the phosphor typically limits the speed of overall optical response. Because our design is only a prototype and the limiting factor for data transmitting speed will be determined by the receiver circuit (limited bandwidth), we don't have to take this factor into account.

In the domain of white light you can choose 3 main types of white light LED's : Cool, Warm and Natural White LEDs (see Figure 2.2)

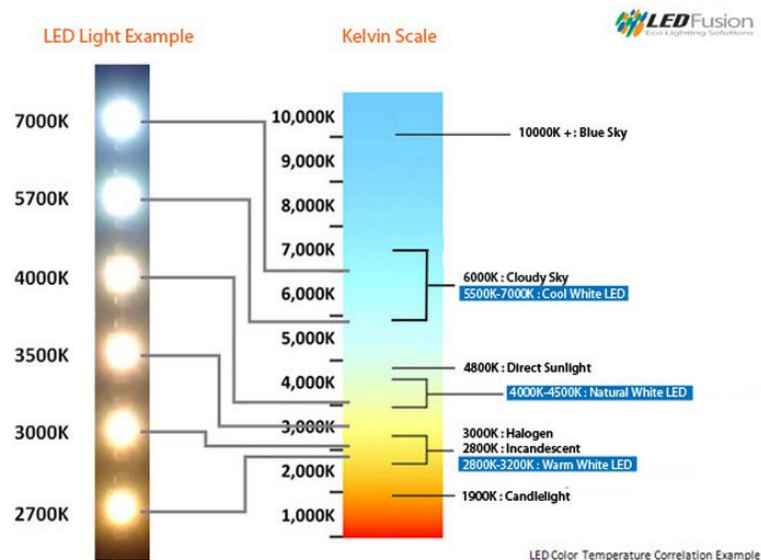


Figure 2.2: Different kinds of white light.[9]

With traditional incandescent or halogen bulbs you could not choose the colour of the light that the bulb produced as this was down to pure physics. Traditional bulbs work by passing electricity through a wire that glows hot. This glow produces the light that can be seen. Because LED bulbs are a completely different technology and much more technically advanced it is possible to choose the colour of the light that you want.

The colour temperature of light can be measured as a number on the Kelvin scale. This is represented by a number followed by the symbol 'K'. High colour temperatures of over 5000 K are said to be cool colours and often have a blue tone to them. Whereas low colour

temperatures of 2,500 - 3,200 are said to be warm colours and have yellow or orangey tones to them.[13]

In domestic lighting, 75 - 80 % of customers buy warm white LED bulbs as they offer the closest match to their old incandescent or halogen bulbs that they are replacing. The light given off by a warm white LED bulb can be described as soft and warm with a slight yellow hint to it.

Some customers purchase cool white LED bulbs if they want the brightest light possible or if they want to achieve a modern look to a room. This particularly suits colour schemes that are bright and bold and use primary colours, or even colour schemes that are predominantly white where a clean minimal look is required.

In commercial lighting the majority of customers opt for cool white bulbs. These give a brighter light but you should be careful to ensure that the colour temperature is not too high, or in practical terms the light not too cool, otherwise the effect may seem very harsh or displeasing to the eye.

These 3 main different white LEDs have a different Intensity division in the spectral band.(See Figure 2.3)

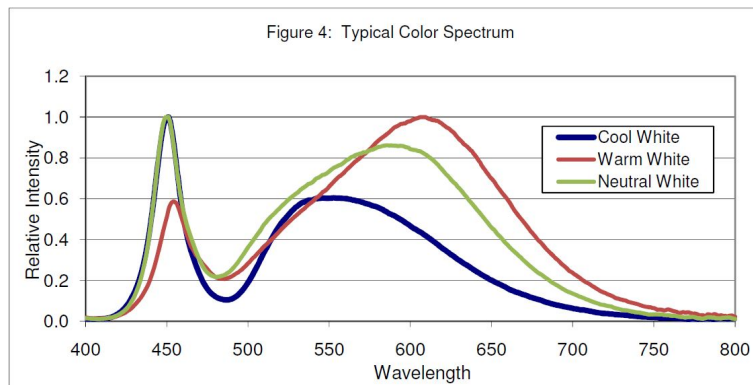


Figure 2.3: Typical Color Spectrum.[3]

2.1.3 Photodiode

The first part of a VLC receiver, is a photosensitive element. Mostly, a photodiode is used at the input of the receiver. The current through the photodiode, is proportional to the light falling on it. The more light it receives, the larger the current will be.

The photodiode has to operate in the same spectrum as the LED. We tested two photodiodes:

the BPW21R photodiode from Vishay and the S1223 photodiode from Hamamatsu. We will discuss both diodes and give the most important characteristics.

- In figure 6.8, you can see the spectrum of the chosen warm-white LED in comparison with the range of the BPW21R photodiode [12] (see figure 2.5). As you can see, the photodiode is sensitive in the wavelength spectrum of the LED.

Wavelength Characteristics at Rated Test Current, $T_j=25^\circ\text{C}$

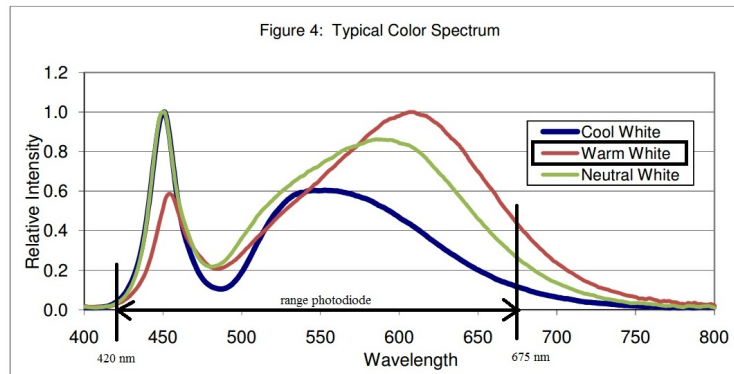


Figure 2.4: Spectrum of warm-white LED in comparison with the range of the photodiode

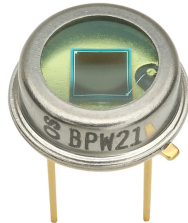


Figure 2.5: Photodiode BPW21R

Hereby, We summarized the most important characteristics of the photodiode BPW21R in table 2.1:

Table 2.1: Characteristics of BPW21R photodiode

Parameter	Value	Unit
<i>Sensitive area</i>	7.5	mm ²
<i>Short circuit current at 1 klux</i>	9	μA
<i>Sensitivity</i>	9	$\frac{\mu\text{A}}{\text{lux}}$
<i>Wavelength of peak sensitivity</i>	565	nm
<i>Range of spectral bandwidth</i>	420 to 675	nm
<i>Viewing angle</i>	± 50	°
<i>Rise time</i>	3.1	μs
<i>Fall time</i>	3.0	μs
<i>Diode capacitance if V_R=0 V</i>	1.2	nF

The photodiode has a large sensitive area, what is interesting to capture enough light from the LED. The current will increase pretty fast when more light is falling on the diode because of its high sensitivity. The highest sensitivity takes place on a wavelength of 565 nm, which is in the spectrum of the LED. The rise time and fall time of a photodiode is defined as the time for the signal to rise or fall from 10% to 90% or 90% to 10% of the final value respectively. The bandwidth of the photodiode can be estimated using the formula:

$$\text{BW} \approx \frac{0.35}{t_{\text{rise}}} \approx \frac{0.35}{3.1 \mu\text{s}} \approx 112.9 \text{ kHz}$$

The BPW21R has also a wide viewing angle, so it can pick up light from several directions. The capacitance of the diode can play an important role at higher frequencies.

In the figure below, you can see the graph of the short circuit current in function of the illuminance. As expected from a photodiode, the current is proportional to the incident light. An interesting thing to see, is that the photodiode has to be in short circuit for linear working.

The relative spectral sensitivity versus the wavelength is shown in figure 2.7. We see that the range of spectral bandwidth (where $S(\lambda)_{\text{rel}} = 0.5$) goes from 420 nm to 675 nm, as already mentioned in the characteristics. They also show the spectral response of the human eye $V_{\lambda} \text{Eye}$. The human eye is also a photosensitive element. It is most sensitive at a wavelength of 560 nm. The BPW21R photodiode gives thus an approximation to the spectral response of the human eye.

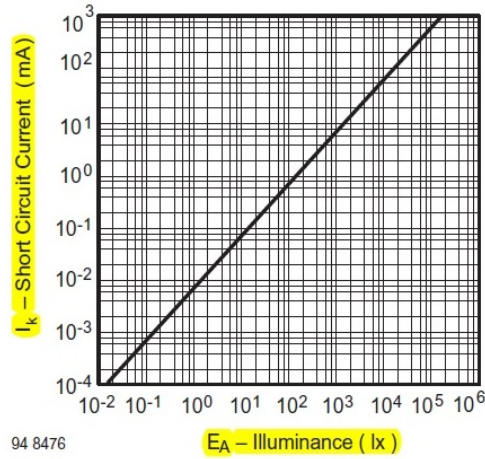


Figure 2.6: Short Circuit Current vs. Illuminance

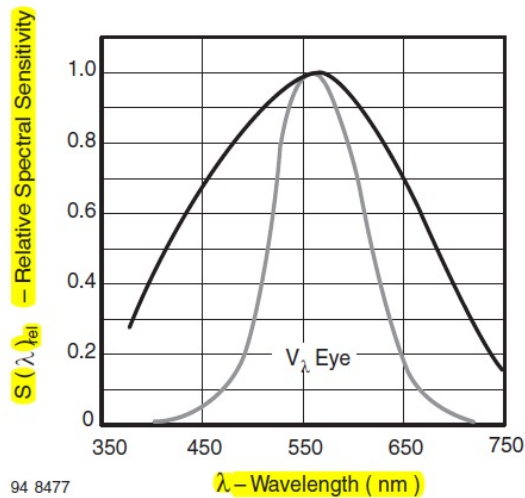


Figure 2.7: Relative Spectral Sensitivity vs. Wavelength

- The BPW21R photodiode is a good choice as a receiver for the visible light communication, but can give some problems at higher frequencies because of the limited bandwidth and the capacitance. So, we also searched for another photodiode to encounter these problems. We found the S1223 photodiode from Hamamatsu [6] (see figure 2.8).

We summarized the most important characteristics of the S1223 photodiode in

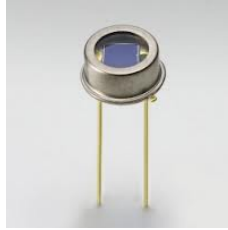


Figure 2.8: Photodiode S1223

table 2.2:

Table 2.2: Characteristics of S1223 photodiode

Parameter	Value	Unit
<i>Sensitive area</i>	6.6	mm ²
<i>Short circuit current at 100 lux</i>	6.3	μA
<i>Sensitivity at 660 nm</i>	0.45	$\frac{A}{W}$
<i>Wavelength of peak sensitivity</i>	960	nm
<i>Range of spectral bandwidth</i>	320 to 1100	nm
<i>Bandwidth</i>	30	MHz
<i>Diode capacitance if $V_R=20 V$</i>	10	pF

If we compare these characteristics with the characteristics of the other photodiode, we can notice some things:

The range of spectral bandwidth is larger. The photodiode is also sensitive in the infrared region. The diode is most sensitive in this region, but he is also sensitive in the spectrum of the LED. So, we can also use this photodiode as a receiver because we will filter the ambient light. This photodiode is more appropriate at higher frequencies because of the higher bandwidth and the smaller capacitance. The rise time can now be estimated using the formula:

$$t_{\text{rise}} \approx \frac{0.35}{\text{BW}} \approx \frac{0.35}{30\text{MHz}} \approx 11.667 \text{ ns}$$

In figure 2.9, you can see the spectral response of the photodiode.

As already mentioned, the diode is most sensitive in the infrared region.

If we want to simulate a circuit with a photodiode in LT Spice, a photodiode is not available in the component library. So, we will use an equivalent model that represents the photodiode. First, we used a normal diode in parallel with a current

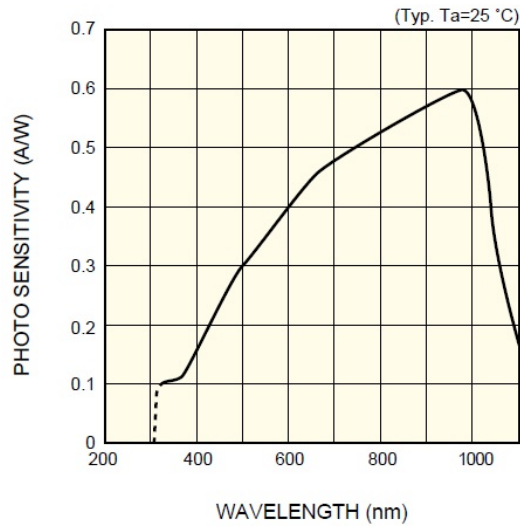


Figure 2.9: Sensitivity vs. Wavelength

source to model the photoelectric effect. The more light falling on the photodiode, the more current will flow through it. However, the results from the simulations and the measurements were not very similar. Then we modified the model by placing a capacitor in parallel with the diode and the current source. The capacitor represents the photodiode capacitance. You can find this value in the datasheet. Now the results were much better. Simulations and measurements were very similar. Eventually we can also add the dark resistance of the diode. But this is usually very high ($\approx 30 \text{ G}\Omega$), so this is not very necessary. Finally we used the model below in LT Spice to represent the photodiode.

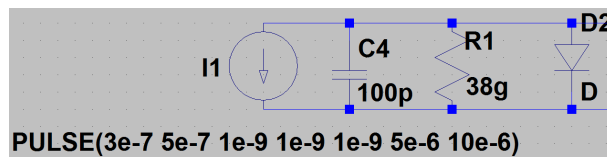


Figure 2.10: Model of photodiode

2.1.4 Conclusion

Because of the lowcost, the lower complexity and since we will transmit data over the whole visible spectrum, we chose the blue led with yellow phosphor coating approach. Whereas we want to implement our application in daily life, we chose to work with warm

light. The warm light has the closest match to the old incandescent or halogen bulbs that they are replacing and it has more intensity around 565 nm. After a chat with Frederic Truffer who works in the light department of the HES-SO in Sion we decided to use the BXRA-30E0740-A-00 star array led. This BXRA reaches a Typical Pulsed Flux of 860 lm. This should be enough to be detected by the photodiode at a distance of several metres. The characteristics of the chosen LED are listed in figures 2.11 and 2.12.

We chose to work with the S1223 photodiode because of the high bandwidth and the small capacitance. We will transmit data at $200 \frac{\text{kbits}}{\text{second}}$, so that the photodiode has enough time to respond to the incoming light. Because of the high sensitivity and the large sensitive area, the distance between the LED and the photodiode should be no problem. The photodiode has a sensitive range that lies in the wavelength spectrum of the LED. If we consider this, the S1223 photodiode should be an appropriate photodiode for the visible light communication.

Values for test with testcurrent 350 mA	
Typical Pulsed Flux Tj 25°C	860 lm
Typical DC Flux Tcase 70°C	770 lm
Vf (Typ)	28,1 V
Power (Typ)	9,8 W
Efficacy (Typ at Tj 25°C)	88 lm/W

Figure 2.11: Main characteristics of the BXRA-30E0740-A-00 [3]

Characteristics	Minimum	Typical	Maximum	Unit
Color Temperature	2870	3045	3220	K
Viewing angle		120		Degrees
Typical Center Beam Candle Power		260		cd
Forward Voltage at test c	25,3	28,1	30,9	V
DC Forward Current			500	mA
Peak Pulsed Current			700	mA
Reverse Voltage			-45	V

Figure 2.12: Other characteristics of the BXRA-30E0740-A-00 [3]

Chapter 3

LED driver circuit

3.1 Introduction

To power the LED with the mains we first need to convert the 230 V_{AC} to a DC voltage and control the current through the LED. On the market you can buy a readily available LED driver, for our led we could use the 3-Watt MagTech LED Driver L03U-350 (see Figure 3.1).

3-Watt MagTech LED Driver Features:

- 100 ~ 277VAC Input
- 4 ~ 12VDC Output
- 350mA Output Current
- 1 ~ 3 Watts Output Power
- 1.57"L x 1.57"W x 0.87"H
- UL and CE Certified
- Short Circuit & Overload Protection

Figure 3.1: LED DRIVER

We could use this driver or make our own driver and use this together with a switch for our VLC application (see Figure 3.2).

Most drivers use Switch Mode Power Supply (SMPS) topologies where the output voltage or current is determined by the frequency at what a switch is turned on and off. We wonder if it is possible to use this switch instead of an extra switch after the AC/DC converter to send information in a VLC application and control the output current at the same time. So we will construct our own LED driver and control the current through the LED. At first we will investigate the possibilities of sending data by controlling the switch of a Switch Mode Power Supply with an FPGA. The use of a switch after the LED driver will also be

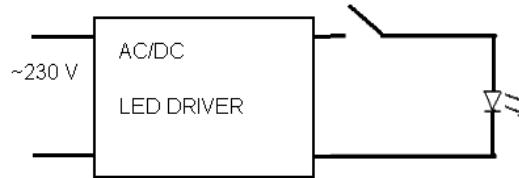


Figure 3.2: VLC Circuit

investigated.

3.2 Power Supply Topologies

There are a lot of Power Supply Topologies we could use for our application. We must use an Switch Mode Power Supply as we want to control the switch for sending our data. In general this method is more interesting in comparison with series-controlled regulators that don't use a switch, looking at the power loss. A list with a short description of these SMPS topologies is given in appendix A.

We will use the flyback converter due to his low cost, simplicity of design, small size and intrinsic efficiency. Other advantages of the flyback transformer over circuits with similar topology include isolation between primary and secondary and the ability to provide multiple outputs.

3.3 Flyback Converter

A flyback converter is a transformer-isolated converter based on the basic buck boost topology. The basic schematic and switching waveforms are shown in Figure 3.3. In a flyback converter, a switch (Q_1) is connected in series with the transformer (T_1) primary. The transformer is used to store the energy during the ON period of the switch, and provides isolation between the input voltage source V_{IN} and the output voltage V_{OUT} . In a steady state of operation, when the switch is ON for a period of T_{ON} , the dot end of the winding becomes positive with respect to the non-dot end. During the T_{ON} period, the diode D_1 becomes reverse-biased and the transformer behaves as an inductor. The value of this inductor is equal to the transformer primary magnetizing inductance L_M , and the stored magnetizing energy from the input voltage source $V_{IN} \cdot E_p = \frac{1}{2} \cdot I_{PK}^2 \cdot L_M$ & $I_{PK} = \frac{V_{IN} \cdot T_{ON}}{L_M}$

Therefore, the current in the primary transformer (magnetizing current I_M) rises linearly from its initial value I_1 to I_{PK} , as shown in 3.3(D). As the diode D_1 becomes reverse-biased, the load current (I_{OUT}) is supplied from the output capacitor (C_O). The output capacitor

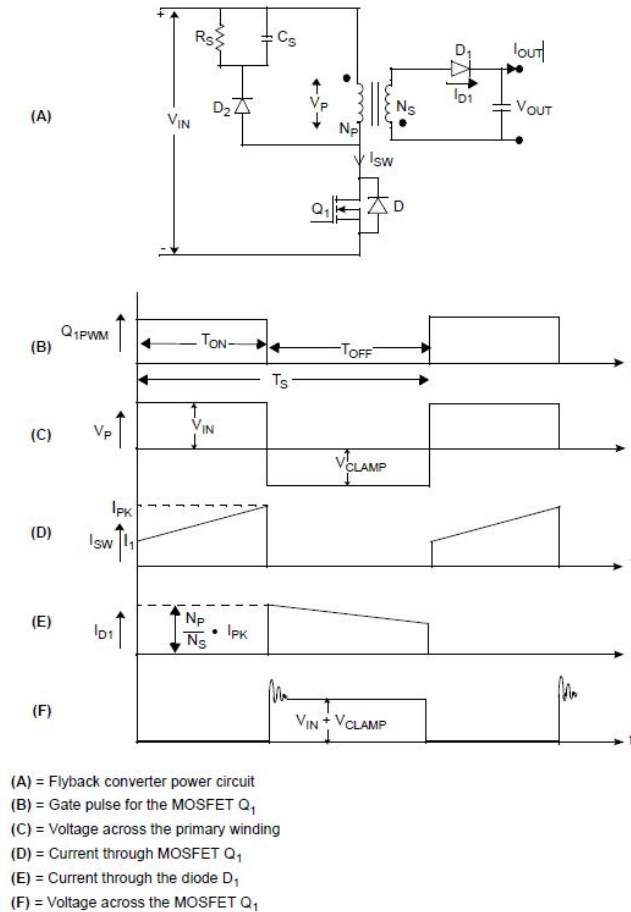


Figure 3.3: Flyback Converter Circuit[7]

value should be large enough to supply the load current for the time period T_{ON} , with the maximum specified drop in the output voltage. At the end of the T_{ON} period, when the switch is turned OFF, the transformer magnetizing current continues to flow in the same direction. The magnetizing current induces negative voltage in the dot end of the transformer winding with respect to non-dot end. The diode D_1 becomes forward-biased and clamps the transformer secondary voltage equal to the output voltage. The energy stored in the primary of the flyback transformer transfers to secondary through the flyback action. This stored energy provides energy to the load, and charges the output capacitor. Since the magnetizing current in the transformer cannot change instantaneously at the instant the switch is turned OFF, the primary current transfers to the secondary, and the amplitude of the secondary current will be the product of the primary current and the transformer turns ratio, N_P/N_S .

3.4 LFSR

To control the mosfet for sending signals with the LED we need a certain signal to be generated. We chose to use a semi-random generated signal. For this we use a shift register expanded with xor functions (this is an example of an LFSR).

3.4.1 First tests

For the first tests we chose to work with a 3-bit LFSR (figure 3.4). This LFSR generates a bit sequence of 0,0,1,0,1,1,0.

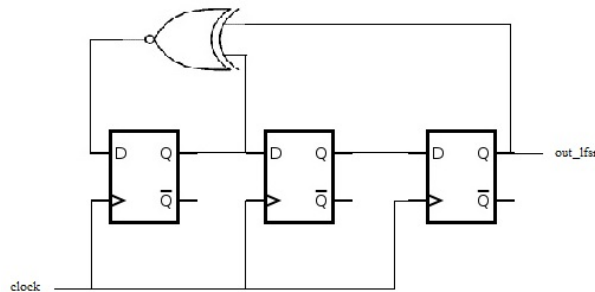


Figure 3.4: 3-bit LFSR

We can program this LFSR in an FPGA. We used an FPGA board available at school with a Spartan XC3S500E FPGA from Xilinx on it to do the testing.

3.4.2 Final LFSR

As will be explained in the further text, we will only need a semi-random LFSR to be generated by a FPGA we will mount on the board. To be able to detect these LFSRs, we will use correlation. Later on we will test the response for LFSRs with different lengths. We always use maximum feedback xor combinations to generate a binary sequence with a length of 2^{m-1} bits, where m is the length of the used register. [8]¹

3.5 LED Heatsink

When a voltage is applied across the junction of an LED, current flows through the junction generating light. It is a common misconception that LEDs don't generate heat. While

¹a list of taps for maximal feedback can be found on http://www.newwaveinstruments.com/resources/articles/m_sequence_linear_feedback_shift_register_lfsr.htm

essentially no heat is generated in the light beam (unlike conventional light sources), heat is generated at the junction of the LED Array and must be effectively managed. As LEDs are not 100 % efficient at converting input power to light, some of the energy is converted into heat and must be transferred to the ambient. The amount of heat generated from the LED Array that must be transferred to the ambient may be conservatively estimated to be 85 % of the power that is applied to the LED Array and is calculated as in Equation 3.1

$$P_d = V_f \cdot I_f \cdot 0.854 \quad (3.1)$$

Where:

P_d is the thermal power to be dissipated

V_f is the forward voltage of the device

I_f is the current flowing through the device

The power calculation should be made for maximum dissipated power, which is computed using the Maximum V_f at the drive current desired.

Heat generated at the LED junction must be transferred to the ambient via all elements that make up the thermal management solution. These elements include the LED Array, the thermal interface material used between the LED Array and heat sink, the heat sink, the luminaire enclosure (if applicable), and other components that come in contact with the lighting assembly. These elements transfer heat to the ambient. We will use a heat sink to manage the temperature at the case of the LED array. The calculation of the maximum needed thermal resistance of the heat sink is included in appendix B.

We chose to use the *SA-LED-151E - HEATSINK, LED, 50.8MM* of OHMIT which have a thermal resistance of $3.2^{\circ}\text{C}/\text{W}$, which is way below the maximum of $11.85^{\circ}\text{C}/\text{W}$. This heat sink should keep the LED junction on a temperature below 70.5°C and the casing on a temperature below 57.01°C .

3.6 Testing circuit 1

For the first tests, we designed a transformer circuit (schematic included in annex H.1) where we implement a power switch chip for the start up cycle and also to do the first tests. With a switch we can turn of the function of the POWER switch and give the control of the flyback to the FPGA. For the feedback of the power switch we use an analog feedback circuit, for the feedback to the FPGA we can expand our PCB with a ADC to measure the current through the LED. By reading this current, the FPGA will interact by changing the duty cycle of the control signal for the transformer switch when needed.

3.6.1 Tests

First test

For the first test, we only looked at the functionality of the transformer operating with the power switch chip. The block diagram is given in fig 3.5. The results are discussed in appendix E.1.

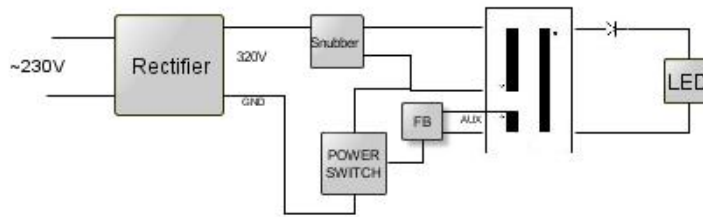


Figure 3.5: Block diagram 1

As seen in appendix E.1, a custom made transformer should be calculated to have a maximum efficiency and maximum luminance. Due to the restriction of time and great advantages (shown in table 3.1) of the topology when placing a switch in serie with a LED driver (see following sections) we will continue by using the topology where we place a switch after the flyback.

Table 3.1: Comparative table between different switch uses

Property	using switch of flyback	using switch in series with LED
Transformer	custom made	general transformer can be used with tolerance in values
Design	multiple feedbackloops needed, change of flyback switch after startup, larger circuitry and more components	relatively simple design
Transmission frequency	fixed and determined by transformer	can be chosen by the user

Second test

We choose to obtain the same circuitry, only using the power switch chip and placing a capacitor in parallel with the LED as load storage. A switch will be used to generate current pulses trough the LED. For this tests we use a external programmed FPGA (We place it on

the same ground of our circuitry) to control the switch. This is shown in block diagram 3.6. The results are shown in appendix E.5.

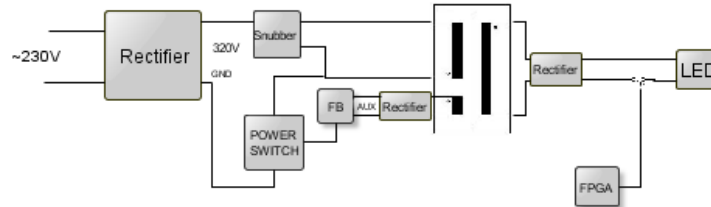


Figure 3.6: Block diagram 2

As the results of the second test met our expectations, we designed a new circuitry that will be discussed in the next paragraph.

3.7 Testing circuit 2

For the new circuit, we will mount an FPGA on the board. We will use one FPGA to generate the binary sequences for all the different LEDs to obtain a lower cost and a synchronization of the different LEDs. We expect that the synchronisation of the LEDs will make it easier to distinguish the different bit sequences in the received signal. We used the Actel igloo AGL 060V5 -VOG100. Because of his PROM memory we don't have to program it every time we power it. The Actel igloo don't support complicated designs, what we don't need as we just want to generate LFSRs. This FPGA was used in a previous project at HES-SO Valais/Wallis what makes it easier to implement it in our design. We used that previous design as a template for our application. We used only a part of testing circuit 1 and added the FPGA, a DC/DC converter(to power the FPGA), a mosfet (as switch in serie with the LED) and an extra storage capacitor for a normal working of the Flyback. The schematic is added in appendix H.3. We use one schematic for the PCBs with and without FPGA. All the components with * (snm) don't have to be mounted on the PCBs without FPGA. The VHDL code and block diagrams used to program the FPGA can be found in section 3.7.3.

3.7.1 Twisted pair cable

To drive the MOSFETs of the different LED circuits, we need to transport a blockwave on different places in the room. To transport these signals we have to look how we can transport them without great distortion or noise. The simplest way to do this is to send through a twisted pair cable. We test the deformation of the signal on a twisted pair cable of 6 meter (largest distance of 2 LEDs in a room). The result is given in figure 3.7. We see a

ringing effect at the end of the cable. As this ringing only lasts for about 300 ns and our bit period is 5 μ s, this signal is only used as switch and we already have noise and peaks in our transmitted light signal (which are not detected by the receiver). This result is sufficient. We will connect the output of the FPGA directly with the mosfets in series with the LED. If we would work on a higher frequency, we would have to take this ringing into account.

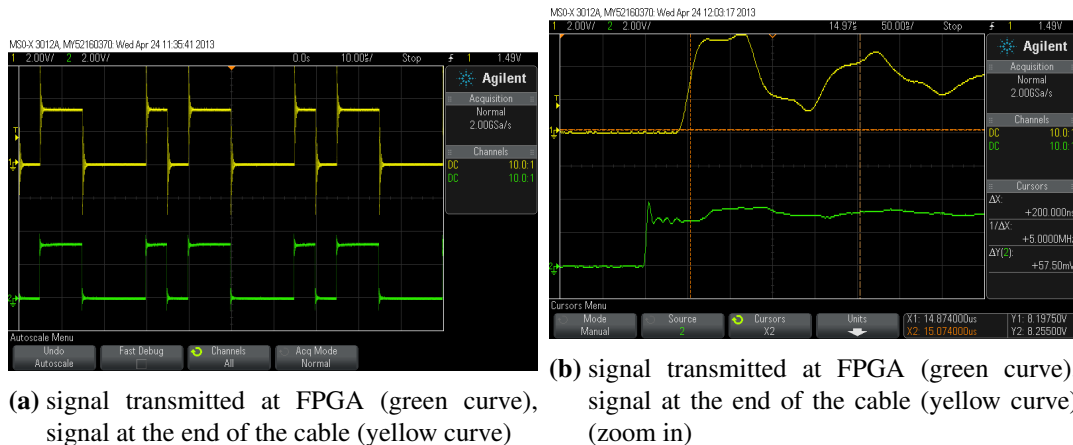


Figure 3.7: Result twisted pair cable (6m)

3.7.2 Tests

We regulate the potentiometer to get a current of maximum 500 mA through the LED and program the FPGA to generate LFSRs. In figure 3.8 you can see the current through the LED in comparison with the signal sent by the FPGA. We can see that we have some ripple and spikes on the current signal. We won't try to prevent this because this has no effect on the received signal by the receiver circuit. To regulate the LEDs on the same intensity we can look on the oscilloscope with a current probe. But we will check on every board by looking at V_{AUX} (here 16.5 V). We checked and tested the resistance of the potentiometer and it was each time around 15 k Ω .

3.7.3 VHDL of sender FPGA

As already mentioned we generate sequences with a FPGA. For each led we need a different LFSR (3.9a). The MSB of the LFSR is sent to the mosfet of each LED which get in conduction (1) or isolation(0) mode. When the MOSFET is in conduction mode, current flows through the LED and the LED emits light. When the MOSFET is in isolation mode, no current can go through the LED so the LED is not illuminating. For the readability of the code we wrote all the LFSRs in one block and used this for every LED. The correct LFSR is chosen by the number of the LED the block gets as a generic. The code can be found in appendix I.1.3.

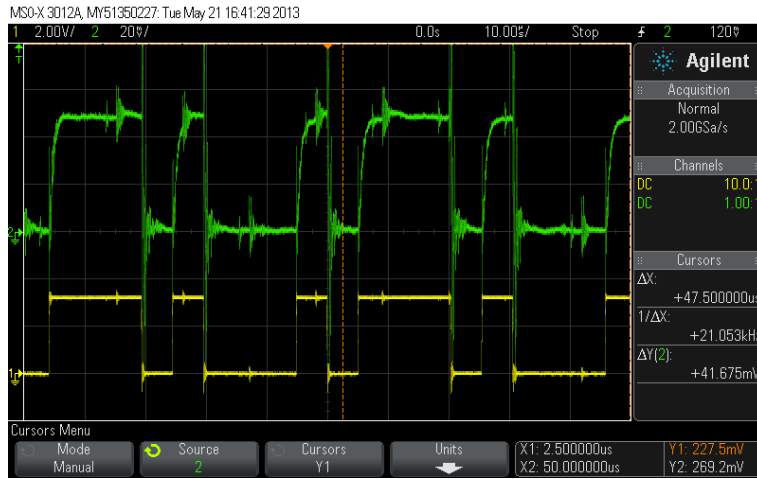
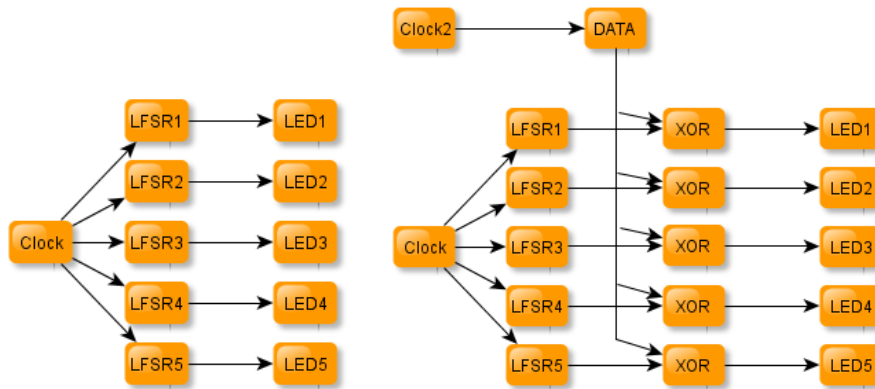


Figure 3.8: Signal from on-board FPGA (yellow) and current trough LED (green)

We also did some tests in appendix G to send data over these LFSRs by just inverting the sequences. The block diagram is shown in figure 3.9b. The second clock represent a clock with a frequency of $\frac{f_{clock}}{2^{n-1} * m}$ where n is the length of the register of the LFSR and m is a natural value. We did the tests by shifting a sequence and 'xor'ed this with the output we already had. The code is not included in this report because of its simplicity.



(a) LFSR sending sequence generation (b) LFSR sending sequences generation with data.

Figure 3.9: Block diagrams

Chapter 4

The receiver circuit

4.1 Introduction

We already discussed the photodiode we will use for the visible light communication. As mentioned before, the short-circuit current through the photodiode is proportional to the incident light. If we want to know the bits that are transmitted, we have to use several blocks. First of all, we will use a current-voltage converter, so that the voltage is also proportional to the incoming light. We also need a high pass filter to filter out the ambient light and the 100 Hz frequency components coming from the lighting in the building. Since the signal we will receive is weak, we will have to use an amplifier. Then, we also need an ADC to digitalize the received signal. Before the ADC, an anti-aliasing filter is needed. Finally, the digital samples have to be transmitted to a PC so we can use software to make some calculations and to find the position. We will use an Ethernet cable for the connection.

The block diagram of the receiver circuit is shown below. We will discuss the several blocks and the power supply as well in this chapter. The Ethernet connection and the data processing on the PC will each be discussed in a separate chapter.

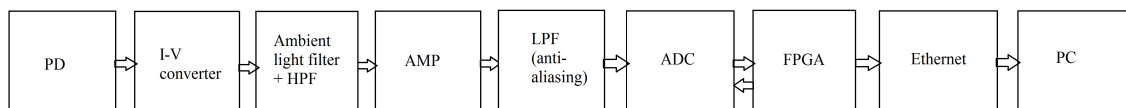


Figure 4.1: Block diagram receiver

4.2 Power supply

Since we want to make a handheld device, a possibility to supply the circuit is a battery. A normal battery delivers a voltage of 1.2 V or 1.5 V, what is not enough for the power supply. We can use a step-up converter (boost converter) to obtain a larger voltage. There are several good chips you can use for this purpose. We found a circuit to boost the 1.2 V to 3.3 V in the datasheet of the *MCP 1640* of Microchip ¹. This 3.3V can be enough to supply the circuit. You can see the circuit of the step-up converter in figure 4.2.

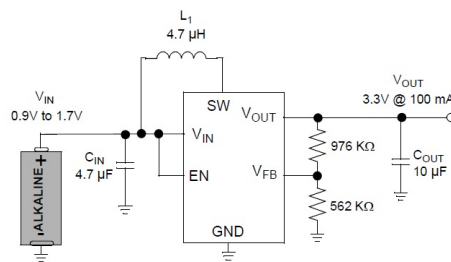


Figure 4.2: Step-up converter (1.2V to 3.3V)

We have to put some components around the MCP 1640 chip. The input capacitor of 4.7 μF is recommended to decouple the voltage source. For low power applications, this capacitance is sufficient at the input. We have to use an X5R ceramic capacitor with a low ESR. The inductor of 4.7 μH is an essential element for a step-up converter. It is a buffer for electrical energy and it is needed if we want a higher output voltage than the input voltage. To achieve a high efficiency, we also need an inductor with a low ESR. To calculate the values of the resistors (voltage divider), the datasheet gives the formula:

$$R_{TOP} = R_{BOTTOM} \cdot \left(\frac{V_{OUT}}{V_{FB}} - 1 \right)$$

If we want $V_{OUT} = 3.3 \text{ V}$, and with $V_{FB} = 1.21 \text{ V}$ and $R_{BOTTOM} = 300 \text{ k}\Omega$, we find: $R_{TOP} = 518.18 \text{ k}\Omega \approx 510 \text{ k}\Omega$.

The output capacitor of 10 μF ensures a stable output voltage during sudden load transients and reduces the voltage ripple. To have a low ripple, we need a capacitor with a low ESR. Again, we can use an X5R ceramic capacitor.

The question is now whether this voltage is enough to supply the op-amps we will use in our circuitry. So, we first searched for the op-amps and see what supply they need. We chose the AD817AN op-amp (see further). This op-amp cannot be supplied with a single 3.3 volt. We supplied it with a single 5 volt. Another way to supply the circuit had to be used.

We also found a step-up converter to boost 1.2 volt to 5 volt (see below). When we build the

¹<http://ww1.microchip.com/downloads/en/DeviceDoc/22234B.pdf>

circuit on a breadboard (with low ESR inductor and short connections), the circuit worked when there was no load. Once we connected the receiver, the voltage dropped to 3.5 volt what is not enough to supply our circuit. The step-up converter can't deliver enough current for a proper working of the receiver circuit.

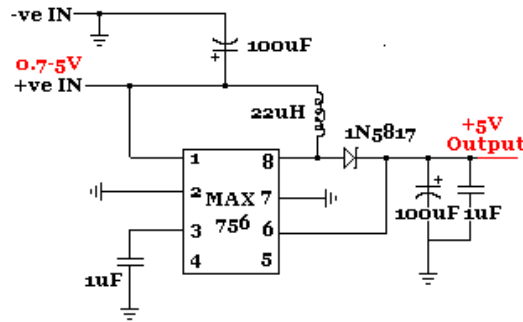


Figure 4.3: Step-up converter (1.2V to 5V)

Then we tried to feed our circuit with the 5 volt from the USB port. This power supply is inherently unstable and depends on the load of the computer. Normally the supply can vary between 4.75 and 5.25 volt. But it can also be possible that the voltage drops to 4.4 volt or even lower. When we tried this to power our circuit, it worked. This 5 volt is used to supply the op-amps. This supply is not so critical, so our circuit will also work properly even if we don't have a stable supply voltage.

4.3 Current-voltage converter

To convert the current from the photodiode in a voltage, we have to use a current-voltage converter. A common way to do this, is to use an op-amp for it, as shown in figure 4.4. In an ideal op-amp the voltage on the minus pin is the same as the voltage on the plus pin. So, the photodiode is in short circuit. If we look to the graph of the short circuit current vs. illuminance (see figure 6.5), we see that the current is proportional to the incoming light if the photodiode is in short circuit.

To have an idea about the current through the photodiode and the voltage after the I-V converter, we set up the circuit of figure 4.4 (with $R_1 = 100 \text{ k}\Omega$, an LM324 op-amp and the BPW21R photodiode) on a breadboard and did some measurements. To determine the current, we can measure the voltage across the resistor. First, we looked what the voltage was in the office with infalling sunlight and with a fluorescent lamp at about two metres from the photodiode ($V_{R_1,light}$). Then, we looked what the voltage was when no light was falling on the diode, $V_{R_1,no-light}$. We found:

$$V_{R_1,light} \approx 0.3 \text{ V}$$

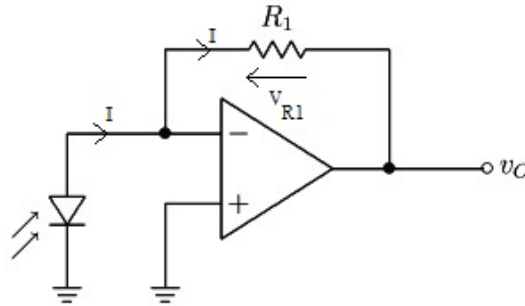


Figure 4.4: Current-voltage converter

$$V_{R1, no-light} \approx 5 \text{ mV}$$

The current through the photodiode is (almost) equal to the current through R_1 :

$$I_{light} = \frac{0.3\text{V}}{100\text{k}\Omega} = 3 \mu\text{A}$$

$$I_{no-light} = \frac{5\text{mV}}{100\text{k}\Omega} = 50 \text{ nA}$$

The output voltage $V_O = -V_{R1}$.

$$V_{R1, light} \approx -0.3 \text{ V}$$

$$V_{R1, no-light} \approx -5 \text{ mV}$$

4.4 Ambient light filter

The photodiode will also pick up ambient light and light coming from the lighting in the building. Those have to be filtered out. The lighting in the building are fed from the net of 230 V_{RMS} at 50 Hz. Due to the rectification, there will be radiation of 100 Hz frequency components that can influence the working of the photodiode. So, we need a high pass filter with a cut-off frequency $f_{\text{cut-off}}$ above 100 Hz to filter the ambient light and the components of 100 Hz. Our first thought was to build a simple RC high pass filter (see figure 4.5) after the current-voltage converter. [2]

The formula for $f_{\text{cut-off}}$ is then as follows:

$$f_{\text{cut-off}} = \frac{1}{2 \cdot \pi \cdot R \cdot C}$$

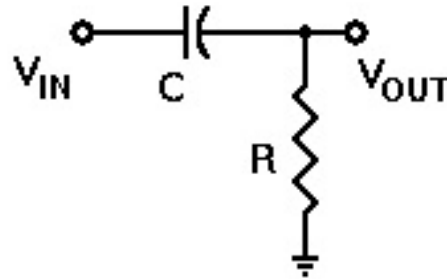


Figure 4.5: RC high pass filter

If we take $f_{\text{cut-off}}=200$ Hz and $R=100 \Omega$, then we find: $C=7,9577 \mu\text{F}$. We take the standard value $C=10 \mu\text{F}$.

The cut-off frequency becomes: $f_{\text{cut-off}}=159,155$ Hz. This value is good to filter the unwanted light.

If we make the circuit, the results are very poor. The bandwidth of the photodiode is large enough for the 100 kHz signal, so the problem must be something else. The characteristics of the LM324 op-amp aren't good enough for a square wave of 100 kHz. Two important characteristics of an op-amp are the bandwidth and the slew-rate. We found for the LM324 op-amp:

- Bandwidth: 1 MHz
- Slew rate: $0.4 \frac{\text{V}}{\mu\text{s}}$

If we use another op-amp (AD817AN), the results are better:

- Bandwidth: 50 MHz
- Slew rate: $350 \frac{\text{V}}{\mu\text{s}}$

In first instance, the op-amp is supplied with ± 15 volt. If we switch the LED on and off at 100 kHz, we get a signal at the output of the receiver as you can see in figure 4.6.

We can see that the DC component is filtered out, but there is oscillation at the transition from low to high, so the circuit is not ideal to filter the ambient light.

Another thought for the filtering was to put a capacitor (10 nF) between the photodiode and the minus pin of the op-amp as you can see in figure 4.7.

Now we get a signal like you can see in figure 4.8.

The signal is better, but we can still see some high frequency components in the signal. The original rectangular shape is not so good visible anymore, the edges are flattened.

We chose to look for another way to filter the ambient light. After some research we

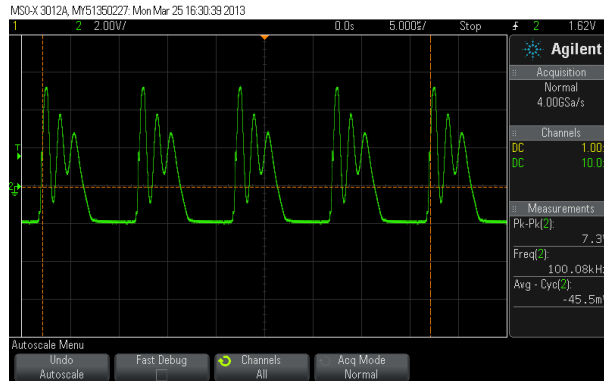


Figure 4.6: Output receiver with RC-filter

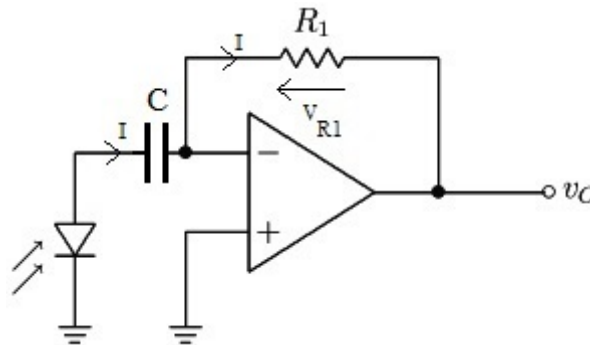


Figure 4.7: Ambient light filter circuit with capacitor

came to the circuit of figure 4.9. We use a feedback circuit that cancels out frequency components below a certain frequency. You can find more explanation about the circuit and the calculation of the transfer function and the cut-off frequency in appendix C.

Until now, we always supplied the op-amps with ± 15 volt. We need three connectors to provide this supply. It should be easier to work with a single supply so we just need two connectors. Since we want to make a portable device, we will use lower voltages. The AD817AN op-amp needs at least a 5 volt single supply, so we will use this voltage to feed the op-amps.

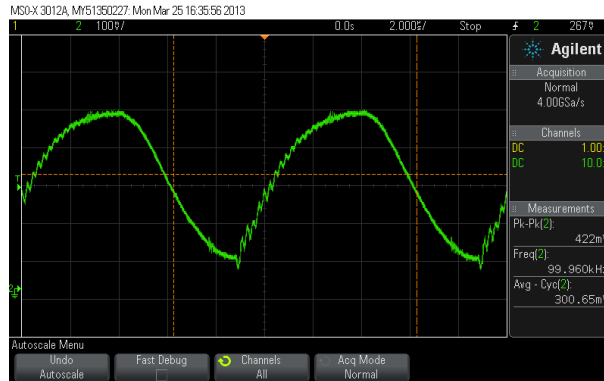


Figure 4.8: Output receiver with condenser

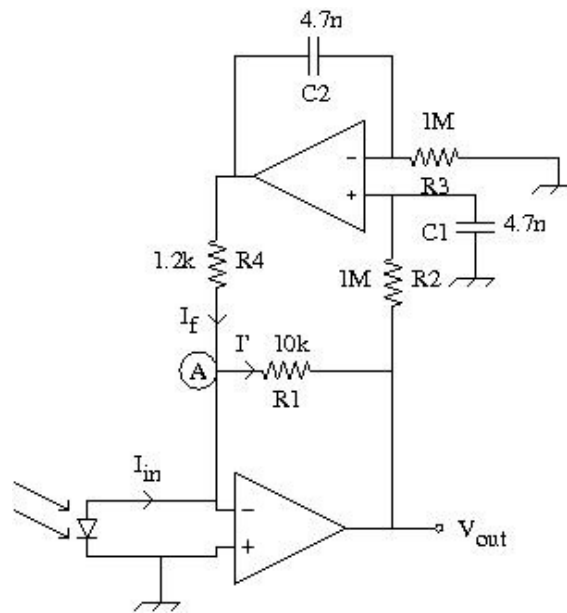


Figure 4.9: Receiver circuit

By using a single positive supply, negative voltages cannot be generated. We will use a virtual ground instead of the normal ground. This special *ground* is usually a voltage reference half way between 0 volt and the supply voltage. In our case, the virtual ground will be 2.5 volt and can be made with a voltage divider and a voltage follower as shown in figure 4.10. [10]

Now we can just use this virtual ground instead of the normal ground. If we want to go

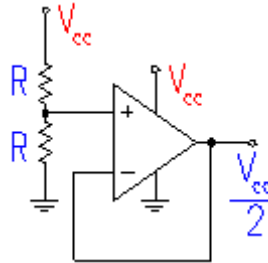


Figure 4.10: Creating a virtual ground for single supply operation

from dual to single supply operation, we have to replace the normal ground by the virtual ground.

4.5 Amplifier

Since the signal is too weak after filtering, we will use an amplifier. The amplification is needed if we want to use the full range of the ADC. After some measurements, we took an inverting op-amp with a gain of 100. Before the amplifier we put a high pass filter with $f_{cut-off} = \frac{1}{2 \cdot \pi \cdot 5100 \cdot 10 \cdot 10^{-9}} = 31.2$ Hz and a gain of 1. We did this because we still had an offset after the ambient light rejection circuit. We also tried to use two op-amp stages with each a gain of 10, but the results weren't so good. We still had an offset and there was more noise in the signal. So, we decided to use a high pass filter with a gain of 1 and an inverting amplifier with a gain of 100.

When we simulated the ambient light rejection circuit, no offset was visible. We just had a DC offset of 2.5 volt because we are working with a virtual ground.

We also tried to build a high pass filter with a gain of 100, but an offset was still visible.

4.6 Anti-aliasing filter

It is recommended to use a low pass filter before the ADC to prevent aliasing. The Nyquist theorem says that the sample rate must be minimum twice the highest frequency component in the analog signal, if we want to reconstruct it properly. As we want to sample at 1 MHz, we will use a filter with a cut-off frequency of maximum 500 kHz.

The easiest and most common way to make a low pass is just to use a simple RC filter. In this case, it's not a good idea. RC filters are cheap, but at higher frequencies LC filters are a better choice. LC filters don't make use of resistors, so they don't dissipate power (in the ideal situation). We want a high order low pass filter with a steep attenuation curve to prevent aliasing. We chose to work with a 7th order filter (4 inductors and 3 capacitors) to have enough attenuation at 1 MHz. A 7th order filter has an attenuation of $7 \cdot 20 \frac{\text{dB}}{\text{dec}} =$

140 $\frac{\text{dB}}{\text{dec}}$. To calculate the values of the components, we used the program *RFSim99* at the recommendation of Mr Rieder Médard. In this program we can calculate and simulate the filter we need. We have to give the filter type (Butterworth or Chebyshev), the cut-off frequency, the order of the filter and the input and output impedance. We chose for a 7th order Butterworth filter with a cut-off frequency of 500 kHz and an input and output impedance of 50 Ω . In the beginning, we chose this impedance because the spectrum can be easily measured with a network analyzer (optimized for input and output impedance of 50 Ω). Later on, we replaced the input and output impedance with these of our circuit and looked how the spectrum of the filter changed. Once we have the calculated values, we took the nearest standard values. Finally, we had a result as you can see in figure 4.11.

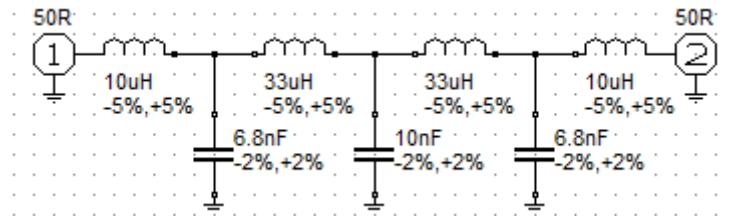


Figure 4.11: LC filter with $f_{cut-off} = 500$ kHz and $Z_{in} = Z_{out} = 50 \Omega$

With the help of Mr Rieder Médard, we built the filter (figure 4.12) and measured the frequency spectrum of the filter with a network analyzer. The result can you see in figure 4.13. We can see that the filter is ideal to prevent anti-aliasing. We have a cut-off frequency (3 dB attenuation) of 474 kHz and at 1 MHz (sample rate), we have an attenuation of 36 dB.

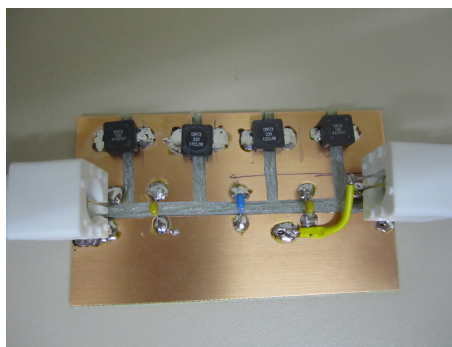


Figure 4.12: LC filter with $f_{cut-off} = 500$ kHz and $Z_{in} = Z_{out} = 50 \Omega$ (PCB)

The problem is that the output and input impedance of the filter is supposed to be 50 Ω . In our circuit, this is not the case. The question is how the filter will react if we deviate from

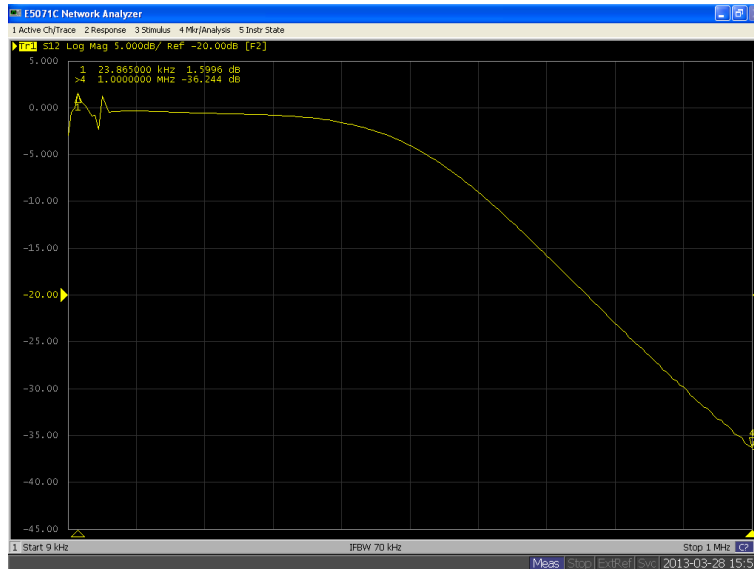


Figure 4.13: Spectrum LC filter measured with network analyzer (50 Ω input and output impedance)

these impedances. The input impedance in our circuit is low ($\approx 1 \Omega$), because the filter comes after an op-amp stage. The output impedance is determined by the voltage divider to buffer the ADC and is about 300 Ω . We simulated the filter with LT Spice with those impedances and had a look at the frequency spectrum (figure 4.14). We can see that there are some peaks above 0 dB in the spectrum which can cause oscillation. Nevertheless, we built the filter on a PCB to see what it gives. To build a good filter, it is also important to use inductors and capacitors with a low ESR so the losses are as low as possible.

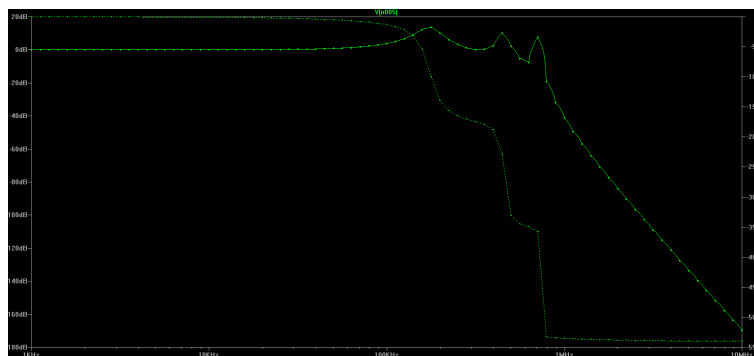


Figure 4.14: Simulation spectrum LC filter with $Z_{in} = 1 \Omega$ and $Z_{out} = 300 \Omega$

We made the whole circuitry on a PCB (from the photodiode to the ADC) and had a look at

the signal before and after the filter (after the voltage divider). First, we switched the light on and off on the rhythm of a 100 kHz square wave at a distance between the LED and the receiver of 2,3 m. The results are shown in the picture below. The green curve represents the signal before the filter and the yellow curve after the filter. As you can see, the high frequency components and the noise are filtered out. The square wave has become almost a sine wave.

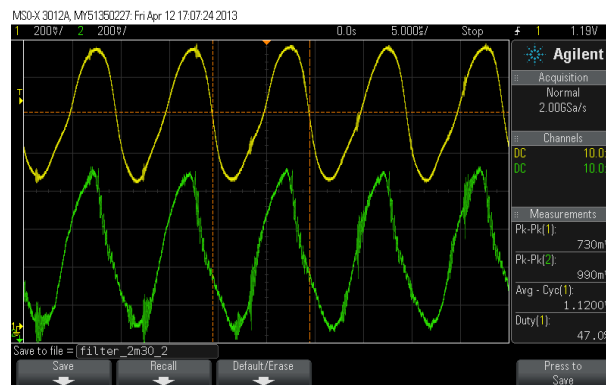


Figure 4.15: Signal before and after filter at 2,3 m (square wave)

If we want to see for oscillation, we drive the LED with a 3-bit LFSR as discussed in chapter 3. Now we can see some oscillation in the signal. If the signal goes from low to high or vice versa, an overshoot occurs. As this is not very desirable, we tried to rebuild the filter.

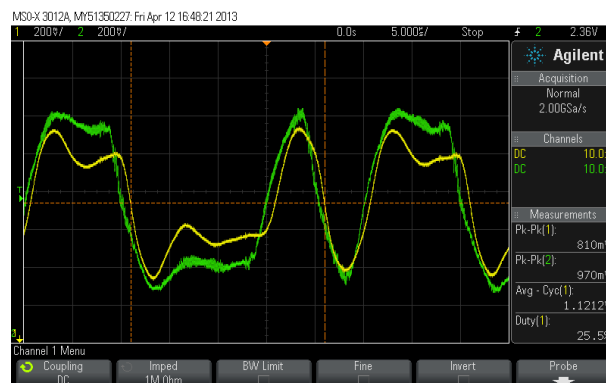


Figure 4.16: Signal before and after filter at 2,3 m (LFSR)

Again, the *RFSim99* program was used. Now we tried to build a filter with $Z_{in} = Z_{out} = 300 \Omega$ and afterwards we adapted the input and output impedance to respectively 1Ω and 300Ω and simulated it in LT Spice. Now we have a spectrum as shown below. The characteristics

of the filter are much better. There is almost no oscillation anymore. The filter can be found in the schematic of the receiver in appendix H.2.

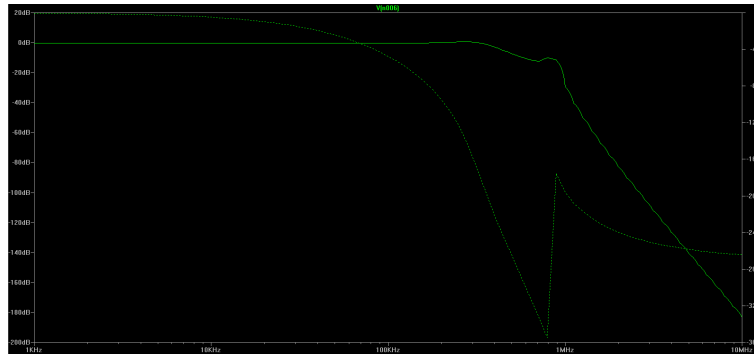


Figure 4.17: Simulation spectrum LC filter with $Z_{in} = 1 \Omega$ and $Z_{out} = 300 \Omega$ (redesign)

We changed the filter on the PCB and looked at the signal when we drive the LED with an LFSR. We can see that the high frequency components are filtered and there is almost no oscillation. We will use this filter in our circuit.

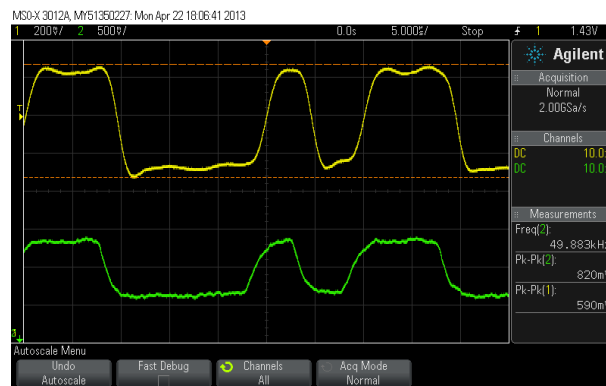


Figure 4.18: Signal before and after filter at 2,3 m (LFSR) with new LC filter

4.7 ADC

If we want to send the information to a computer, we need to digitalize the data. We chose to work with a 12-bit resolution ADC. With 12 bits, we can get $2^{12} = 4096$ possible digital values at the output of the ADC. If we supply it with 3.3 volt, the input range goes then from 0 volt to 3.3 volt. If we supply it with 5 volt, the input range goes then from 0 volt to 5

volt. As the FPGA is also supplied with 3.3 volt, the ADC also has to be supplied with this voltage, because we are sending data from the ADC to the FPGA. The FPGA doesn't accept signals with an amplitude of 5 volt. So, we have a resolution of $\frac{3.3V}{2^{12}-1} = 805.86 \mu V$. We need a high resolution if we want to accurately measure the amplitude of the input signal.

The ADC consists of a successive approximation register with internal track-and-hold. The conversion rate is determined from the serial clock (SCLK) and can be up to 1 MSPS. The timing diagram for the ADC can you see in the figure below.

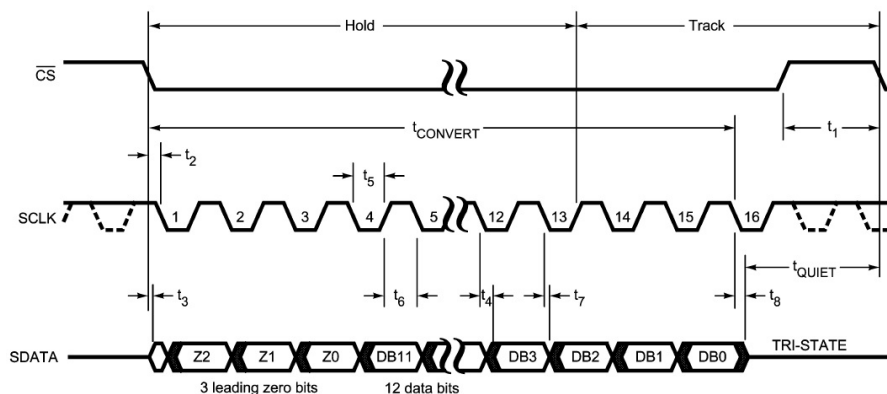


Figure 4.19: Timing diagram 12-bit ADC

A conversion process begins on the falling edge of \overline{CS} . This process takes 16 periods of SCLK. On the first three falling edges of SCLK, a zero is transmitted. After this, the 12 data bits are clocked out on the rhythm of SCLK, beginning with the MSB of the digital value. When the process is done, \overline{CS} is brought back high and the output returns in tri-state. \overline{CS} must stay high during t_{QUIET} before a new conversion process can begin.

We drove the ADC by programming an FPGA so the signals \overline{CS} and SCLK are properly sent like shown in the timing diagram. We chose to work with $f_{SCLK} = 22 \text{ MHz}$ and $t_{QUIET} = \frac{6}{22\text{MHz}} = 272.727 \text{ ns}$. For one sample we need 16 periods of SCLK for the conversion and 6 periods of SCLK when \overline{CS} is high. In this case, we have a sample rate of $\frac{22}{16+6} = 1 \text{ MSPS}$. The datasheet mentions that f_{SCLK} should be maximum 20 MHz. But thus we tried it with 22 MHz and it also worked. If we drive the ADC, we have a serial output as shown below. The upper signal is the \overline{CS} signal, the signal below is the serial output. In appendix I, you can find the code we used to program the FPGA. The simulation of the ADC can also be found in annex D.1.

We used an FPGA board (see figure below) available at school with a Spartan XC3S500E FPGA from Xilinx on it to generate the signals \overline{CS} and SCLK.

The board has to be supplied with a single 5 volt. On the right side, there are two 26-pin connectors so we can connect the FPGA with another circuit. We used this connector to exchange data between the receiver and the FPGA. There are also some possibilities to

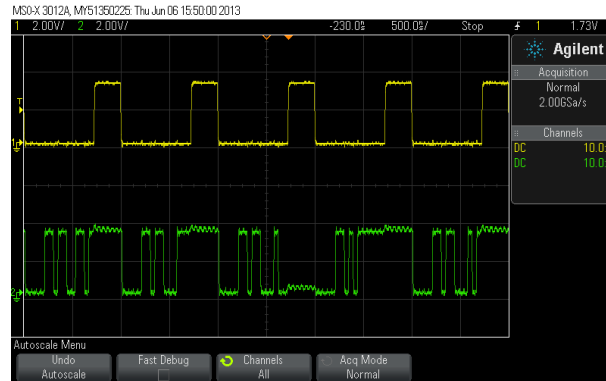


Figure 4.20: Signals \overline{CS} and SDATA

connect the board to a PC. There is an RS232 port, an Ethernet port and a USB port. The schematic of the board and the UCF file can be found at the Wikipedia website of HES-SO Valais/Wallis². We will also use this board for the connection to the PC. This will be explained in the next chapter.

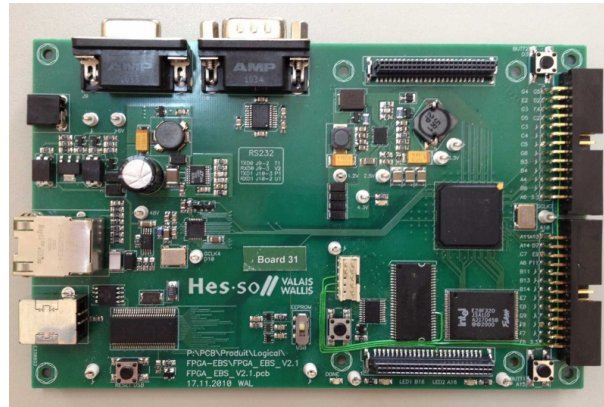


Figure 4.21: FPGA board

Every ADC contributes to the increase of noise at the input, the so-called sampling noise. The datasheet mentioned that the ADC will deliver best performance when driven by a low-impedance source. The voltage divider before the ADC (and after the filter) has to be done with low resistances. We chose the values $51\ \Omega$ and $240\ \Omega$ so that we have the full range ($0 \rightarrow 3.3$ volt) at the input of the ADC at small distances between the LED and the receiver (less than 50 centimeters). The input of the ADC sees the $240\ \Omega$ of the voltage

²<http://wiki.hevs.ch/uit/index.php5/Hardware/FPGAEBES>

divider (see the schematic of the receiver in appendix H.2) in parallel with some other resistances. But the $240\ \Omega$ will determine the source impedance of the ADC. If we look to the graph *THD vs. Source impedance* (see figure 4.22), $240\ \Omega$ should be low enough to prevent distortion.

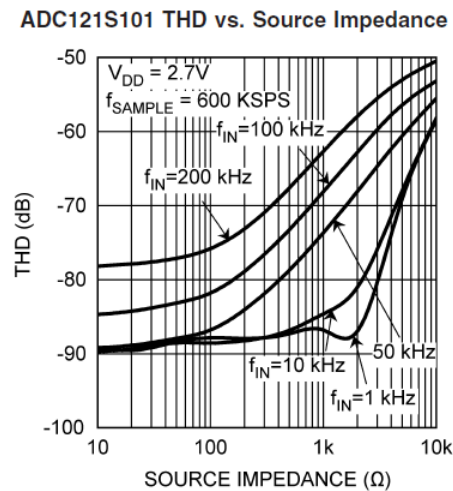


Figure 4.22: THD (dB) vs. Source impedance (Ω)

The datasheet also says that the sampling will cause input current pulses that result in voltage spikes at the input. Every time that the ADC goes from track mode to hold mode or vice versa (see figure 4.19), a spike will occur. When we drive the ADC, we see the signal below at the input.

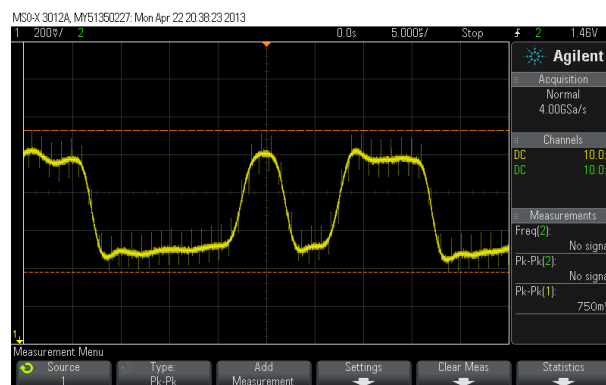


Figure 4.23: Input signal ADC when driven by FPGA (LFSR at 2,3m)

We can clearly see some additional noise and spikes at the charging and discharging moment

of the sampling capacitor. There is a negative spike when going from hold to track mode and a positive spike when going from track to hold mode (see timing diagram). The spikes have an amplitude between 100 and 275 mV and they last for a time between 10 and 50 ns. When making the measurement of the signal with the probe, the loop we make with the ground wire should be as small as possible. Otherwise, the signal will look like below (see figure 4.24). So, we removed the probe tip and the ground wire and measured the signal directly, without making use of the ground wire.

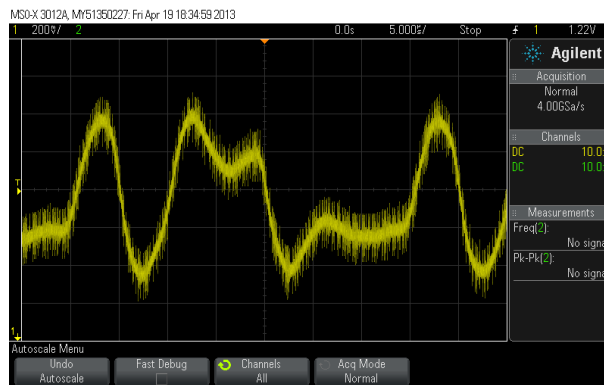


Figure 4.24: Input signal ADC when driven by FPGA measured with probe with ground wire.

As the signal seen on the oscilloscope is maybe not the real signal (because of the capacitance of the probe), we want to send the data to a computer and plot it in a software program such as Matlab, Octave or Python. Another thing we are interested in, is the influence of the spikes for the sampling value.

The final schematic of the receiver (from the photodiode to the ADC) is added in appendix H.2. The circuit on PCB is shown in the picture below.

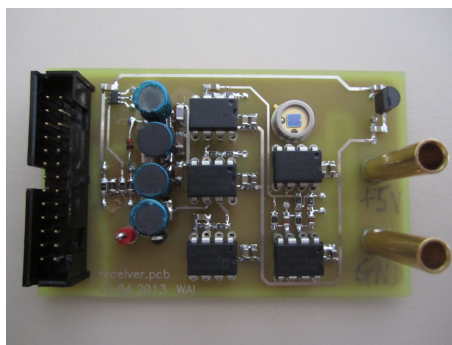


Figure 4.25: PCB receiver

Chapter 5

Connection to the PC

5.1 Introduction

As we have now the data from the ADC, we want to visualize it on the computer to see how the sampled signal looks like. To make the connection, we have several possibilities: RS232, USB, Ethernet, ... Since we are clocking out the data at a frequency of 22 MHz, the link we will using should be fast enough.

RS232 is intended for data rates up to 20 kbps. So, this is too slow for our application.

USB 2.0 can be used for speeds up to 480 Mbps.

Ethernet can reach speeds up to 100 Mbps.

Since the driving of the Ethernet port is already available in VHDL at HES-SO Valais/Wallis, we chose to work with Ethernet for the FPGA-PC communication. If we want to send data on an Ethernet link, we have to encapsulate it with a header. This packet is called an Ethernet frame. In the header, you can find the MAC address of the sender and the receiver, the IP address of sender and receiver, a CRC,... The data should be at least 46 bytes and maximum 1500 bytes.

5.2 Structure Ethernet communication

The structure to send and receive Ethernet packets with the samples of the ADC included, can be seen in figure 5.1. The block on the right represents the ADC. We get an analog input value, this is converted into serial data (*SDATA*) with the signals *SCLK* and *CS_n*. In the *UDPApplication* block, we find the ADC controller that converts the serial data into parallel data. The parallel data is stored in RAM and put in a frame at certain moments (explanation see further). In this block, we decide the destination UDP port, IP address and MAC address. If we want to send a frame, the data and some information for the header (UDP port, IP address, MAC address) is given to the *UDPFifo* block. The data is processed via a FIFO structure. This block gives the data to the *MIItoRAM* block that is an interface between the *UDPFifo* block and the Ethernet controller on the FPGA board (or

the Ethernet controller on the computer). The two blocks on the left are just used for testing purposes. They represent the MII (Media Independent Interface) sender and receiver on the computer side to send and receive Ethernet frames.

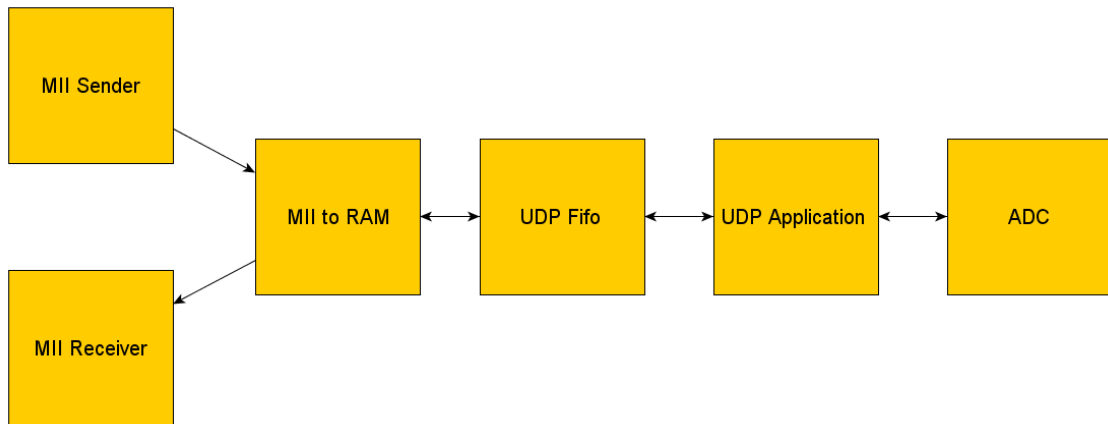


Figure 5.1: Structure to send and receive packets over Ethernet

We don't have to change the *UDPFifo* block and the *MIItoRAM* block. In our case, we just have to adapt the *UDPApplication* block that determines which data we put in RAM and when we want to send a new frame.

To write the samples from the ADC in the Dual Port RAM and to send them in a frame, we used the structure as shown in figure 5.2.

The ADC controller drives the ADC and gets the serial data that represents the taken samples. To put the samples in RAM, we used an interface between the ADC controller and the RAM: *ADCtoRAM*. This block writes the samples in RAM. When CS_n becomes high, a new sample is available. To put the 12-bit samples easily in RAM, we added four zeros before each sample so we have 16 bits (two bytes). Because the RAM is byte addressable, it's easier to work with 16 bits than with 12 bits. The signal *writeEnA* is high during two clock periods. During the first clock period, the eight MSB's of the sample are stored in RAM. During the second clock period, the eight LSB's are stored in RAM. The signal *addressA* is incremented once we have stored a new byte. If half of the RAM is stored with new samples, the signal *sendFrame* is brought to 01 for one clock period. If *sendFrame* = 01, the first half of the RAM is read and sent via *txData*. While reading from the RAM, we can keep on writing in the other half of the RAM. If *sendFrame* = 10, the second half of the RAM is transmitted. Meanwhile we can keep on writing in the first half of the RAM.

The code we used for the different blocks can be found in appendix I.1.2. Once we had written the code, we could program the FPGA and make the connection to the PC with an Ethernet cable. There are two types of Ethernet cables: a straight through Ethernet cable and a crossover Ethernet cable. Normally we should need a crossover Ethernet cable to

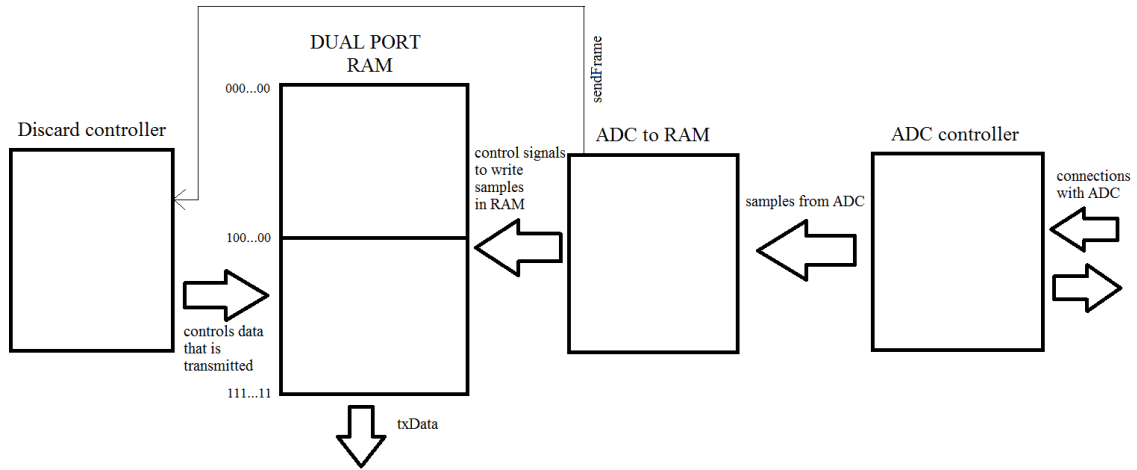


Figure 5.2: Structure to write the samples in RAM and read from RAM

make the connection between the FPGA and the PC. But the Ethernet controller on the FPGA board can automatically detect which cable we are using. So no distinction should be made. We chose to work with a straight through cable because they are more common. If there is data transport over the cable, we can see two status LEDs blink. The left LED is constantly on and is a speed indicator. If this LED is on, it means that we have a 100 Mbps connection. The other LED blinks if there is some activity on the cable. We used a CAT 5e Ethernet cable which is fast enough for the 100 Mbps data transport.

Now we have connection with the PC, we can see the received packets in Wireshark as shown in figure 5.3. We can see the IP address given by the FPGA (169.254.111.1). The destination IP address is put to 0.0.0.0, because the FPGA doesn't know our IP address yet. We first have to send a frame to the FPGA, so he can store our IP and MAC address. We put 1024 data bytes (512 samples) in one packet and every second we received almost 2000 packets. Now we can see the packets in Wireshark, we can use software to process the data.

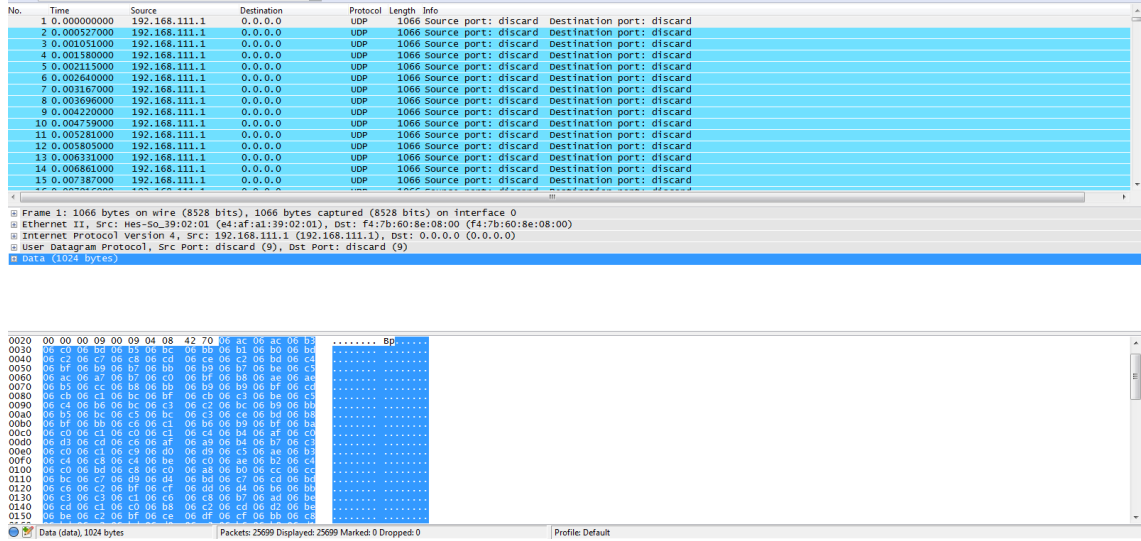


Figure 5.3: Received UDP packets in Wireshark

The construction of the receiver and the Ethernet connection to the PC is shown in figure 5.4. The receiver circuit is connected with the FPGA board. The FPGA drives the ADC on the PCB and sends UDP packets to the Ethernet connector. When a connection with the PC is established, we can see the LEDs blinking on the connector. The supply for the receiver circuit and the FPGA board is done with a USB cable.

How we send a packet from the PC to the FPGA, is explained in the next chapter.

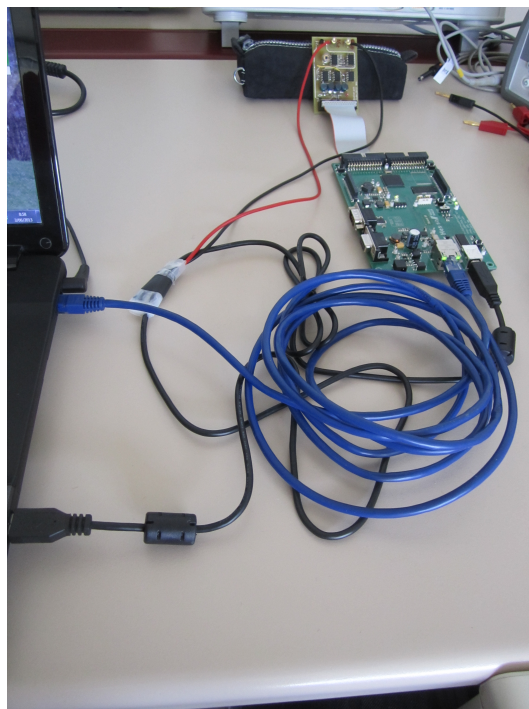


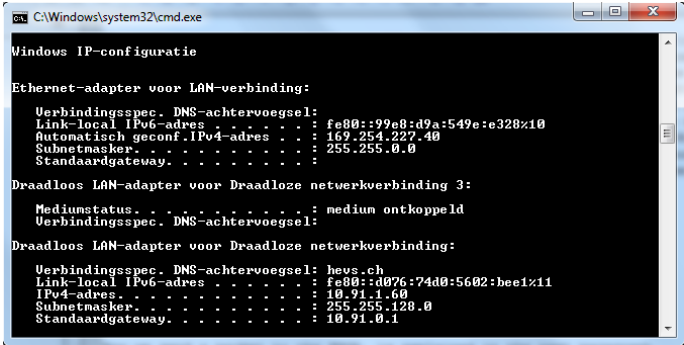
Figure 5.4: Construction of receiver and connection to the PC

Chapter 6

Processing with Python

6.1 Receiving the data

We chose to work with Python to process the data from the ADC. We use a socket so we can send and receive UDP packets. There is a library *socket* that we can import in the program. Before receiving the samples, we first have to send a UDP packet to the FPGA so he can store the IP address and MAC address of the PC. As we have several interfaces (network adapters) on the computer, we first have to bind the socket to the interface we want. In the *Command* window, we can see the several interfaces and their IP addresses (see figure 6.1) via the command `arp -a`. As we want to use the Ethernet adapter, we bind to IP address 169.254.227.40. Once we bind to the socket, we can send data to the FPGA and we can choose a UDP port. The Python code we used to send a UDP packet to the FPGA, can be found in appendix I.2.1. Later on, we put this code in a function and we call it in the `main` program. The structure of our code is explained in the next paragraph.



```
C:\Windows\system32\cmd.exe
Windows IP-configuratie

Ethernet-adapter voor LAN-verbinding:
Verbindingsspec. DNS-achtervoegsel:
Link-local IPv6-adres . . . . . : fe80::99e8:d9a:549e:c328%10
Automatisch geconf. IPv4-adres . . : 169.254.227.40
Subnetmasker. . . . . : 255.255.0.0
Standaardgateway. . . . . :

Draadloos LAN-adapter voor Draadloze netwerkverbinding 3:
Mediumstatus. . . . . : medium ontkoppeld
Verbindingsspec. DNS-achtervoegsel:

Draadloos LAN-adapter voor Draadloze netwerkverbinding:
Verbindingsspec. DNS-achtervoegsel: hevs.ch
Link-local IPv6-adres . . . . . : fe80::d076:74d0:5602:bee1%11
IPv4-adres . . . . . : 10.91.1.60
Subnetmasker. . . . . : 255.255.128.0
Standaardgateway. . . . . : 10.91.0.1
```

Figure 6.1: Network adapters on PC

Firstly, we didn't see the packet in Wireshark so no packet was sent to the FPGA. When we looked in the ARP table of the Ethernet interface, the IP address and MAC address of

the FPGA couldn't be seen. So, we added manually the IP address and MAC address of the FPGA to the ARP table via the command: `arp -s 169.254.111.1 e4-af-a1-39-02-01`. This operation needs administrator rights. If we now look to the ARP table (see figure 6.2), we can see that the addresses are added.

```

C:\Windows\system32\cmd.exe
Microsoft Windows [versie 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Alle rechten voorbehouden.

C:\Users\Jelle>arp -a

Interface: 169.254.227.40 --- 0xa
Internetadres    Fysiek adres    Type
169.254.111.1    e4-af-a1-39-02-01    statisch
169.254.255.255    ff-ff-ff-ff-ff-ff    statisch
224.0.0.22        01-00-5e-00-00-16    statisch
224.0.0.251       01-00-5e-00-00-fb    statisch
224.0.0.252       01-00-5e-00-00-fc    statisch
239.255.255.177   01-00-5e-7f-ff-b1    statisch
239.255.255.250   01-00-5e-7f-ff-fa    statisch
255.255.255.255   ff-ff-ff-ff-ff-ff    statisch

Interface: 10.91.1.174 --- 0xb
Internetadres    Fysiek adres    Type
10.91.0.1         60-73-5c-4e-96-bf    dynamisch
10.91.1.90        00-1e-4c-5d-d2-73    dynamisch
10.91.1.210       4e-8d-79-e7-36-e8    dynamisch
10.91.127.255     ff-ff-ff-ff-ff-ff    statisch
224.0.0.22        01-00-5e-00-00-16    statisch
224.0.0.251       01-00-5e-00-00-fb    statisch
224.0.0.252       01-00-5e-00-00-fc    statisch
224.0.0.253       01-00-5e-00-00-fd    statisch
224.0.1.60        01-00-5e-00-01-3c    statisch
239.255.255.177   01-00-5e-7f-ff-b1    statisch
239.255.255.250   01-00-5e-7f-ff-fa    statisch
239.255.255.253   01-00-5e-7f-ff-fd    statisch
255.255.255.255   ff-ff-ff-ff-ff-ff    statisch

C:\Users\Jelle>

```

Figure 6.2: ARP table

If you look at figure 6.3 you can see the packet in Wireshark, so the packet is sent to the FPGA. If the FPGA receives the UDP packet properly, the destination IP and MAC address in the FPGA are changed to those of the PC as we can see in Wireshark. When the FPGA receives a UDP packet, a LED on the FPGA board is blinking.

Now we can send a packet to the FPGA, we can start receiving the ADC samples. We bind the socket to UDP port 9 because the frames are send to this port. We can receive packets from all IP addresses who send a packet to port 9. The processing of the received samples is explained in the next paragraph.

The screenshot shows a Wireshark interface with a list of network packets. The selected packet (No. 10105) is a UDP packet from 169.254.111.1 to 169.254.227.40. The detailed view shows the Ethernet II, Internet Protocol Version 4, User Datagram Protocol, and Data (94 bytes) sections. The data section contains a hex dump and ASCII representation of the packet payload.

```

0000 64 af a1 39 02 01 b8 70 f4 7b 60 8e 08 00 45 00  ...9...p {...E.
0010 00 7a 0a bc 00 00 80 11 00 00 a9 fe e3 28 a9 fe  .....f {...(.
0020 6f 01 ce 14 00 09 00 66 a6 9e 48 63 6c 6c 6f 20  0.....f...Hello
0030 77 6f 72 6c 64 20 36 4c 43 20 43 6f 66 6d 73 6e  worl...V...C...Commun
0040 69 63 61 74 69 6f 6e 20 43 6f 72 72 65 6c 61 74  ication Correlat
0050 69 6f 6e 20 66 69 6c 74 65 72 20 50 79 74 68 6f  ion FILT er Pytho
0060 6e 20 43 74 68 65 72 6e 65 74 2c 20 74 65 73 74  n ethern et. test
0070 68 65 73 73 61 67 65 20 74 6f 20 73 65 64      message to send
0080 20 64 6f 20 46 30 47 41                          to FPGA

```

Figure 6.3: Sending a UDP packet to FPGA

6.2 Processing the data

To make the code more readable, we chose to work with a main script where we call the functions we want to implement. The functions are defined in a separate file and can be imported by the main.

The block diagram of the main is shown below.

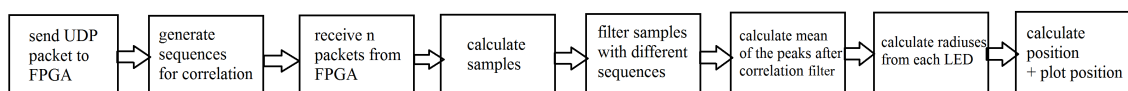


Figure 6.4: Block diagram of the main code

The main script starts with sending a UDP packet so the FPGA knows our IP address and MAC address. Then, we calculate the bit sequences sent by the LEDs. These bit sequences are used to filter the received data.

Because the time between the samples of the ADC is $1 \mu\text{s}$ ($\frac{1}{1\text{MHz}}$) and the time between the bits in the sequences is $5 \mu\text{s}$, we get 5 samples for 1 bit. We repeat each bit in the generated sequence 5 times to match the generated and the received sequences. If we have for instance the bit sequence 0 1 1, we used a function to make 00000 11111 11111.

We can also determine in the main on how many packets we want to do the calculations. In one packet, 1024 bytes (512 samples) are included. As we have 5 samples per bit, we have

about 100 bits in one packet. This is not enough if we send a long bit sequence. In appendix F, we tested several bit sequences: going from a 6-register LFSR sequence ($2^6 - 1 = 63$ bits) till a 15-register LFSR sequence ($2^{15} - 1 = 32767$). The results are discussed in the appendix. For the indoor positioning, we chose to work with a 10-register LFSR sequence ($2^{10} - 1 = 1023$). Now, we can start receiving the data. Every time a new packet arrives, we put the 1024 data bytes in a bytearray. To make it easier to do calculations, we convert the bytearray in a list of integers. Since we want to have the value of the samples (which are two bytes), we calculate the samples and store them in a list of 512 elements. If we have for instance `06FF`, we can calculate the sample via $255 + 6 \cdot 16^2 = 1791$. Now we have calculated the samples, we can also calculate the corresponding analog values as seen on the oscilloscope with the formula : $\frac{\text{value sample}}{\text{maximum value}} \cdot 3.3 \text{ volt} = \frac{\text{value sample}}{4095(FFF)} \cdot 3.3 \text{ volt}$. In our example we find: $\frac{1791}{4095} \cdot 3.3 \text{ volt} = 1.44 \text{ volt}$. If we want to represent the samples in a graph, we have to determine what the time is between two successive samples. This is given by the sample frequency: 1 MHz. So, the time between two samples is $\frac{1}{1\text{MHz}} = 1 \mu\text{s}$. As we could still see some spikes in the graph, we wrote a function to reduce the peaks. In the graph below, you can see the sampled data from one packet (512 samples). The largest peaks are filtered out.

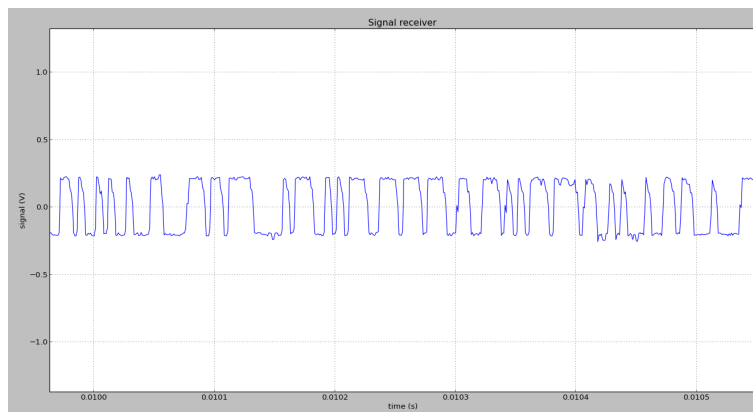


Figure 6.5: Graph received samples (bit sequence of 1023 bits)

Afterwards, we used the function `lfilter` to filter the received signal with each of the bit sequences sent by the LEDs. The function $y = \text{lfilter}(b, a, x)$ requires three arguments. `b` represents the array representation of the nominator coefficients of the filter. `a` represents the array representation of the denominator coefficients. `x` represents the input array that we want to filter. In the Z domain, we can write y in function of x :

$$Y(z) = \frac{b_0 + b_1 \cdot z^{-1} + \dots}{a_0 + a_1 \cdot z^{-1} + \dots} \cdot X(z)$$

In time domain, the different samples of y are been calculated with the formula:

$$a_0 \cdot y_n = b_0 \cdot x_n + b_1 \cdot x_{n-1} + \dots - a_1 \cdot y_{n-1} - \dots$$

If $a = [1]$, we can rewrite the formula:

$$y_n = b_0 \cdot x_n + b_1 \cdot x_{n-1} + \dots$$

If we want to have peaks after the correlation filter, we should flip the bit sequences before doing the filtering. This is explained in following example:

The input signal: $x = [2 \ -2 \ 2 \ 2 \ 2]$

If we want to do the correlation with the bit sequence $b = [1 \ 0 \ 1 \ 1 \ 1]$, we will see whether it is best to do the filtering with this sequence or with the inverted sequence $b_{\text{inv}} = [1 \ 1 \ 1 \ 0 \ 1]$.

Using the above formula, we can calculate the output y in both cases. We find:

$$y = [2 \ -2 \ 4 \ 2 \ 4]$$

$$y_{\text{inv}} = [2 \ 0 \ 2 \ 2 \ 8]$$

If we use the non-inverted bit sequence for the filtering, we can't notice a very clear peak. If we use the inverted sequence for the filtering, we can see a clear peak (with value 8) at the end of the sequence. So if we want to correlate the received signal with a bit sequence, we have to invert the bit sequence.

This correlation filter results in peaks at a certain frequency (according to the length of the bit sequences). If we apply this function for filtering the samples as seen in the graph before, we get a result as shown in figure F.

We also wrote a function for an auto-correlation: we filtered the ideal signal with itself. If we do this for a 10-register LFSR sequence, we get a result as you can see in figure 6.7. There is no noise and the peaks all have the same value. We can only see some noise in the first period because the bit sequence isn't recognized yet.

To determine an eventual time shift in the received signal, we also wrote a function that plots the received signal and the ideal bit sequence under each other. You can only use this if one LED is burning. To synchronize both signals, we searched for the number of most successive ones in each of the signals. Once we know this (and the index where this occurs), we can shift both signals so we can see an eventual time shift. In figure 6.8, you can see this for a 10-register LFSR sequence. If we compare both signals, no time shift is visible.

This function and the function for the auto-correlation can be found in appendix I.2.3. We wrote these functions in a separate file `functions_test.py`.

The amplitude of the peaks depends on the intensity of the bit sequence falling on the

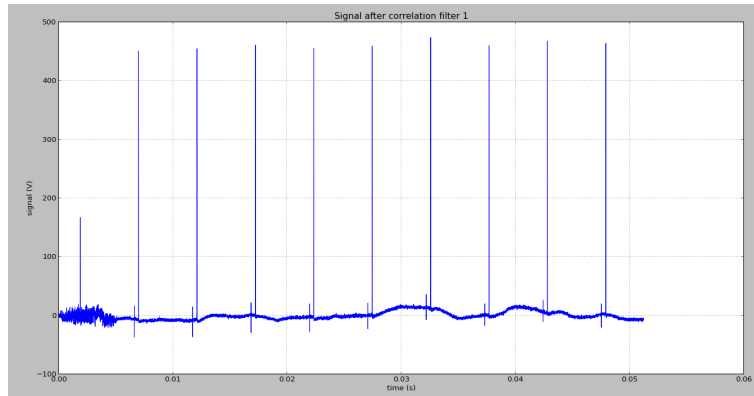


Figure 6.6: Graph received samples after correlation filter (bit sequence of 1023 bits)

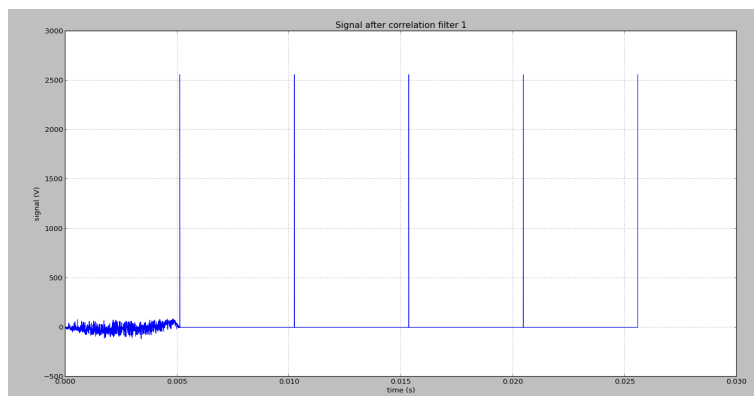


Figure 6.7: Auto-correlation for a 10-bit sequence

receiver. As the peaks don't have all the same amplitude, we wrote a function that returns the mean of all the peaks. We filtered the received signal with each of the bit sequences and calculate the mean of the peaks. These results can then be used to calculate the position as discussed in the next paragraph.

The code of the main script and the file where we put all the functions can be found in appendix I.2.2.

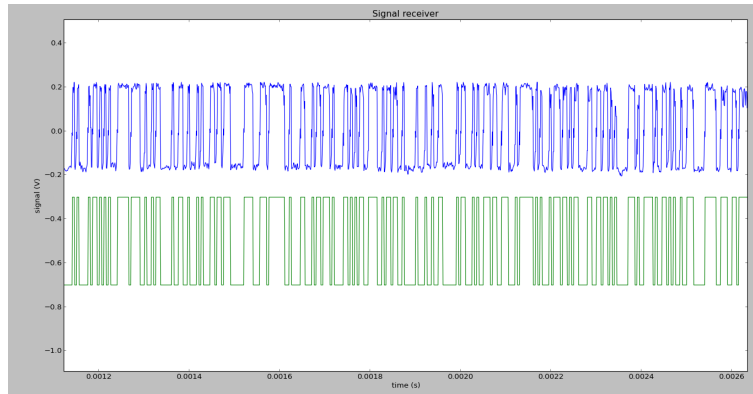


Figure 6.8: Comparison between received signal and ideal signal

6.3 Positioning

6.3.1 Radius calculation

Once we have done the correlation, we have 5 values representing the mean of the peaks after the filtering with each of the 5 bit sequences. Starting from these values, we should calculate the position of the receiver. For the indoor positioning, we assume a constant height (height of the desks), so we just have two dimensions. We did some measurements to determine the link between the correlation values and the radiuses from each LED. If we are under the LED, the radius is zero metres. The larger the radius from the LED, the smaller the correlation value corresponding to the LED. We did this measurement for each of the 5 LEDs when all LEDs are burning. The results are shown in the table. The radius varies from 0 to 3.4 metres. We stopped at 3.4 metres because from that distance the correlation peak values are almost the same as the noise we receive.

On these measurement points, we we fit a function we can use in Python to calculate the radiuses starting from the correlation values. We used `Logger Pro` for the fitting. The best fitting was obtained with a 5th order polynomial. We did this for each LED to find the coefficients of the polynomial. The result from the fitting of the first LED are shown below.

We found as coefficients of a 5th order polynomial ($y = A + B \cdot x + C \cdot x^2 + D \cdot x^3 + E \cdot x^4 + F \cdot x^5$) for the first LED:

$A = 5.668$
 $B = -0.05907$
 $C = 0.0003756$
 $D = -1.253e-6$
 $E = 2.028e-9$
 $F = -1.268e-12$

Table 6.1: Measurements radius versus correlation value with each LED

radius [m]	LED 1	LED 2	LED 3	LED 4	LED 5
	corr. value	corr. value	corr. value	corr. value	corr. value
0	540,3	815,84	586,89	667,78	728,81
0,2	539,85	790,3	568,21	661,3	670,13
0,4	524,82	739,84	560,13	628,54	613,55
0,6	459,4	644,85	508,07	573,38	577,46
0,8	414,37	582,06	439,35	503,93	502,2
1	360,77	488,95	374,84	439,79	434,13
1,2	315,53	416,21	315,06	377,7	362,63
1,4	268,51	360,8	270,51	318,86	315,24
1,6	216,24	285,86	215,9	272,25	258,74
1,8	177,96	232,83	173,2	210,29	208,87
2	142,07	187,07	132,45	172,82	166,79
2,2	118,8	148,8	107,43	149,28	134,57
2,4	104,81	122,17	91,79	126,39	108,83
2,6	90,72	106,16	80,39	100,39	87,21
2,8	81,18	79,3	72,54	84,08	75,63
3	69,56	58,83	61,63	66,59	65,29
3,2	61,15	51,48	48,57	55,24	54,76
3,4	52,98	42,15	42,89	43,25	54,64

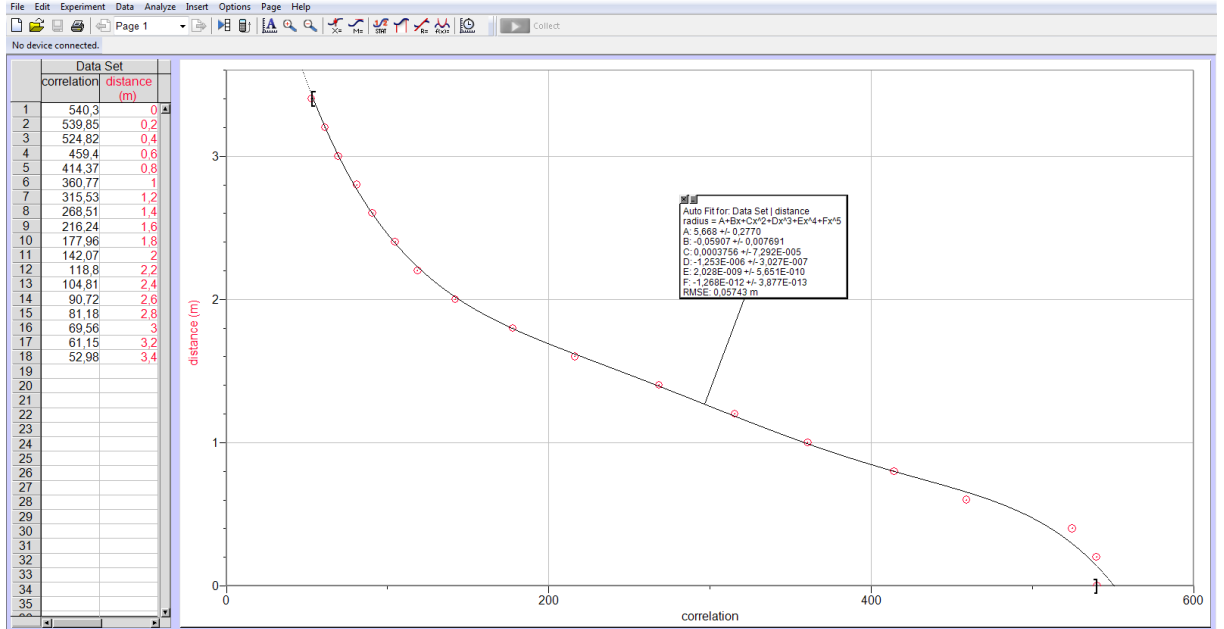


Figure 6.9: Fitting correlation value vs. radius for first LED

In the next paragraph, we explain how we compute the position starting from these radiuses.

6.3.2 Position calculation

After finding the horizontal distance between each led and the receptor. We can compute 2 intersections points with 2 radiuses represented by figure 6.10.

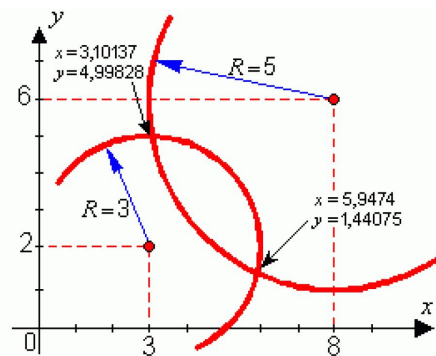


Figure 6.10: intersection points of 2 circles

The first step is to calculate coefficients N, A, B and C to compute coordinates x and y of both intersection points:

$$N = \left(\frac{r_1^2 - r_0^2 - x_1^2 + x_0^2 - y_1^2 + y_0^2}{2(y_0 - y_1)} \right)$$

$$A = 1 + \left(\frac{x_0 - x_1}{y_0 - y_1} \right)^2$$

$$B = 2y_0 \left(\frac{x_0 - x_1}{y_0 - y_1} \right) - 2N \left(\frac{x_0 - x_1}{y_0 - y_1} \right) - 2x_0$$

$$C = x_0^2 + y_0^2 + N^2 - R_0^2 - 2y_0N$$

After calculating these 4 coefficients, the x and y coordinates can be calculated by following formulas:

$$x_{1,2} = \left(\frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \right)$$

$$y_{1,2} = N - x_{1,2} \left(\frac{x_0 - x_1}{y_0 - y_1} \right)$$

If 2 luminous sources have the same y-coordinates, previous formulas are not valid because of division by zero. Therefore if $y_0 = y_1$, we must use below equations:

$$A = 1 \quad ; B = -2y_1 \quad ; C = x_1^2 + x^2 - 2x_1x + y_1^2 - R_1^2$$

$$x = \left(\frac{R_1^2 - R_0^2 - x_1^2 + x_0^2}{2(x_0 - x_1)} \right)$$

$$y_{1,2} = \left(\frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \right)$$

As multiple light sources are used in this project, we must repeat the operation with all the possible combinations. To find the receptors position, we must find the position of the points we are interested in, we assume that the correct points are those who are to closest to each other. Finally, those calculated points are necessarily not at the same place. Therefore, to refine the calculating and to find the receptor position, we must take the geometric average of each point. The written function in Python that computes the receptor position and middle points presented above can be found in appendix I.2.3.

In figure 6.11 you can see the LEDs (blue dots) with the radius around where the receiver can be for 1 led. At every intersection of 2 LED radius circles the calculated intersection points are shown (black and orange triangles). We will only use the orange triangles to calculate the geometric average as these are the closest to each other. Only the intersection points of the LEDs that are closer than 2.5m to the receiver are shown. Because the longer the distance from the LED the less accurate the measurement. We only do the calculations with the most accurate values.

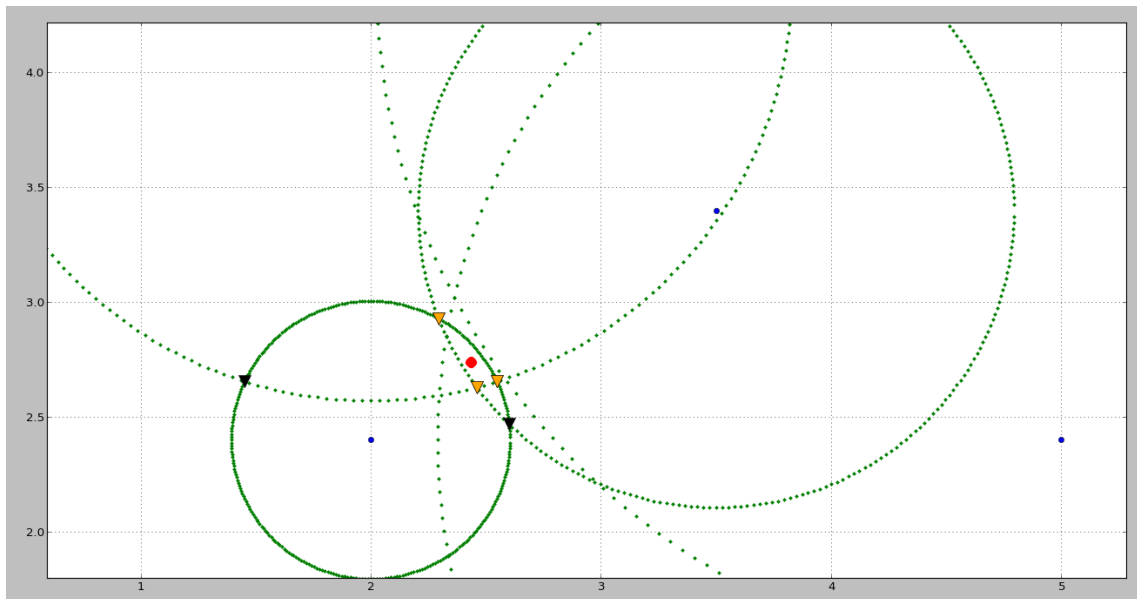


Figure 6.11: Position calculation (graphical view)

6.3.3 Data Transmission

Until now, the bit sequences sent by the LEDs were just used to calculate the position. Moreover, we can also transmit data via the LEDs. If we invert the bit sequence during certain periods, we can transmit zeros and ones. If we use the same correlation filter, we will see positive peaks and negative peaks as well. A positive peak represents a digital zero, a negative peak represents a digital one, or vice versa. We tried this possibility, the results are discussed in appendix G.

Chapter 7

Result

For this master's thesis, we had to design a sender and a receiver for Visible Light Communication so we can determine the position in a room. The sender can be plugged into the mains and the transmitted binary sequences can be controlled by an FPGA.

The receiver is designed to receive at $200 \frac{\text{kbits}}{\text{second}}$. We used an Ethernet connection to send the samples of the received signal taken by the ADC to the PC. To process the data and calculate the position, Python was used.

We can also transmit data besides the indoor positioning by inverting some periods of the sent bit sequence. The final result of the Python code is showed below. You can see the position of the receiver, marked with a red dot. You can see the walls of the room and the position of the LEDs (marked with blue dots). You can also see the coordinates of the calculated position.

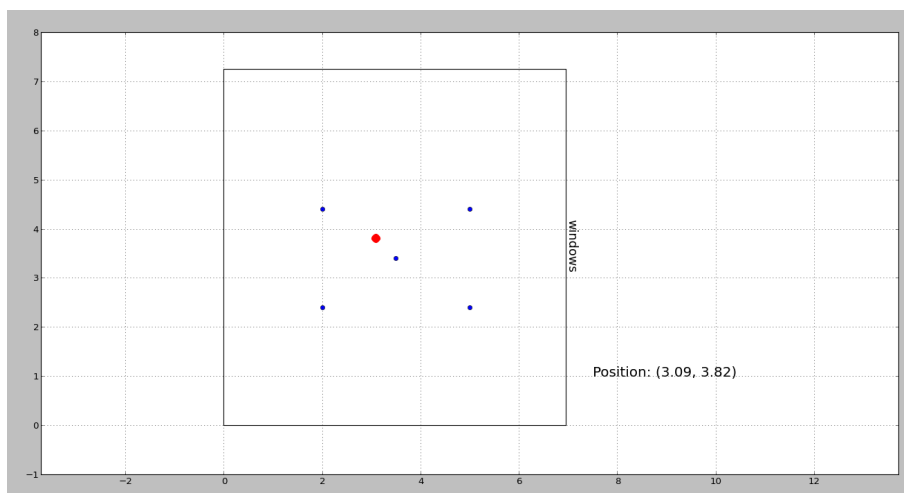


Figure 7.1: Graphical view of the position of the receiver in the room

Chapter 8

Future work

Because of the limited time wherein we had to do our research and developing we couldn't do all possible tests we have in mind relating the subject of this report. Hereby we give a list of future work that can be done additionally to this report.

- More accurate tests can be done to determine the accuracy of the system, and increasing the precision of the positioning.
- Test can be done with different LEDs to look at the different responses of the receiver.
- The use of lenses can be investigated, both on the LEDs or on the photodiode to have a larger working area.
- The LEDs are now controlled by 1 FPGA, test can be done to look if there is any change when using 2 asynchronous FPGAs.
- A smaller circuit can be made to develop a device that can be implemented in a lamp holder.
- Tests of indoor positioning can be done on a greater scale, in a big room, over the whole building.
- Software can be updated to do dynamic positioning.
- Tests and measurements can be done to do 3D positioning.
- Next to the positioning, data(files) can be sent. This can be done by sending the same information over all the LEDs or different information over each LED.
- The frequency can be increased by changing the receiver circuit: using high frequency op-amps with very high slew rates, an anti-aliasing filter with a higher cut-off frequency and a faster ADC (ADC with parallel output).

- The receiver can be made smaller to make a small device that can be easily plugged in a USB or Ethernet port. Therefore the possibility to do Power Over Ethernet (POE) or sending the data over a USB connection have to be investigated.
- The LEDs can be placed more spread in the room, so a more accurate positioning can be done in the corners. Now, the LEDs are placed rather centrally in the room, what means that the signal at the receiver is very low if we are in the corners. We can also use more LEDs for a more accurate positioning.

Bibliography

- [1] Bode/Nyquist Plot Java Applet. <http://www.williamsonic.com/BodeNyquist/index.html>. .
- [2] Bastien Bagnoud. *Indoor positioning Visible Light Communications*. Technical report, HES-SO Valais/Wallis. Bachelor's thesis.
- [3] Bridgelux. *Bridgelux ES Star Array Series*. Product Data Sheet DS23, 101 Por tola Avenue, Livermore, March 2011. Datasheet.
- [4] Bridgelux. *Effective Thermal Management of Bridgelux LED Arrays*. Product Data Sheet DS23, 101 Por tola Avenue, Livermore, January 2011. Datasheet.
- [5] Burr-Brown. *OPT202-Photodiode with on-chip amplifier*. . Datasheet.
- [6] Hamamatsu. *Si PIN Photodiode S1223 series*. . Datasheet.
- [7] Microchip Technology Inc. *Switch Mode Power Supply (SMPS) Topologies (Part I)*. AN10, 2355 West Chandler Blvd. Chandler, Arizona, USA, Oktober 2007. Application Note.
- [8] New Wave Instruments. http://www.newwaveinstruments.com/resources/articles/m_sequence_linear_feedback_shift_register_lfsr.htm. .
- [9] LEDFusion. www.ledfusion.com.au. .
- [10] Single Supply Op-amps. <http://www.swarthmore.edu/NatSci/echeevel/Ref/SingleSupply/SingleSupply.html>. .
- [11] E. Schubert and J. Kim. *Solid-State Light Sources Getting Smart*. volume 308 1274-1278, 2005. Book.
- [12] Vishay Semiconductors. *BPW21R Silicon PN Photodiode*. . Datasheet.
- [13] Burton & Sons Trading. , Lumen House, 3 Darin Court, Crownhill Industrial Estate, Milton Keynes, Buckinghamshire, MK8 0AD, United Kingdom.

Appendix A

List of Main Switch Mode Power Supply (SMPS) Topologies

Boost Converter A basic SMPS topology in which energy is stored in a inductor when a switch is ON, and is transferred to the output when the switch is OFF. It converts an unregulated input voltage to a regulated output voltage higher than the input.

Buck Converter A basic SMPS topology in which a series switch chops the input voltage and applies the pulses to an averaging LC filter, which produces a lower output voltage than the input.

Flyback Converter (FBT) An isolated Buck-Boost SMPS topology in which, during the first period of a switching cycle, the energy is stored in a magnetizing inductance of the transformer. Then, during the second period, this energy is transferred to a secondary winding of the same half-bridge resonant converter where the load is connected in series with the resonant tank capacitor, C, and into the load.

Forward Converter A Buck-derived SMPS topology in which energy is transferred to the secondary of a transformer winding and into the load, when the switching transistor is ON.

Full-Bridge Converter An SMPS topology in which four switches are connected in a bridge configuration to drive the primary of a transformer. This is also known as an H-Bridge Converter.

Half-Bridge Converter A SMPS topology similar to a full-bridge converter in which only two switches are used. The other two are replaced by capacitors.

Half-Bridge LLC Resonant Converter A SMPS half-bridge topology where the series resonant tank consisting of the inductor, L, and the capacitor, C, which is used to generate

another resonant frequency with transformer magnetizing inductance.

Half-Bridge Resonant Converter A half-bridge converter using an LC resonant tank to reduce the switching losses in the MOSFET.H-Bridge Phase-Shift ZVT Converter.

H-Bridge Phase-Shift ZVT Converter A full-bridge converter using the phase-shift ZVS technique is known as an H-Bridge Phase-Shift ZVT topology. In this topology, the parasitic output capacitor of the MOSFETs and the leakage inductance of the switching transformer are used as a resonant tank circuit to achieve zero voltage across the MOSFET at the turn-on transition.S

LLC Resonant Converter A full-bridge converter with an LC resonant tank, which is used to reduce switching losses and the phase shift between the two leg gate pulse defined in the output power flow.

Parallel Resonant Converter (PRC) A half-bridge resonant converter where the load is connected in parallel with the resonant tank capacitor.

Push-Pull Converter An SMPS topology which is using usually a center-tap transformer and two switches that are driven ON and OFF alternately.

Series Resonant Converter (SRC) A half-bridge resonant converter where the load is connected in series with the resonant tank capacitor, C.

Appendix B

Calculations to determine the needed heat sink

The heat flow can be modelled by analogy to an electrical circuit where heat flow is represented by current, temperatures are represented by voltages, heat sources are represented by constant current sources, absolute thermal resistances are represented by resistors and thermal capacitances by capacitors. The diagram shows an equivalent thermal circuit for a semiconductor device with a heat sink. So our LED application can be seen as the circuit in Figure B.1

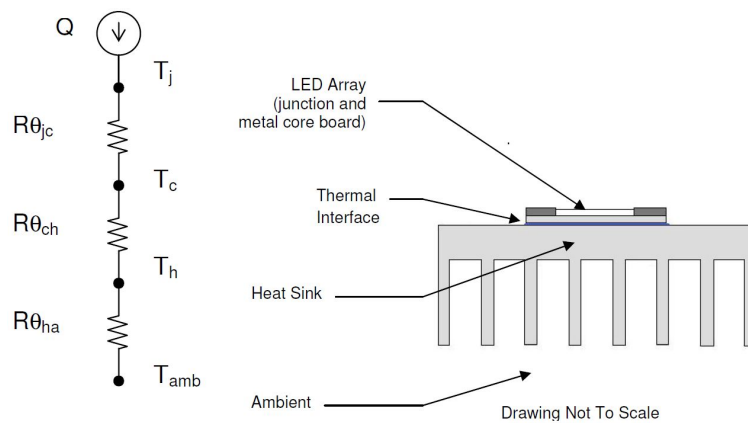


Figure B.1: Thermal circuit[4]

Where

Q is heat flowing from hot to cold, through the LED

T_j is the temperature at the junction of the device

T_c is the temperature at the case of the LED Array

T_h is the temperature at the point where the heat sink is attached to the LED Array

T_{amb} is the ambient air temperature

$R\theta_{jc}$ is the thermal resistance from junction to case of the LED Array

$R\theta_{ch}$ is the thermal resistance between the case of the LED Array and the heat sink

$R\theta_{ha}$ is the thermal resistance of the heat sink

As mentioned in 3.1 we calculate Q :

$$Q = V_{f_{max}} \cdot I_f \cdot 0.85 = 30.9 \cdot 0.35 \cdot 0.85 = 9.19 \quad (\text{B.1})$$

From the datasheets we know:

$$R\theta_{jc} = 1.75^\circ\text{C}/\text{W}$$

$R\theta_{ch} \approx 0.001^\circ\text{C}/\text{W}$ (depends on thickness of the paste layer \rightarrow we take a thickness of 1mm)

$$T_j - T_{amb} = 150^\circ\text{C} - 25^\circ\text{C} = 125^\circ\text{C}$$

$$Q = \frac{T_j - T_{amb}}{R\theta_{jc} + R\theta_{ch} + R\theta_{ha}} \Rightarrow R\theta_{ha} = \frac{T_j - T_{amb} - (R\theta_{jc} + R\theta_{ch}) \cdot Q}{Q} = \frac{125^\circ\text{C} - (1.751 \cdot 9.19)^\circ\text{C}}{9.19} = 11.85^\circ\text{C}/\text{W}. \quad (\text{B.2})$$

We can conclude that the maximum thermal resistance of the heat sink is about $11.85^\circ\text{C}/\text{W}$.

We chose to use a heat sink with a thermal resistance of $3.2^\circ\text{C}/\text{W}$. In equation B.3 & B.4 you can see the calculations for the temperature of the LED junction and the LED casing.

$$Q = \frac{T_j - T_{amb}}{R\theta_{jc} + R\theta_{ch} + R\theta_{ha}} \Rightarrow T_j = (R\theta_{jc} + R\theta_{ch} + R\theta_{ha}) \cdot Q + T_{amb} = 70.5^\circ\text{C} \quad (\text{B.3})$$

$$Q = \frac{T_c - T_{amb}}{R\theta_{ch} + R\theta_{ha}} \Rightarrow T_c = (R\theta_{ch} + R\theta_{ha}) \cdot Q + T_{amb} = 57.01^\circ\text{C} \quad (\text{B.4})$$

Appendix C

Ambient light rejection circuit

The circuit we have used to filter the ambient light, is shown in figure C.1. We found inspiration in the datasheet of the OPT202 photodiode [5]. They mentioned some applications and one of them was a circuit to reject unwanted steady-state background light. We did some adjustments on the circuit and we came to the result below.

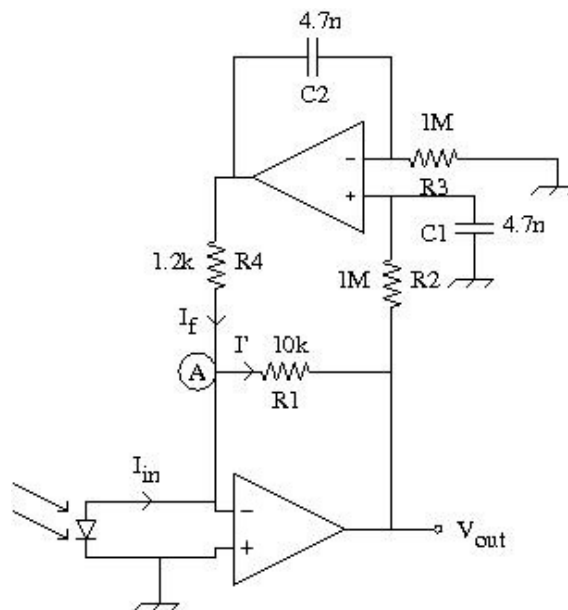


Figure C.1: Ambient light rejection circuit

When light is falling on the photodiode, current flows through it. The current is then converted in a voltage. The feedback circuit cancels out low-frequency background signals. At the output, we will have the AC signal we are interested in.

The circuit can be represented as the general structure of a closed-loop system as shown in

figure C.2.

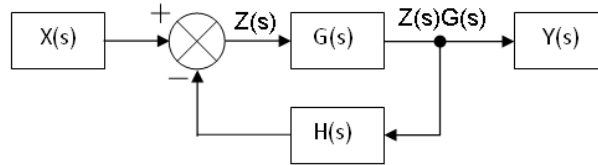


Figure C.2: Closed-loop system

One can easily find the general expression for the transfer function of a closed-loop system:

$$H_{tot}(s) = \frac{Y(s)}{X(s)} = \frac{G(s)}{1+G(s) \cdot H(s)}$$

Now we can identify this structure with the ambient light rejection circuit.

$X(s)$ is the input variable, the current through the photodiode I_{in} . $Y(s)$ is the output variable, the voltage at the output of the op-amp V_{out} . $G(s)$ is the forward transfer function, the transfer function of the current-voltage converter. $H(s)$ is the transfer function of the feedback circuit. $Z(s)$ is the output of the subtraction, the current I' .

The transfer function of the current-voltage converter can be easily calculated as we know that: $V_{out} = -R1 \cdot I'$. So, $G(s) = -R1$.

Now, we have to calculate the transfer function of the feedback circuit as shown in figure C.3.

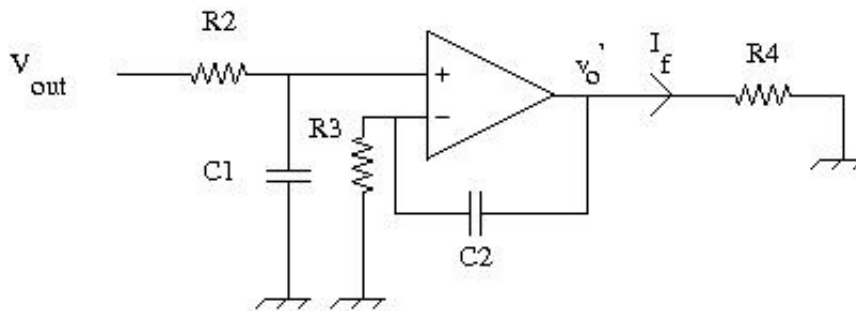


Figure C.3: Feedback circuit

Resistor $R4$ is put on ground, because the voltage at node A is 0 volt. To calculate the transfer function $H(s)$, we replace the capacitors by their Laplace equivalent $\frac{1}{s \cdot C1}$ and $\frac{1}{s \cdot C2}$. v_o' is the voltage at the output of the op-amp. We can calculate the voltage at the minus pin and the plus pin of the op-amp:

$$v_{min} = \frac{R3}{R3 + \frac{1}{sC2}} \cdot v_o'$$

$$v_{plus} = \frac{\frac{1}{sC1}}{R2 + \frac{1}{sC1}} \cdot v_{out}$$

Since these voltages are equal to each other, we can say that:

$$\frac{R3}{R3 + \frac{1}{sC2}} \cdot v_o' = \frac{\frac{1}{sC1}}{R2 + \frac{1}{sC1}} \cdot v_{out}$$

$$\Rightarrow \frac{v_o'}{v_{out}} = \frac{\frac{1}{sC1}}{R2 + \frac{1}{sC1}} \cdot \frac{R3 + \frac{1}{sC2}}{R3}$$

$$\Leftrightarrow \frac{v_o'}{v_{out}} = \frac{1}{R2 \cdot s \cdot C1 + 1} \cdot \frac{R3 \cdot s \cdot C2 + 1}{R3 \cdot s \cdot C2}$$

$$\Leftrightarrow \frac{v_o'}{v_{out}} = \frac{1}{R2 \cdot s \cdot C1 + 1} \cdot \left(1 + \frac{1}{R3 \cdot s \cdot C2}\right)$$

If we take: $R2 = R3 = R$ and $C1 = C2 = C$, we can rewrite the formula:

$$\Rightarrow \frac{v_o'}{v_{out}} = \frac{1}{R \cdot s \cdot C + 1} \cdot \left(1 + \frac{1}{R \cdot s \cdot C}\right)$$

$$\Leftrightarrow \frac{v_o'}{v_{out}} = \frac{1}{R \cdot s \cdot C + 1} \cdot \frac{R \cdot s \cdot C + 1}{R \cdot s \cdot C}$$

$$\Leftrightarrow \frac{v_o'}{v_{out}} = \frac{1}{R \cdot s \cdot C}$$

If we look to the circuit, we can say that: $I_f = \frac{v_o'}{R4}$.

$$\Rightarrow H(s) = \frac{I_f}{v_{out}} = \frac{1}{s \cdot C \cdot R \cdot R4}$$

This transferfunction represents a non-inverting integrator.

Now we know $G(s)$ and $H(s)$, so we can calculate $H_{tot}(s)$.

$I' = I_f + I_{in}$ (Kirchhoff's current law in node A), so we have to add the input current and the feedback current. In the general structure of a closed-loop system, we will do an addition instead of a subtraction as shown in the figure below.

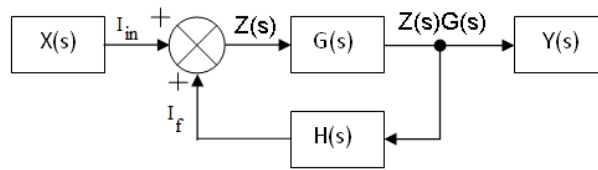


Figure C.4: Closed-loop system with addition

$$\Rightarrow H_{tot}(s) = \frac{G(s)}{1 - G(s) \cdot H(s)}$$

$$\Leftrightarrow H_{tot}(s) = \frac{-R1}{1 + R1 \cdot \frac{1}{s \cdot C \cdot R \cdot R4}}$$

$$\Leftrightarrow H_{tot}(s) = \frac{-R1 \cdot s \cdot C \cdot R \cdot R4}{R1 + s \cdot C \cdot R \cdot R4}$$

This represents the transfer function of a first-order inverting high pass filter with a gain in the pass band of $R2$.

We take the following values for the components:

$$R1 = 10 \text{ k}\Omega$$

$$R2 = R3 = R = 1 \text{ M}\Omega$$

$$C1 = C2 = C = 4.7 \text{ nF}$$

$$R4 = 1.2 \text{ k}\Omega$$

Using a Java Applet [1] we can see the bode plot of the transfer function:

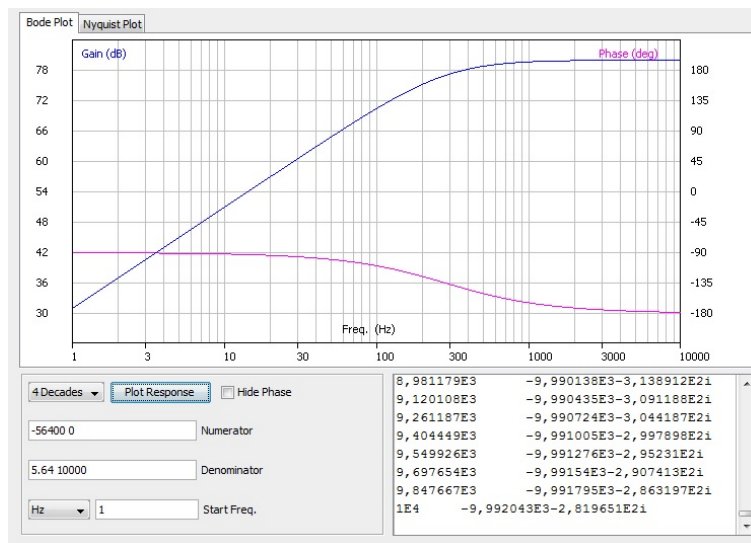


Figure C.5: Bode plot $H_{tot}(s)$

We can see that we have a gain of 80 dB ($20 \cdot \log(10\,000)$) in the passband and that we have a cut-off frequency about 300 Hz.

We can derive a general formula for the cut-off frequency $f_{cut-off}$.

$$\Rightarrow |H_{tot}(j \cdot \omega)|_{\omega=\omega_{cut-off}} = \frac{R1}{\sqrt{2}}$$

$$\Leftrightarrow \frac{R1 \cdot \omega_{cut-off} \cdot C \cdot R \cdot R4}{\sqrt{R1^2 + (\omega_{cut-off} \cdot C \cdot R \cdot R4)^2}} = \frac{R1}{\sqrt{2}}$$

$$\Leftrightarrow \frac{(\omega_{cut-off} \cdot C \cdot R \cdot R4)^2}{R1^2 + (\omega_{cut-off} \cdot C \cdot R \cdot R4)^2} = \frac{1}{2}$$

$$\Leftrightarrow (\omega_{cut-off} \cdot C \cdot R \cdot R4)^2 = \frac{1}{2} \cdot (R1^2 + (\omega_{cut-off} \cdot C \cdot R \cdot R4)^2)$$

$$\Leftrightarrow \frac{1}{2} \cdot (\omega_{cut-off} \cdot C \cdot R \cdot R4)^2 = \frac{1}{2} \cdot R1^2$$

$$\Leftrightarrow \omega_{cut-off}^2 = \frac{R1^2}{(C \cdot R \cdot R4)^2}$$

$$\Rightarrow \omega_{cut-off} = \frac{R1}{C \cdot R \cdot R4}$$

$$\Rightarrow \boxed{f_{cut-off} = \frac{R1}{2 \cdot \pi \cdot C \cdot R \cdot R4}}$$

If we fill in the values for the components, we find: $f_{cut-off} = 282.19$ Hz. This frequency is good enough to filter the DC environmental light and the 100 Hz components from the lighting in the building.

Appendix D

Simulations

In this chapter, we will discuss our simulation results.

D.1 ADC

In the figure below, you can see the test structure of the ADC in HDL Designer. We have an ADC controller and the ADC itself. The tester generates input values for the ADC. The ADC is controlled via the signals CS_n and $SCLK$. The ADC gives a serial output at the rhythm of $SCLK$ for the analog input. The ADC controller converts the serial data to parallel data (signal ADC) and also gives an integer representation of it (signal $voltage$) for the readability.

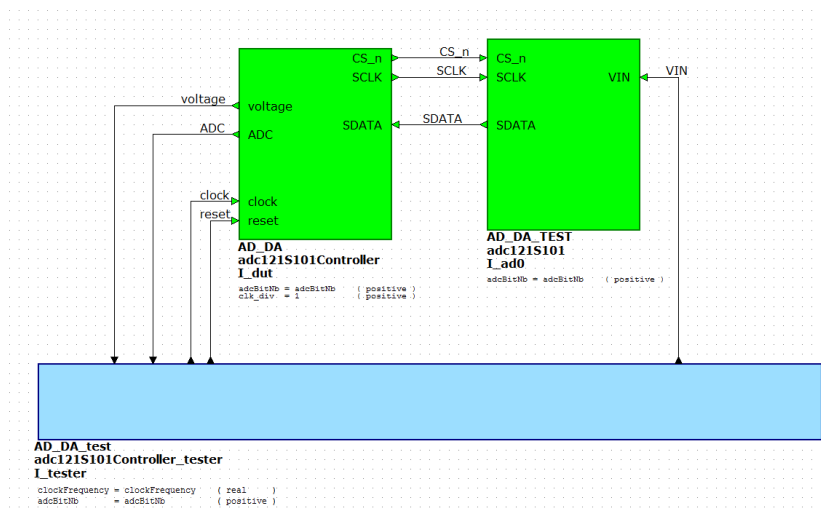


Figure D.1: Testbench ADC

The result of the testing is shown below. First, we can see the analog input vin in integer representation. When CS_n becomes low, a new sample is taken and we can see the serial output (signal $sdata$) at the rythm of $SCLK$. When CS_n becomes high, the parallel data is computed (signal ADC) and the integer representation $voltage$ is shown. We see that this value is the same as the input value, so the ADC controller works.

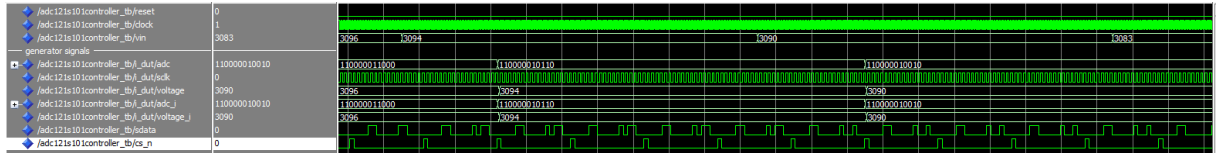


Figure D.2: Simulation ADC

D.2 Sending Ethernet frames

In figure D.3, you can see the test structure for sending and receiving Ethernet frames. The ADC determines which data is put in the Ethernet frames when we want to send one. On the right, we have the ADC who communicates with the *UDPApplication* block. This block transmits the data that is stored in RAM and creates a header around it (MAC address, IP address, UDP port...). The next block is the *UDPFifo* block. The data is processed via a FIFO structure. The *miiToRAM* block is used as the interface to connect the FIFO block with the Ethernet controller on the FPGA board. The two blocks on the left, *miiSender* and *miiReceiver* are used to simulate the behaviour for sending and receiving Ethernet frames on the computer side.

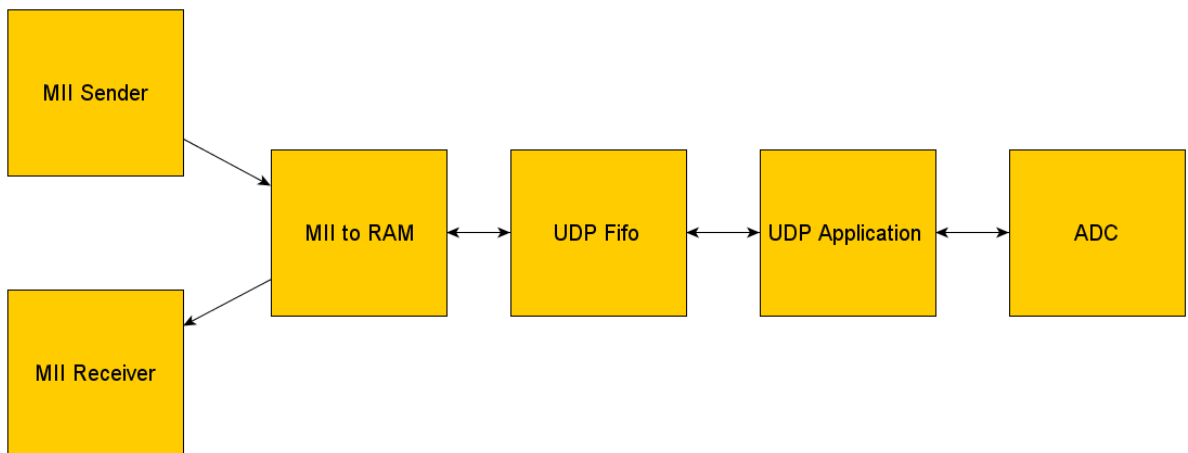


Figure D.3: Testbench Ethernet

In figure D.4, you can see what happens when we send a frame. When we want to send an Ethernet frame, the signal *sendFrame* is brought to 01 or 10 (depending on which part of the RAM we want to send) for one clock period. The content of the RAM is read via signal *addressb* and the data is transmitted to the FIFO. When the FIFO is full, we stop writing for a while until there is place again. We can see that the *MIIReceiver* on the computer side receives the data. When there is a frame, *mii.txen* is brought high so the data can be received.

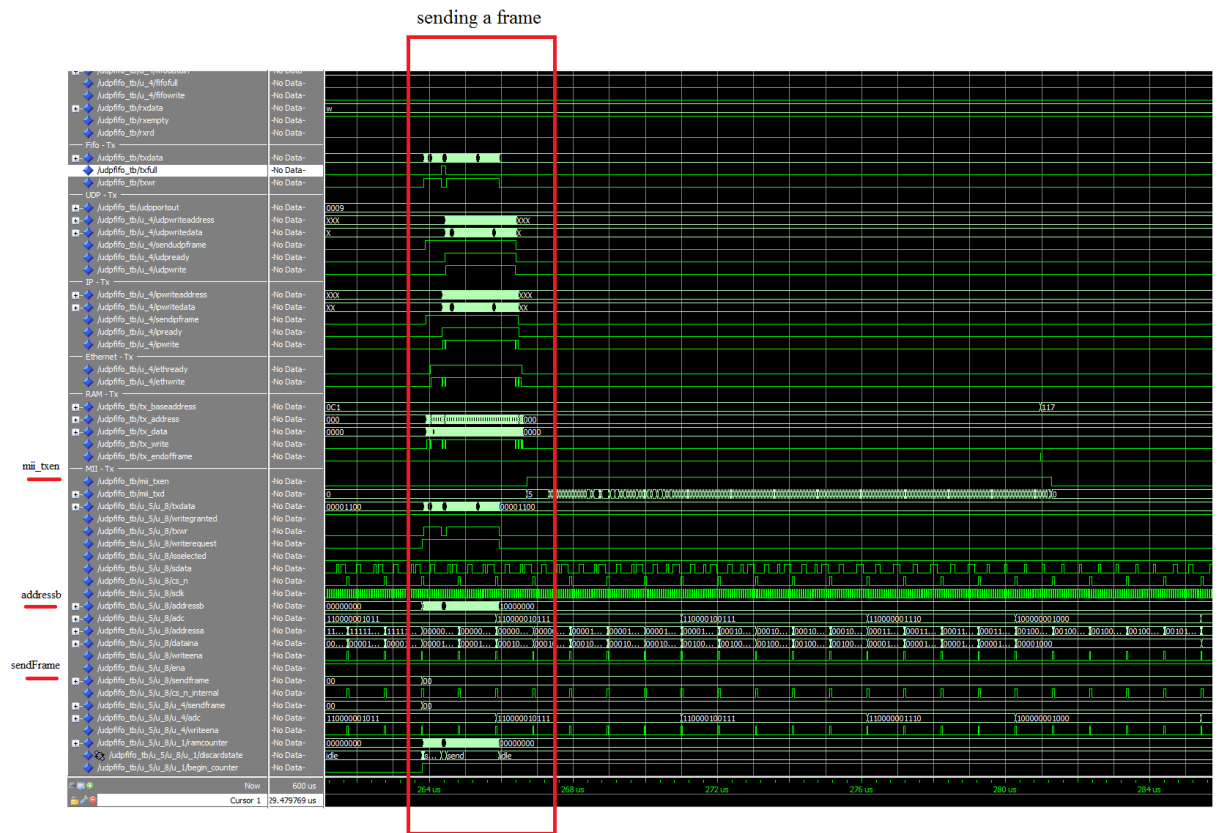


Figure D.4: Simulation sending Ethernet frames

Appendix E

Tests of sending circuit

E.1 Results Test 1 of circuit 1

During this test we look at the functionality of the transformer operating with the power switch, to look what reference voltage we need for the feedback and how the circuit is reacting.

E.2 General findings

In general we can see that the rectified voltage of 320V has a ripple of 5V, what also has influence on the input current (current through the transformer) as shown in figure E.1. Whereas this has not real influence on the function of our transformer we won't try to compensate this behaviour.

If we look closer at the input and output current (figure E.2) we can see that the circuit behaves like expected for a flyback converter. Both currents have some ripple. Whereas this has no big influence on the emitted light, this ripple will be ignored.

E.3 Changing the duty cycle

We can regulate the duty cycle of the power switch by changing the value of a resistance in the feedback loop. This is possible by the implemented potentiometer (P1). If we want to keep the output voltage low, the power switch will stay isolated for multiple periods (Figure E.3). As we are not interested in this behaviour, we look at the behaviour where the power switch is always switching at a frequency of 100 kHz and changing the duty cycle.

During the test shown in figure E.4 we got a V_{AUX} of 16.9 V and a duty cycle less then 20 %. This is way to low where as we want to use a duty cycle of 50% at least to use maximum illuminance for our led. When changing the resistance value in the feedback loop we don't get a lot of change in the duty cycle, only V_{AUX} goes up. This is due to the related Power

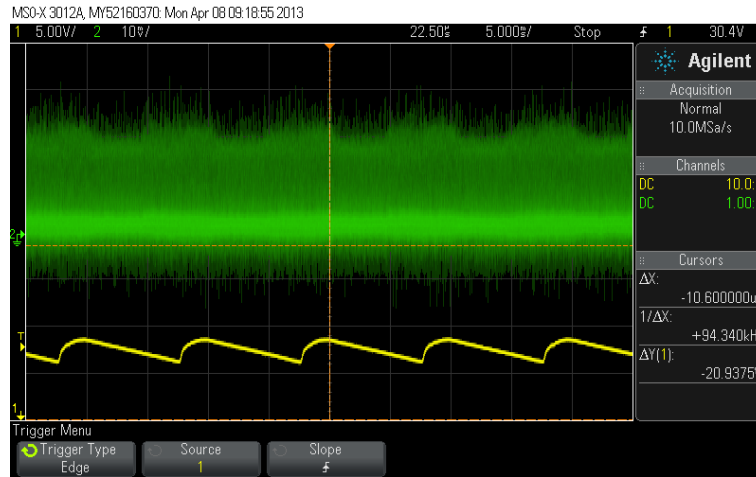


Figure E.1: Rectified voltage(Yellow) and input current(green)

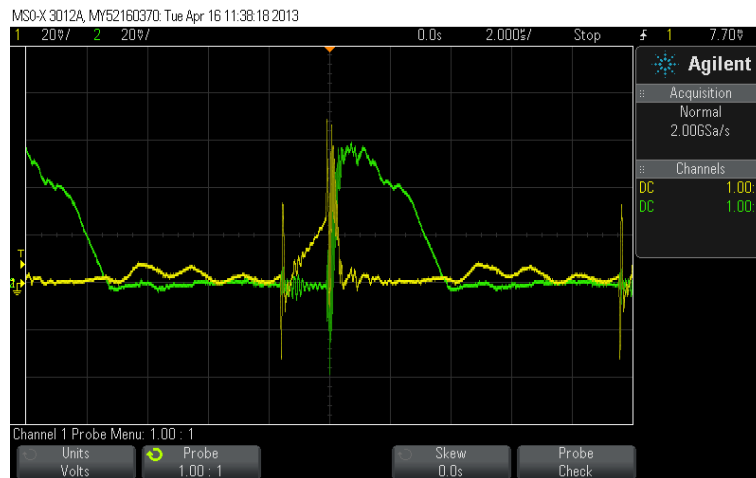


Figure E.2: Primary current(Yellow) and secondary current(green)

dissipation of secondary and auxiliary of the transformer. Whereas the secondary side of the transformer behave like a current source (because we don't use a capacitor as rectifier), the output voltage over the LED will always be around 30 V when conducting the current. At the auxiliary side of the transformer this power dissipation is related to the winding ratio $\frac{N_s}{N_{aux}} = \frac{P_s}{P_{aux}} = \frac{18}{10}$. As we are limited at the auxiliary side for power dissipation we can't go up with the duty cycle if we don't dissipate more energy at the auxiliary side.

To have more power dissipation at the auxiliary we connect a high power potentiometer of

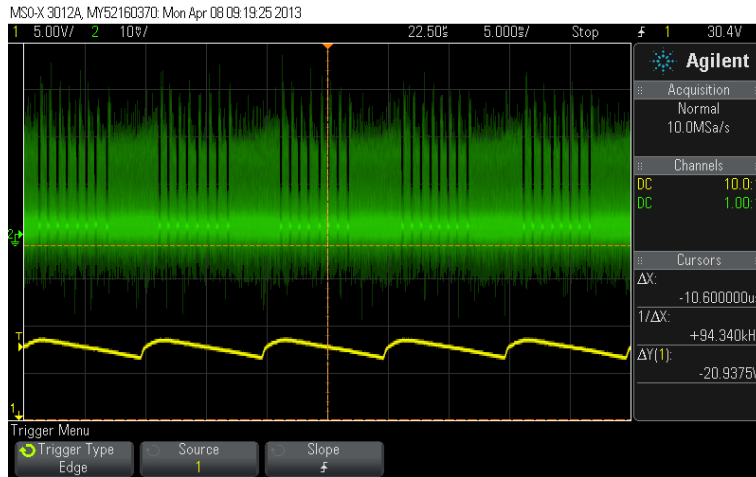


Figure E.3: Primary current(green) and rectified input voltage(Yellow)

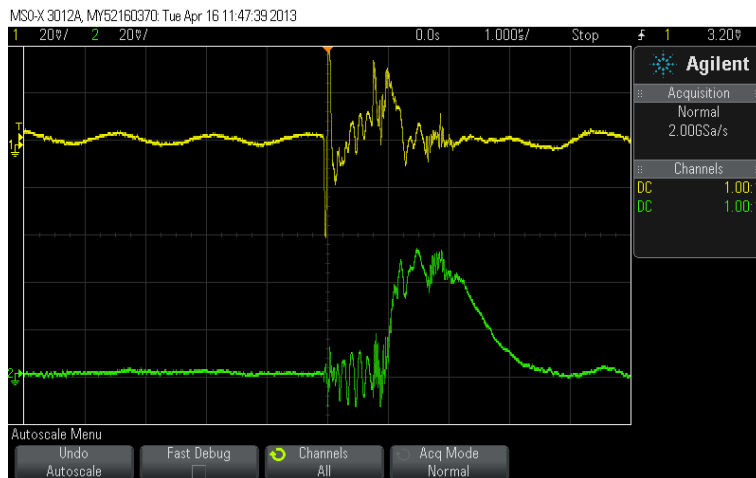


Figure E.4: Primary current(Yellow) and secondary current(green)

125 to 0 Ω .

$$P_{aux} = \frac{V_{aux}^2}{R} = \frac{16^2}{125} = 2W$$

$$P_{sec} = P_{aux} \cdot \frac{N_s}{N_{aux}} = 3.6W$$

The LEDs use normally 9 W when in typical use, with a duty cycle of 50 % this should be 4.5 W, what can be achieved by lowering the resistance at auxiliary. This permits us to raise

the duty cycle. In figure E.5 we can see the output current through and output voltage over the LED. The voltage over the potentiometer at the auxiliary of 125 Ω is 13 V .

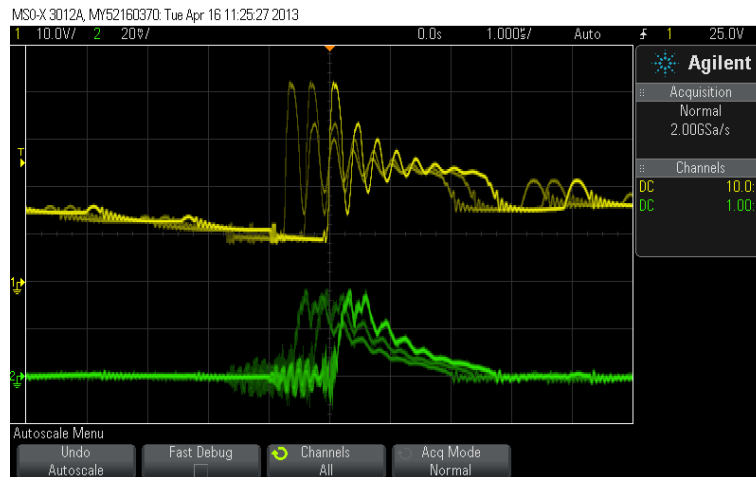


Figure E.5: Voltage over the LED (Yellow) and current through the LED(green)

In Figure E.6 we went up to an auxiliary voltage of 15.6 V. We didn't go higher because the current peaks at the secondary are up to 600 mA. This is already at the limits of the maximum ratings of our LEDs. So this time the duty cycle is limited by the LED maximum ratings.

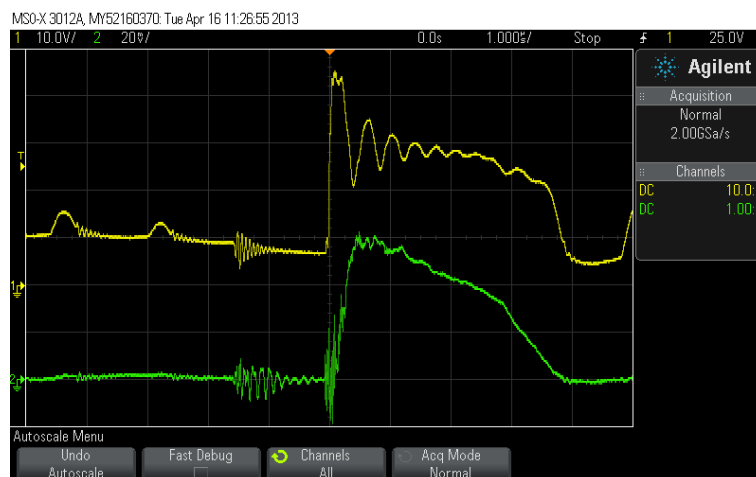


Figure E.6: Voltage over the LED (Yellow) and current through the LED(green)

E.4 Conclusion

To get a higher efficiency of our circuit we need to have another winding ratio between auxiliary and secondary. We can also give the feedback from the secondary and don't use the Auxiliary. The power supply for the other components can be created serial or parallel with the LED (that's harder to achieve a high efficiency). If we want to lower the current peak to our LED we need to have a higher primary inductance so the currents at primary and secondary charges and discharges the transformer slower. We can easily calculate this without taking losses and other side effects into account with the following formulas.

$$V_{prim} = L_{prim} \frac{\Delta i_{prim}}{t_{on}}$$

$$\Delta i_{prim} \cdot N_{prim} = \Delta i_{sec} \cdot N_{sec}$$

Where as the transformer at secondary side is seen as current supply, the voltage at the secondary side is determined by the characteristics of the LED. This voltage is around 30 V when using current peaks around 700mA. For choosing the winding ratio we will respect the general formula: $\frac{V_{prim}}{V_{sec}} = \frac{320}{30} = \frac{N_{prim}}{N_{sec}} = k$ what gives a winding ratio of 10,6. Using this in previous equations we can calculate L_{prim} for a duty cycle of 50% at a frequency of 100 kHz and with a current peak through the LED of 700 mA.

$$L = V_{prim} \frac{t_{on}}{\frac{\Delta i_{sec}}{k}} = 320V \frac{5\mu s}{\frac{0.7A}{10.6}} = 24.23mH$$

E.5 Results Test 2 of circuit 1

During this tests we worked with circuitry 1 and expanded it with the extra components on a breadboard. The schematic of the extra circuitry is shown in figure E.7a. The used mosfet was an IRF2804, this has a V_{GS} threshold voltage between 2 and 4 V. The voltage we get from the FPGA did not reach the threshold voltage of the mosfet, so we used a simple offset circuit shown in E.7b to give an offset to the signal from the FPGA. With the potentiometer in the feedback loop we could control the current true the LED and made the current go from 0 A to 600 mA. This gives a V_{AUX} around 17 V what is still in the limits of the POWERSWITCH chip V_{CC} . The results of the voltage over the LED, the current through the LED and the signal from the FPGA are shown in figures E.8, E.9 and E.10.

Considering the figures, we notice the presence of spikes on the signal due to the flyback current peaks. This is not really seen at the Receiver circuit at a frequency of 100 kHz, so we want to try to eliminate this ripple. Because this new circuitry is working like we wanted a new PCB is made.

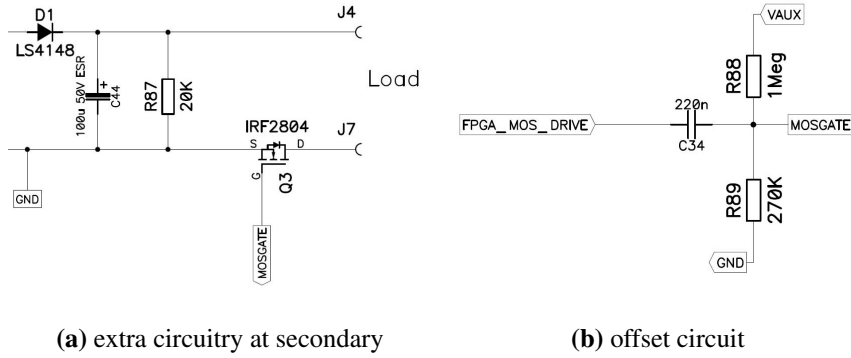


Figure E.7: extra circuitry on breadboard

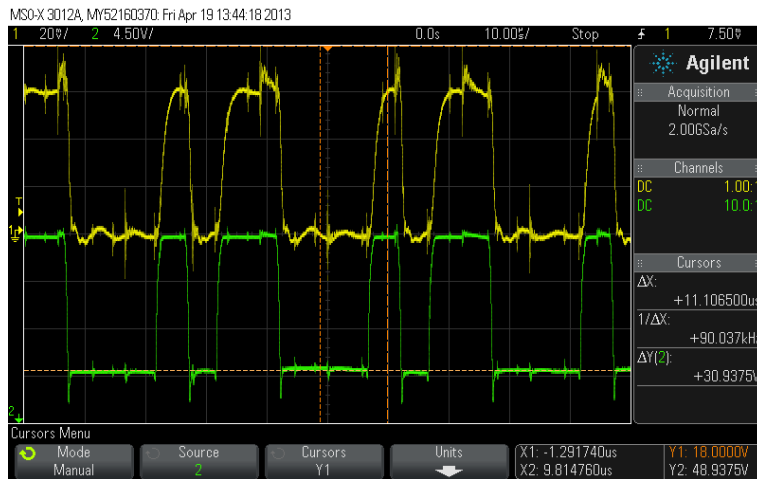


Figure E.8: Current through LED(Yellow) and voltage over the LED (17.8V to 30.7V)(green)

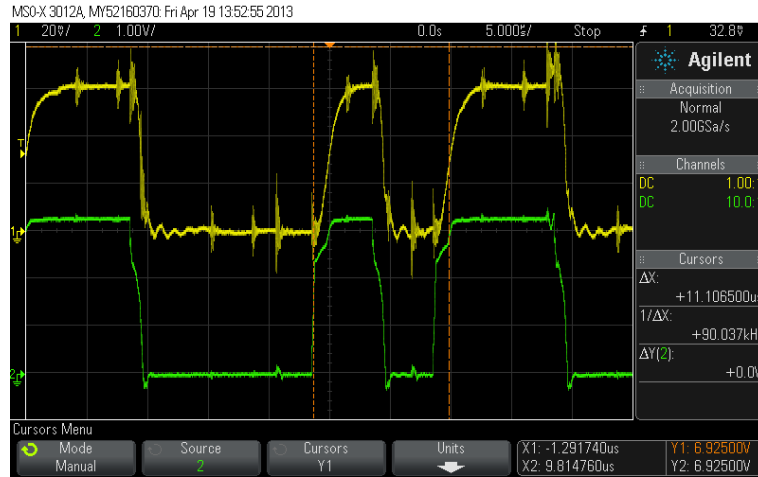


Figure E.9: Current through LED(Yellow) and signal from FPGA (green)

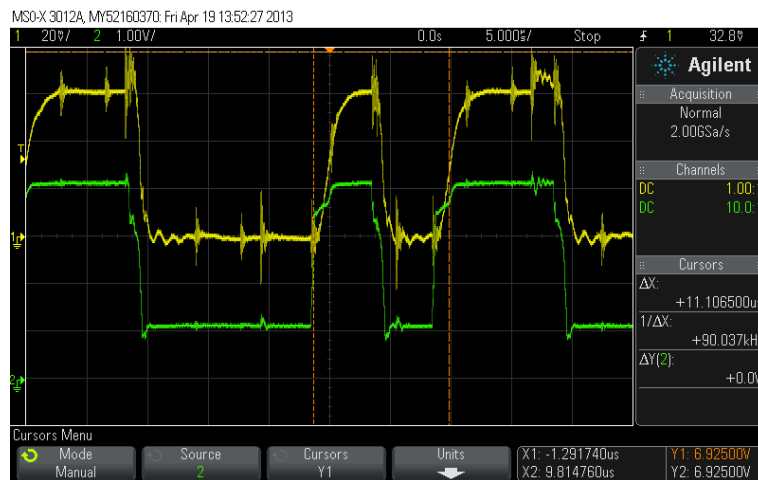


Figure E.10: Current through LED(Yellow) and signal at gate(green))

Appendix F

Correlation with an n-bit register

In this chapter, we will discuss the results from the correlation filter using an n-register LFSR (with n going from 6 to 15). We will investigate whether the correlation is better for longer sequences.

We used one or five LEDs that were sending an n-bit sequence and filtered the received signal with the sent bit sequence(s). We always used enough packets so we could see several peaks. The results are discussed below.

F.1 n = 6

We started sending a 6-register LFSR sequence by one LED. If we filtered the received signal with the sent bit sequence, we can see several peaks (see figure F.1). The bit sequence is thus recognized. The values of the peaks are located between 20 and 25. The noise is small and is located around zero.

If all 5 LEDs are burning, the several sent bit sequences are not so easy to distinguish. If we place the receiver near the first three LEDs, the bit sequences sent by these LEDs are still recognized as you can see in the figures below. We can see that the noise is increased. The values of the peaks are located around 30. These values depends on several factors: the distance (and angle) between sender and receiver, the sent bit sequence, the number of LEDs that are burning, illumination of the LEDs,...

If we filtered the received signal with the sequence sent by the other two LEDs, we couldn't see clear peaks because these LEDs are located too far from the receiver.

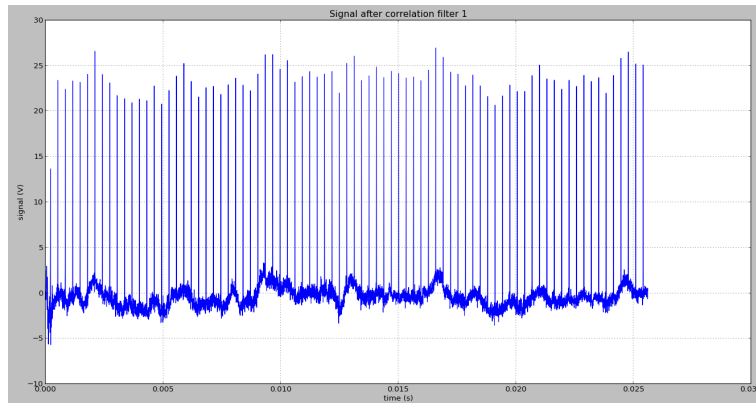


Figure F.1: Graph received signal after correlation filter (6-register LFSR), one LED burning

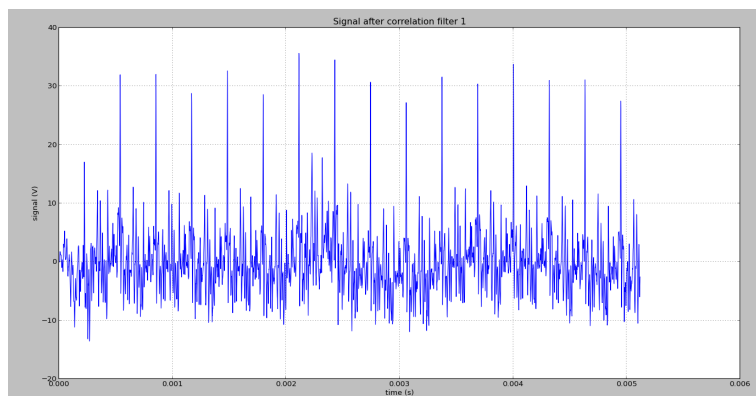


Figure F.2: Graph received signal after correlation filter 1 (6-register LFSR), 5 LEDs burning

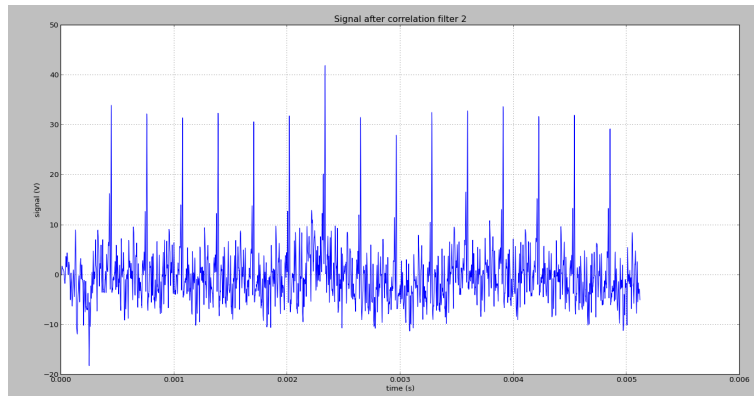


Figure F.3: Graph received signal after correlation filter 2 (6-register LFSR), 5 LEDs burning

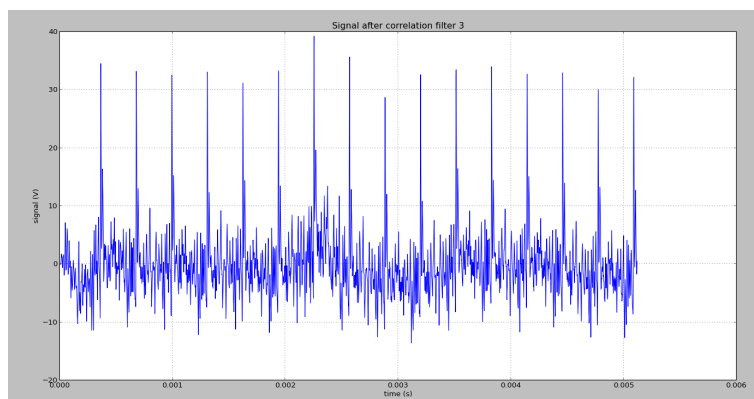


Figure F.4: Graph received signal after correlation filter 3 (6-register LFSR), 5 LEDs burning

F.2 $n = 7$

If we used a 7-register LFSR sequence sent by the 5 LEDs, the filtered signal is shown below. The values of the peaks are higher than when we are sending a 6-register LFSR sequence (about 50). The noise is also increased, but the peaks are better visible.

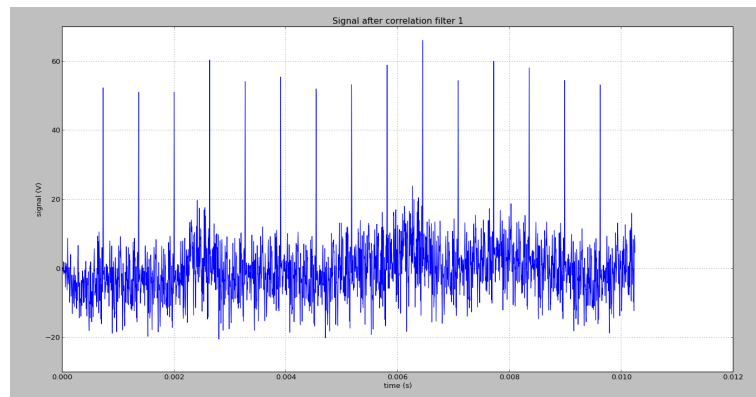


Figure F.5: Graph received signal after correlation filter 1 (7-register LFSR bit LFSR), 5 LEDs burning

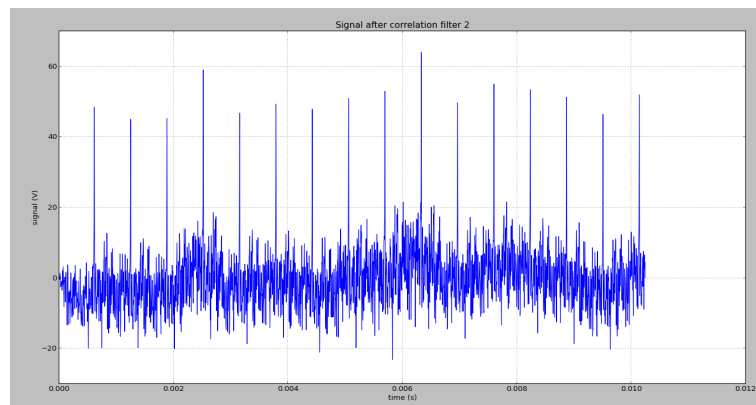


Figure F.6: Graph received signal after correlation filter 2 (7-register LFSR), 5 LEDs burning

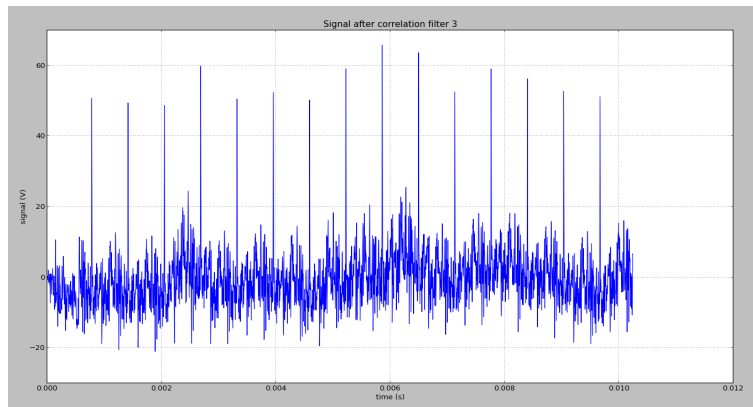


Figure F.7: Graph received signal after correlation filter 3 (7-register LFSR), 5 LEDs burning

F.3 $n = 8$

When sending an 8-register LFSR sequence, we can also see an increase of the values of the peaks (about 90), and an increase of the noise.

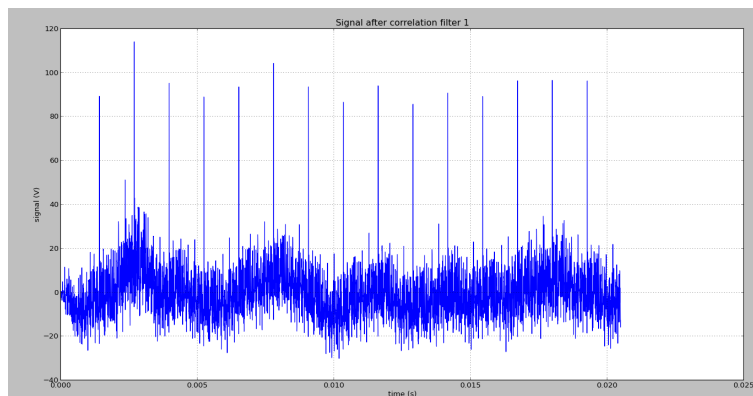


Figure F.8: Graph received signal after correlation filter 1 (8-register LFSR bit LFSR), 5 LEDs burning

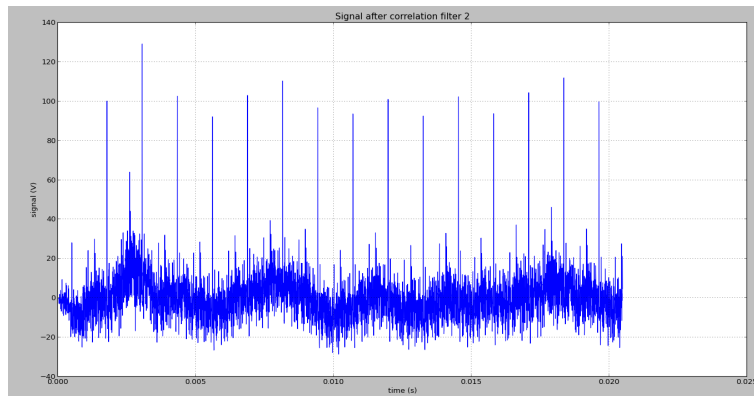


Figure F.9: Graph received signal after correlation filter 2 (8-register LFSR), 5 LEDs burning

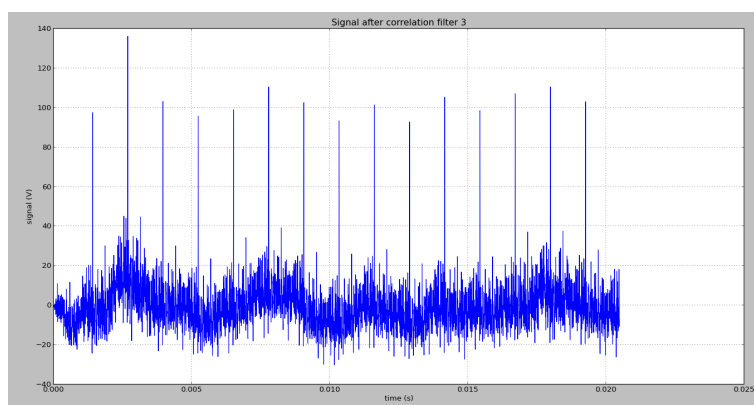


Figure F.10: Graph received signal after correlation filter 3 (8-register LFSR), 5 LEDs burning

F.4 $n = 9$

When sending a 9-register LFSR sequence, we can also see an increase of the values of the peaks (about 170). It takes more time for Python to do the calculations and to plot the several graphs.

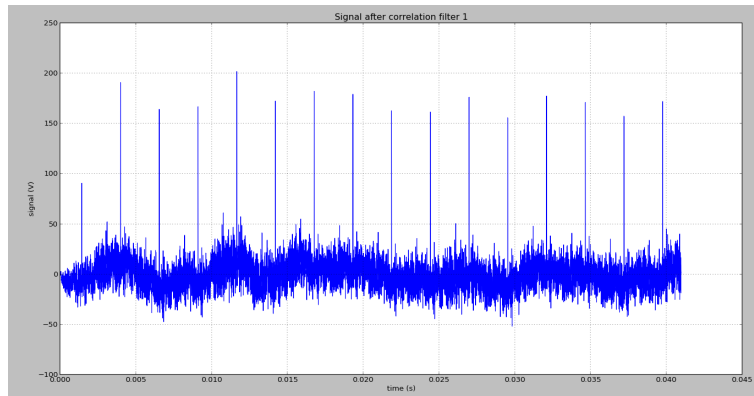


Figure F.11: Graph received signal after correlation filter 1 (9-register LFSR), 5 LEDs burning

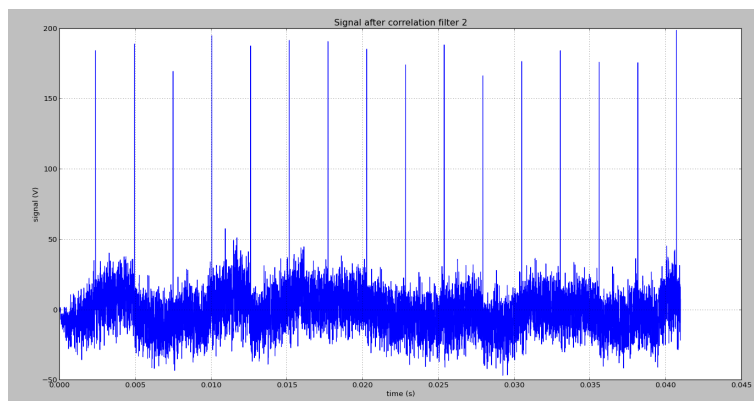


Figure F.12: Graph received signal after correlation filter 2 (9-register LFSR), 5 LEDs burning

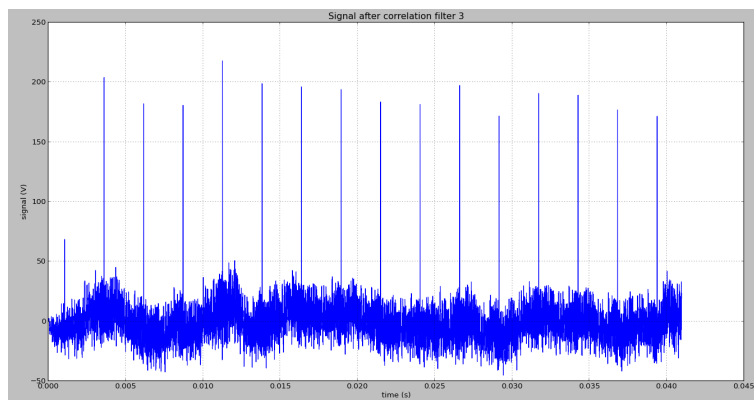


Figure F.13: Graph received signal after correlation filter 3 (9-register LFSR), 5 LEDs burning

F.5 n = 10

When we sent a 10-register LFSR sequence by one LEDs, very clear peaks are visible. The noise is again located around zero and the values of the peaks are located between 300 and 400.

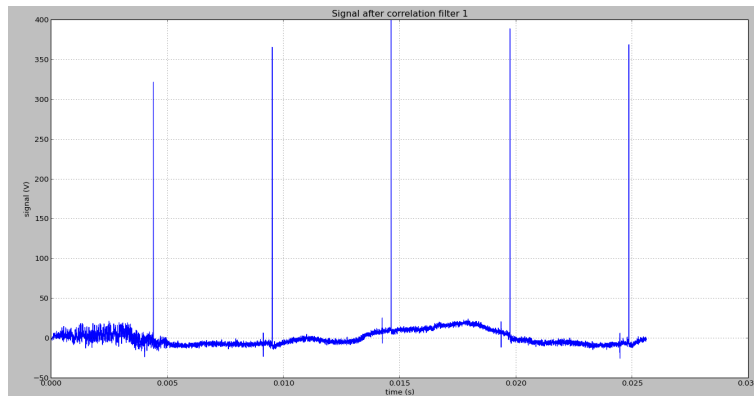


Figure F.14: Graph received signal after correlation filter (10-register LFSR), one LED burning

When all LEDs are burning and we filtered the received signal with the first three sequences, we get a result as seen below. We get high peaks, located around 400.

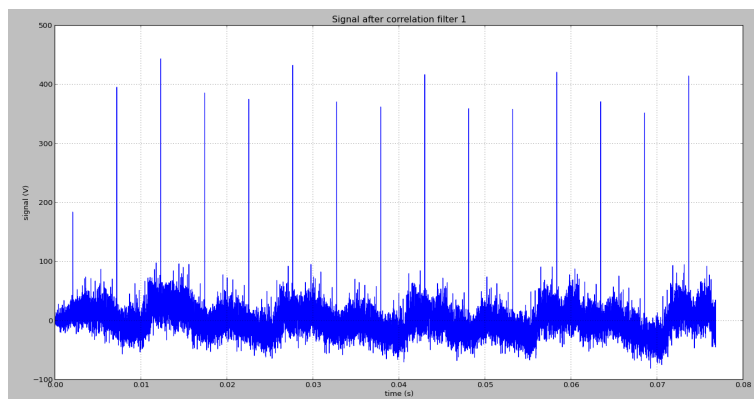


Figure F.15: Graph received signal after correlation filter 1 (10-register LFSR), 5 LEDs burning

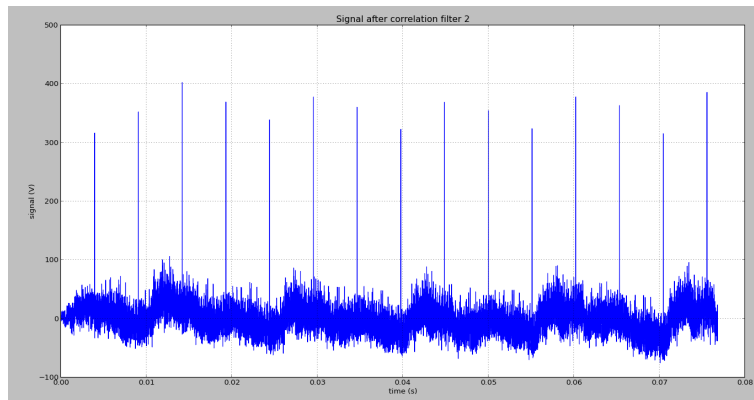


Figure F.16: Graph received signal after correlation filter 2 (10-register LFSR), 5 LEDs burning

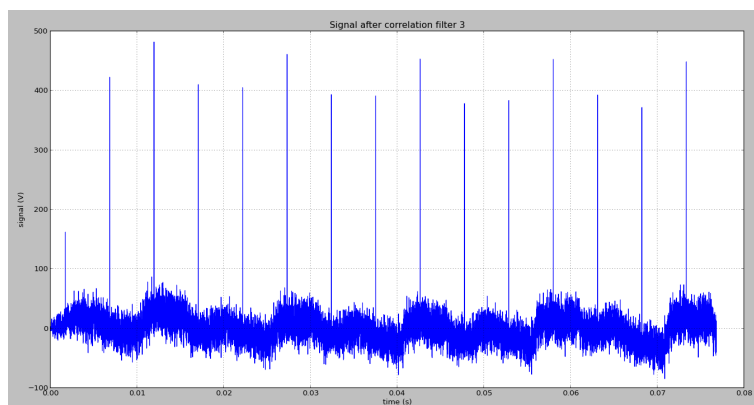


Figure F.17: Graph received signal after correlation filter 3 (10-register LFSR), 5 LEDs burning

F.6 $n = 11$

Because the calculations in Python took too long, we started plotting just one graph. One LED was burning and we filtered the received signal with the sent bit sequence. Peaks are located around 800.

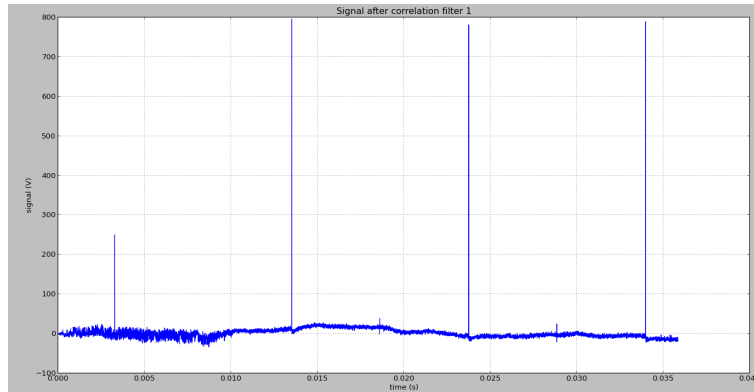


Figure F.18: Graph received signal after correlation filter (11-register LFSR), one LED burning

F.7 $n = 12$

When sending a 12-register LFSR sequence, peaks are located around 1500.

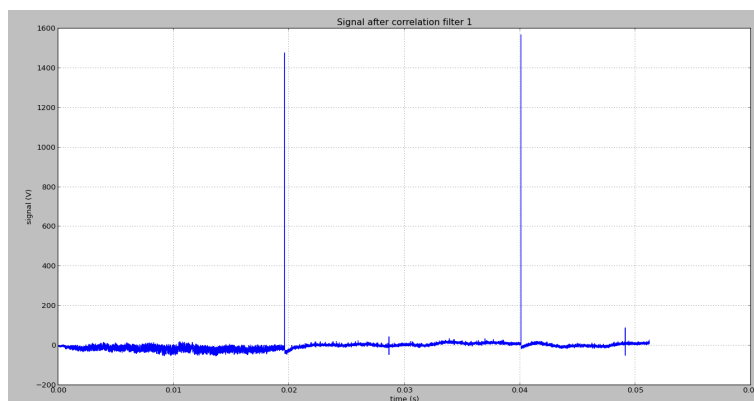


Figure F.19: Graph received signal after correlation filter (12-register LFSR), one LED burning

F.8 $n = 13$

When sending a 13-register LFSR sequence, the values of the peaks are above 3000. The calculations in Python took several minutes.

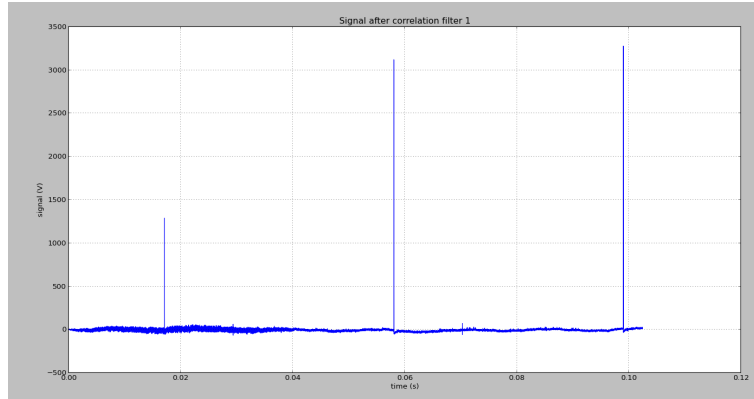


Figure F.20: Graph received signal after correlation filter (13-register LFSR), one LED burning

F.9 $n = 14$

When sending a 14-register LFSR sequence, we get peaks above 6000.

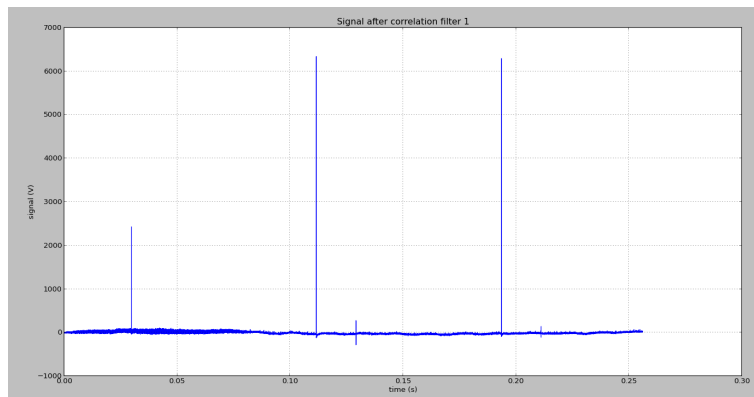


Figure F.21: Graph received signal after correlation filter (14-register LFSR), one LED burning

F.10 $n = 15$

When sending a 15-register LFSR sequence, there was a peak above 12000. The last peak is less high. Maybe, the sequence was too long and there was a little shift in the signal. Because the calculations took more than 15 minutes, we decided not to try any higher bit sequences.

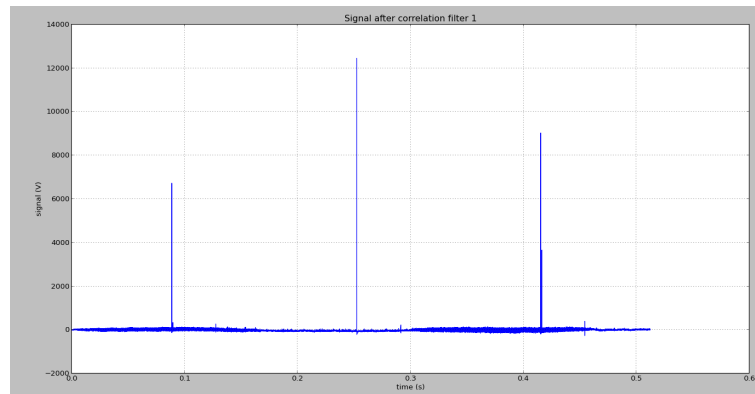


Figure F.22: Graph received signal after correlation filter (15-register LFSR), one LED burning

F.11 Conclusion

After testing several bit sequences, we can draw some conclusions. Peaks are becoming higher when sending longer bit sequences. The longer the bit sequence, the more visible the peaks are and the bigger the resolution. So, it should be best to work with a bit sequence as long as possible. Nevertheless, the calculations took longer and longer when sending longer bit sequences. For the indoor positioning, we decided to work with a 10-register LFSR sequence to have a rather high resolution and not to wait too long for the calculations. The calculations took about 15 seconds when working with a 10-register LFSR sequence.

Appendix G

Data transmission

In this chapter, we will discuss the possibility to transmit data next to the possibility to determine the position. If we invert the bit sequence during some periods, we will see positive and negative peaks after the correlation filter. So we can distinguish a digital one and a digital zero.

We will send a sequence of 1023 bits (10-register LFSR). We want to send the 12-bit data 110110100100. We will discuss two possibilities: we can transmit one bit during one period of the 10-register LFSR sequence or we can transmit one bit during several periods of the 10-register LFSR.

G.1 One sequence per bit

First, we tried to transmit a data bit during one period of the 10-register LFSR sequences. To transmit the 12-bit data, we need thus 12 periods of the bit sequence. The result after the correlation filter is shown below. A positive peak represents a digital zero, a negative peak represents a digital one. As can be seen, not all peaks are clearly visible. Some peaks can barely be seen because they are not larger than the noise. We can only see a clear peak when we send two successive ones or two successive zeros.

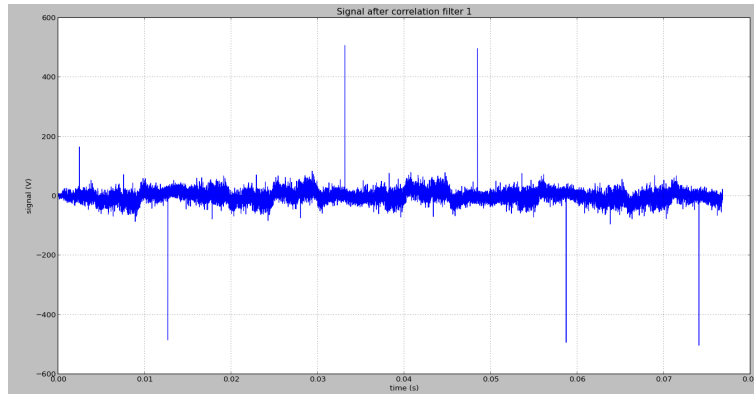


Figure G.1: Graph received signal after correlation filter (10-bit LFSR), one LED burning

G.2 Two sequences per bit

Then, we tried to transmit a data bit during two periods of the 10-register LFSR sequence. We invert the bit sequence during two periods if we want to send a digital one. In this case, we should see two negative peaks. The result after the correlation filter is shown below. If we send a digital one or zero, only one peak is visible. If we send two successive ones or zeros, only three peaks are visible. So, one peak isn't visible. Now, we can recognize the sent data: 110100100110. If we have for instance 3 successive negative peaks, we have send 11.

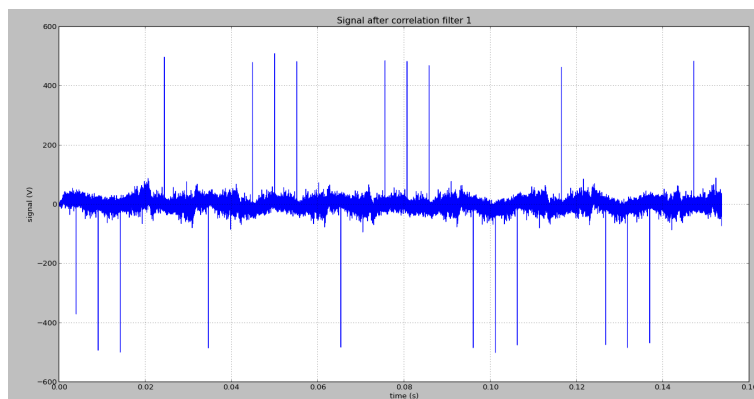


Figure G.2: Graph received signal after correlation filter (10-bit LFSR), 5 LEDs burning

G.3 Five sequences per bit

Finally, we also tried to transmit a data bit during five periods of the 10-register LFSR sequence. Now we can see 4 peaks per bit, the fifth peak is a little bit visible. If we have for instance 9 clear positive peaks, we have send 00. We can also recognize the sent data.

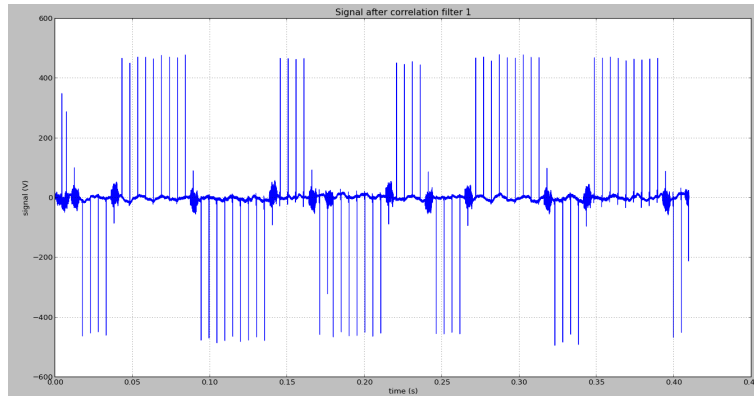


Figure G.3: Graph received signal after correlation filter (10-bit LFSR), one LED burning

G.4 Conclusion

We can send data besides the indoor positioning by looking at negative and positive peaks after the correlation filter. A negative peak represents a digital one, a positive peak represents a digital zero. If we send a bit during one period of the bit sequence, not all peaks are visible. The sent data can't be recognized. If we send a bit during multiple periods of the bit sequence, we can recognize the sent data. There is always one peak that isn't very good visible. So, if we want to transmit data besides the positioning, we need minimum two periods of the bit sequence to transmit a bit. In this case, there is at least one clear peak per bit.

Appendix H

Schematics

H.1 Schematic 1

This section contains the schematic of the first designed circuit and accompanying explanations. The PCB of this circuit is shown below. Only the components for the first tests are mounted.

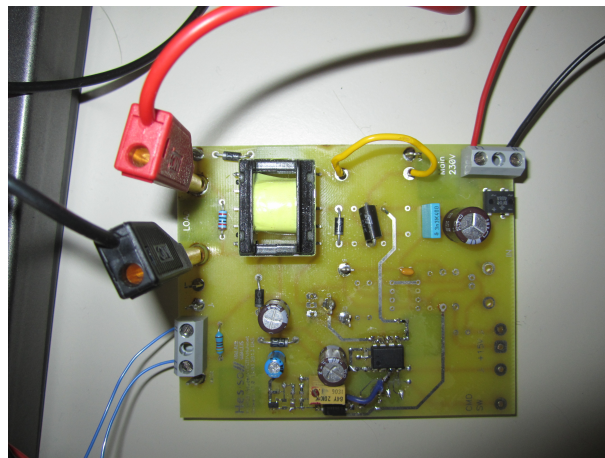
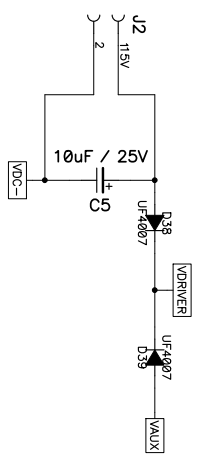
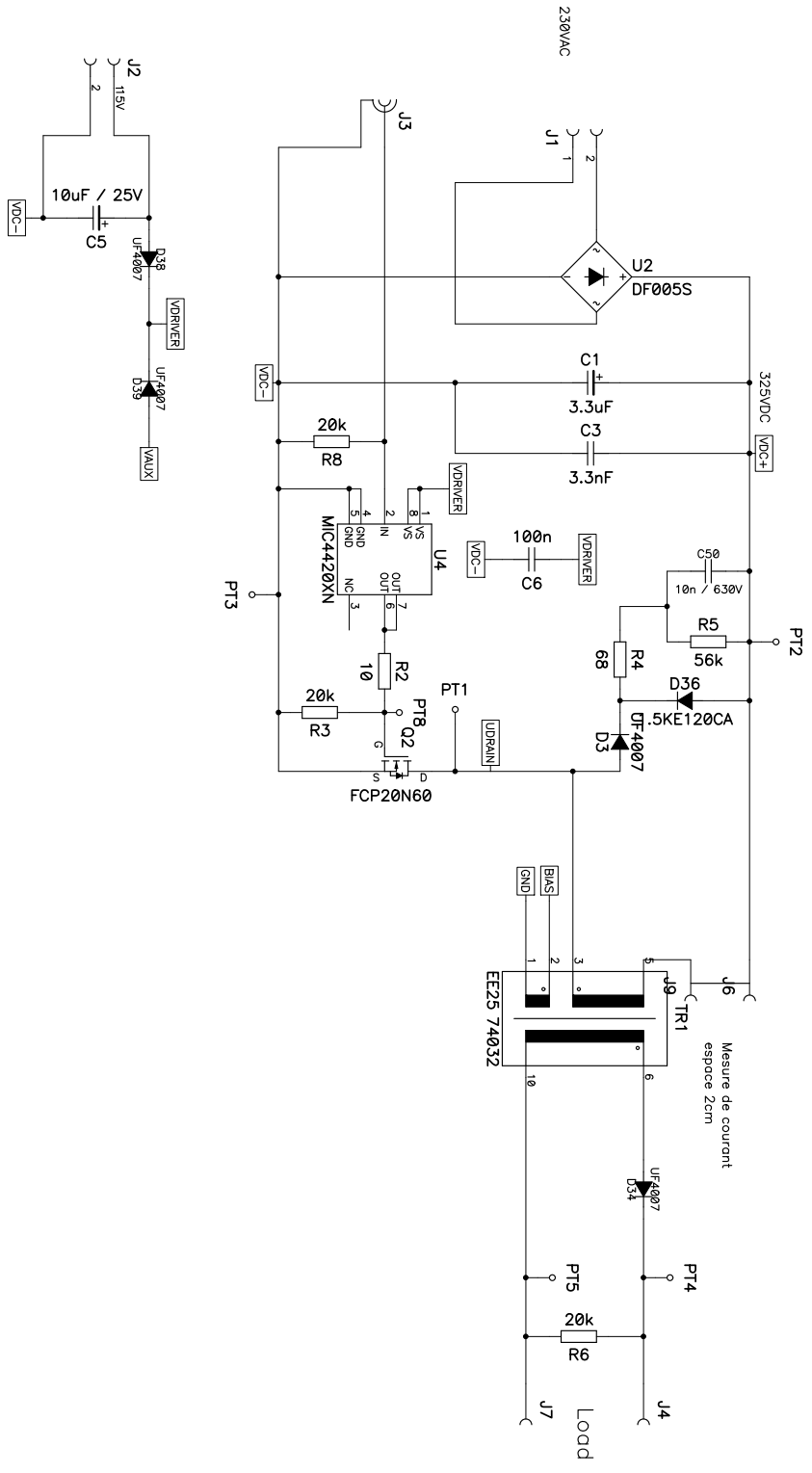
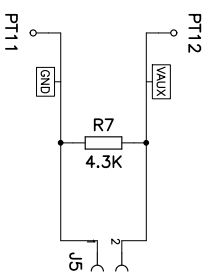
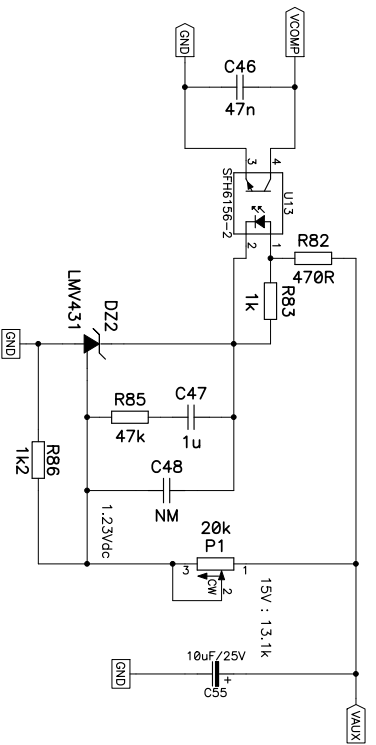
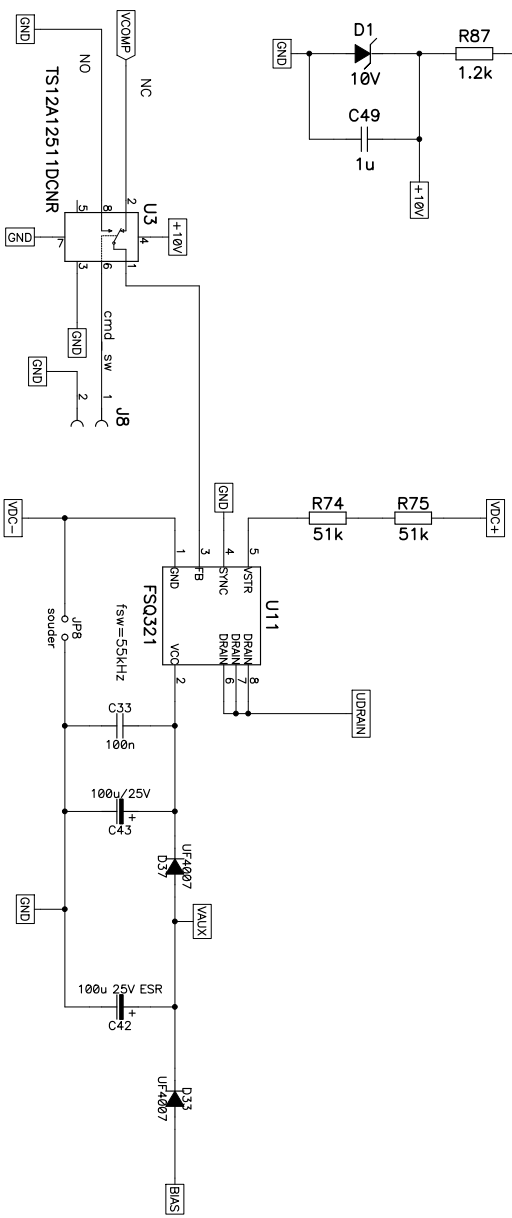
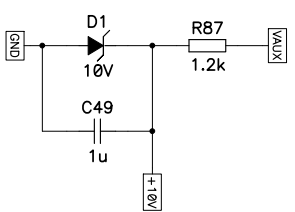


Figure H.1: PCB sender schematic 1, without unneeded parts



VLC LED DRIVER CIRCUIT		Sheet1	
Thesis			
HAUTE ECOLE VALAISANNE			
DES	12.03.2013	Didier Blatter & Rob Holvoet	
REV	V1.0		
1/3	{path}	Ortwerp V1_0.sch	



VLC LED DRIVER CIRCUIT

Thesis
HAUTE ECOLE VALAISANNE

Sheet2

DES	12.03.2013	Didier Blatter & Rob Holvoet
REV	V1.0	
2/3	{path}	Ortwerp V1_0.sch

H.1.1 Rectifier

To get an DC voltage we rectify the mains 230 V AC to a 320 V DC. First we rectify with the DF04424A bridge rectifier and use C1 as a reservoir capacitor to smooth the rectified AC output from the bridge. C2 is used as reserve for possible occurring current peaks. The choice of values of the capacitors are based on previous circuits made at HES-SO.

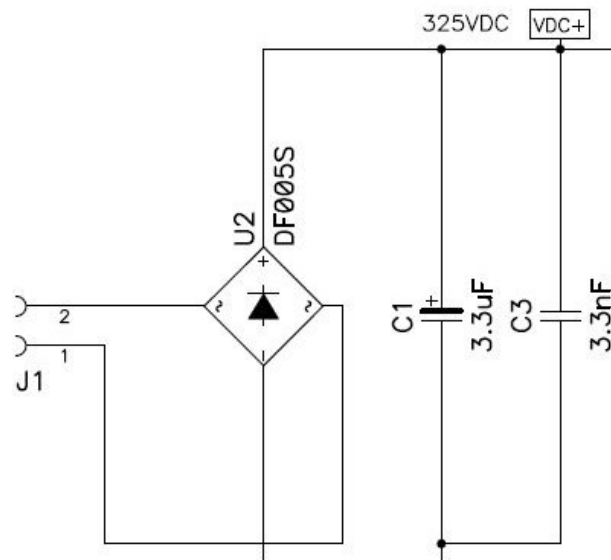


Figure H.2: Bridge rectifier and reservoir capacitors

H.1.2 Snubber

Snubbers are used to prevent the ringing on the drain of the used MOSFET. This design is also based on previous designs at HES-SO. In our first PCB we won't place the resistors nor the capacitor but implement the possibility to place them on a later stage. The zener diode D36 is a zener diode to limit the voltage over the MOSFET in OFF-MODE to 460 V (320+120).

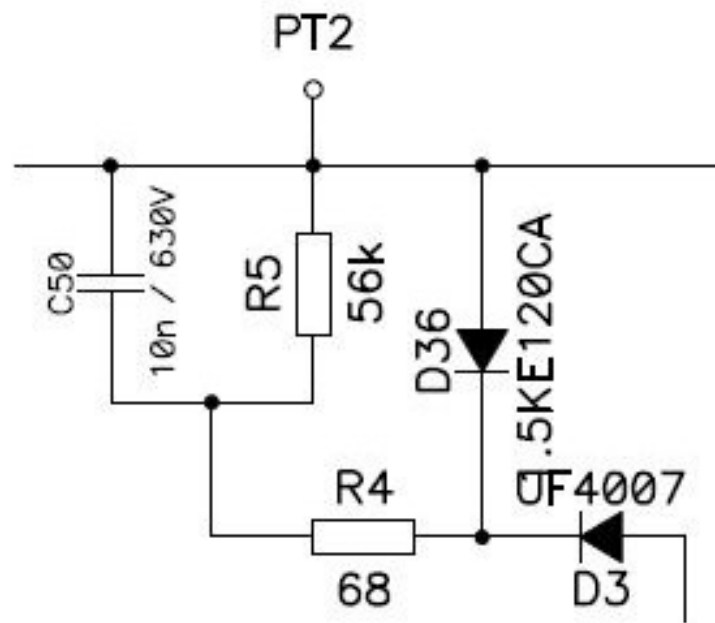


Figure H.3: Snubber

H.1.3 Mosfet and mosfet driver

For a flyback converter, a mosfet is needed to do the switching. Mosfets that are able to function with a drain-source voltage over 400 V need a high gate-source voltage to do the switching. That is why a mosfet driver is needed. R8 and R3 are pull down resistors . R2 limits the peak currents due to the parasitic capacitances of the mosfet. The smaller the resistance, the faster the mosfet responses.

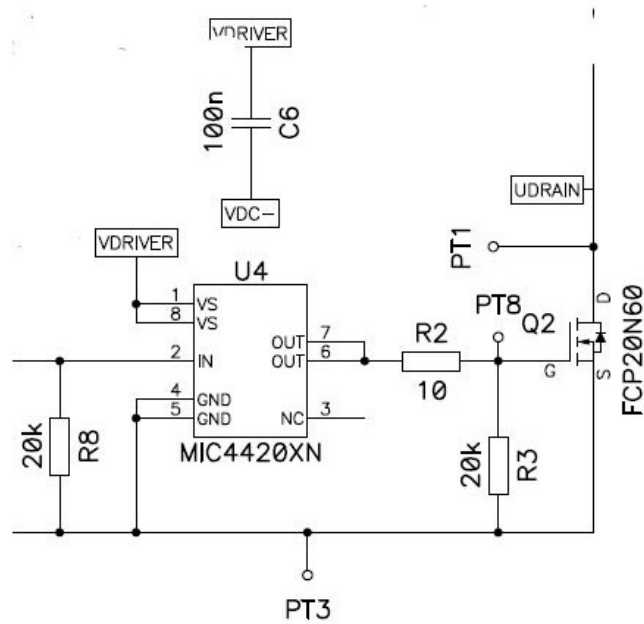


Figure H.4: Mosfet with mosfet driver

H.1.4 Power switch chip and analog switch

With the switch we can choose to turn the chip on or off. This is controlled by the FPGA. We need about 10 V to POWER the switch. We get this from V_{AUX} with a zener diode of 10 V. normally the chip gets power from V_{AUX} . Only for the start up V_{AUX} is 0. That is why he got pin VSTR, during start up he uses this voltage to load C33 and C43. This current may not get lost to Power other components on V_{AUX} , that is why we implement diode D37.

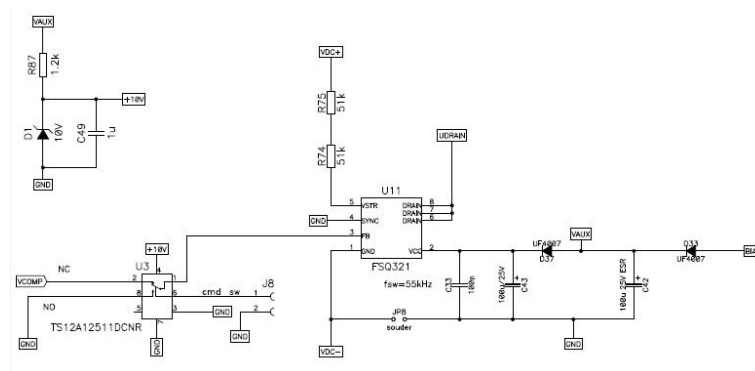


Figure H.5: Power switch chip

H.1.5 Feedback circuit

To let the chip know what voltage we get from the transformer and let him adjust his duty cycle we need a feedback circuit. The circuit we use is a typical feedback circuit with optocoupler. We implemented a potentiometer instead of fixed resistance to be able to adjust the feedback and so adjust the output voltage of the flyback.

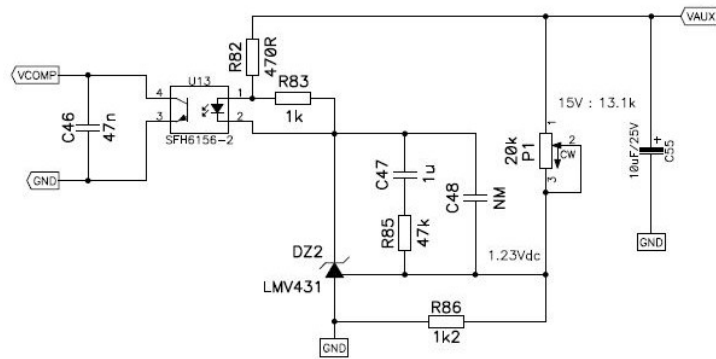


Figure H.6: Feedback circuit for powerswitch

H.2 Schematic 2

Here you can find the schematic of the receiving part.

The PCB of this circuit is shown below.

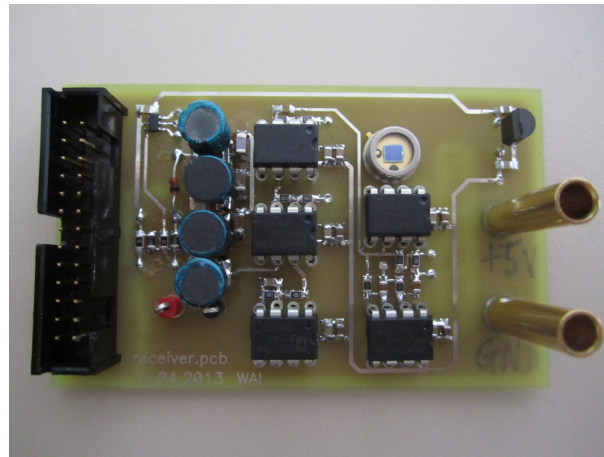
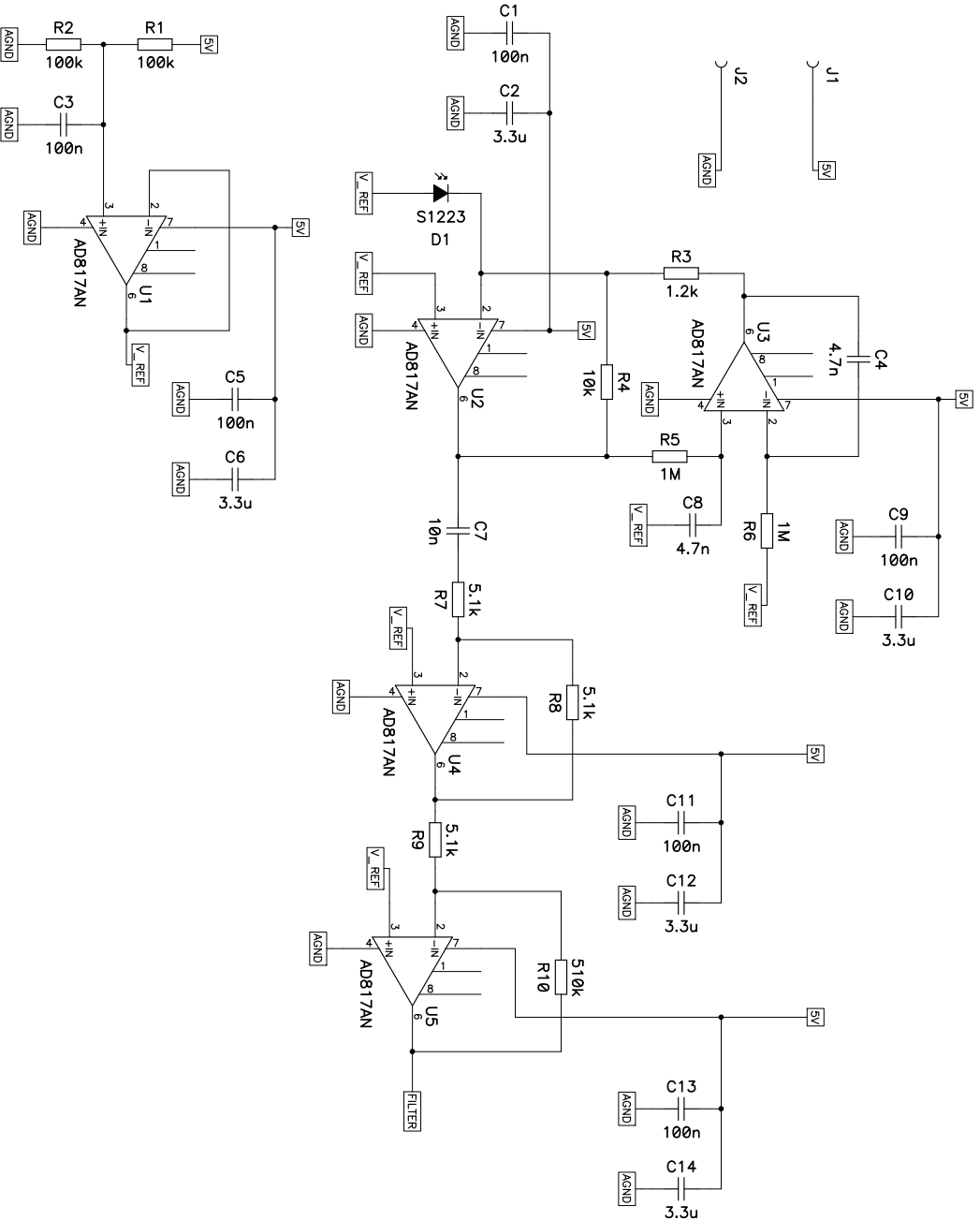


Figure H.7: PCB receiver



Receiver circuit
 {Title 2}

HAUTE ECOLE VALAISANNE

receiver

DES 11/04/2013 Jelle Smets

REV V1.0

1/5 {path} receiver.sch

1

2

3

4

5

6

7

8

9

10

1

2

3

4

5

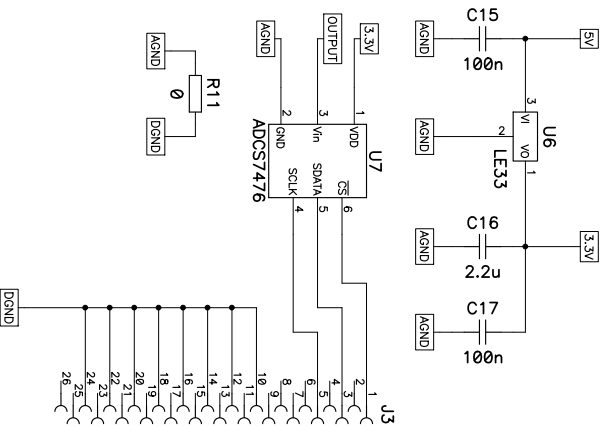
6

7

8

9

10



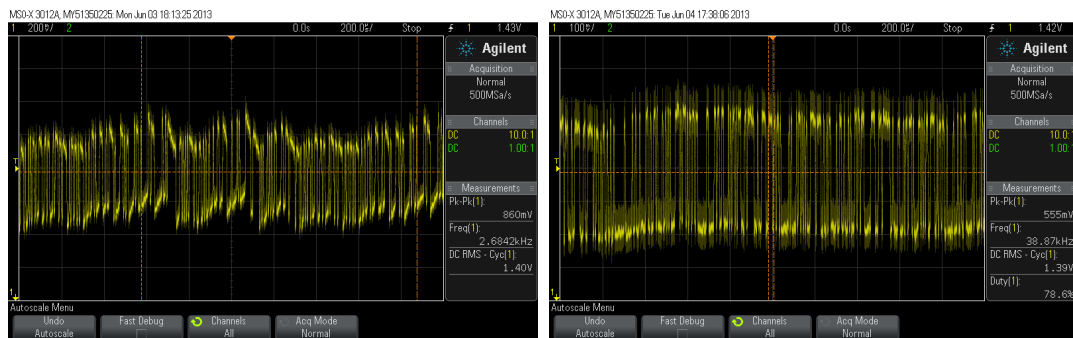
Receiver circuit		DES	11/04/2013	Jelle Smets
{Title 2}		REV	V1.0	
HAUTE ECOLE VALAISANNE		2/5	{path} receiver.sch	
regulator				

H.2.1 Ambient light rejection circuit

The first part of the receiver is the photodiode with a current-voltage converter and an ambient light filter in the feedback loop. This part is already extensively discussed in appendix C.

H.2.2 High pass filter

The next op-amp stage is a high pass filter with a gain of 1 and $f_{cut-off} = \frac{1}{2\pi \cdot 5100 \cdot 1 \cdot 10^{-6}} = 31.2$ Hz. This filter is used because we still had a DC component after the ambient light rejection circuit. Firstly, we used a high pass filter with $f_{cut-off} = 3.12$ kHz. But in the bit sequence at the output of the receiver we could notice a fluctuation of the DC offset (see figure H.8a). This phenomenon occurs because we used an RC high pass filter with a too high cut-off frequency. The lower the time constant $R \cdot C$, the more fluctuation will occur. So, we chose to work with a higher time constant (lower cut-off frequency). The difference between both is shown in figure H.8. It is clear that there is less fluctuation in the output signal. Because the cut-off frequency of the high pass filter is below 100 Hz, other lighting in the building will influence the signal of the receiver. But as the LEDs are also intended for the lighting in the room (so no other lights will be present), this is not really a problem.



(a) Signal output receiver with $f_{cut-off,HPF} = 3.12$ kHz (b) Signal output receiver with $f_{cut-off,HPF} = 31.2$ Hz

Figure H.8: Comparison output signal with different cut-off frequency of HPF

H.2.3 Amplifier

Then we use an inverting amplifier with a gain of 100, so we can detect the LED from farther distances and to use the full range of the ADC.

H.2.4 Voltage reference

Normally, op-amps need a dual supply for proper working. But since we want a single supply, we will use a voltage reference of 2.5 volt who acts as a virtual ground. The virtual ground replaces the normal ground in single supply op-amp circuits.

In the schematics, you can see that the reference was firstly made with two resistors of 100 k Ω and the supply voltage of 5 volt. As the 5 volt from the USB port is not so stable, we chose to make the reference with the 3.3 volt after the voltage regulator. This voltage is stable. We cut the path to the 5 volt and soldered a wire to the 3.3 volt. We also changed the upper resistor to 33 k Ω , so we had again a voltage of about 2.5 volt as the reference.

$$\Rightarrow V_{ref} = \frac{100k}{133k} \cdot 3.3 \text{ volt} = 2.48 \text{ volt}$$

The result of these modifications on the PCB can you see in the picture below. We also brought the photodiode higher and removed the 5 volt connectors. You can see that we have soldered two wires from the USB port to supply the circuit.

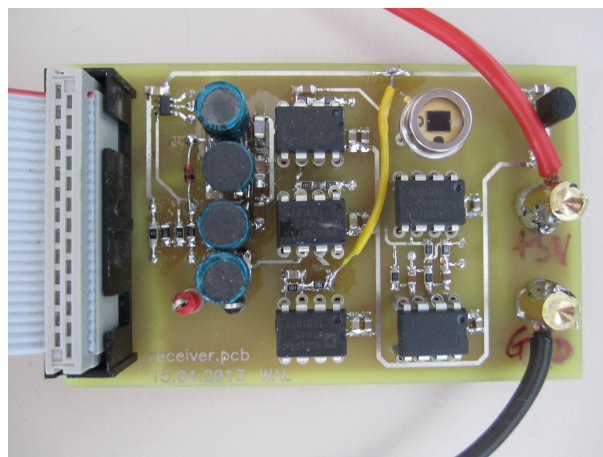


Figure H.9: PCB receiver with some modifications

H.2.5 LC filter

After the op-amp stages, we use an LC low pass filter with $f_{cut-off} \approx 500$ kHz to prevent aliasing. After the filter, a diode and a voltage divider are used. To make sure that the signal going to the ADC is always positive, we use a diode after the filter. To have the full range of the ADC, we use a voltage divider.

H.2.6 Voltage regulator

As the FPGA is supplied with 3.3 volt and the data we will send to the FPGA should be between 0 and 3.3 volt, we decided to supply the ADC also with 3.3 volt. Because the circuitry is supplied with 5 V, a voltage regulator is needed to have a stable 3.3 volt to feed the ADC.

H.2.7 Separation between analog and digital ground

For a proper working of the circuit, we will provide two separate ground planes: one analog ground plane and one digital ground plane. Then we connect the two ground planes at one single point with a 0Ω resistor.

H.2.8 Connection to the FPGA board

For connecting the ADC with the FPGA, we provide a 26-pin connector so we can easily exchange signals between the ADC and the FPGA.

H.3 Schematic 3

This section contains the schematic of the second designed sender circuit. The PCB of this circuit is shown below.

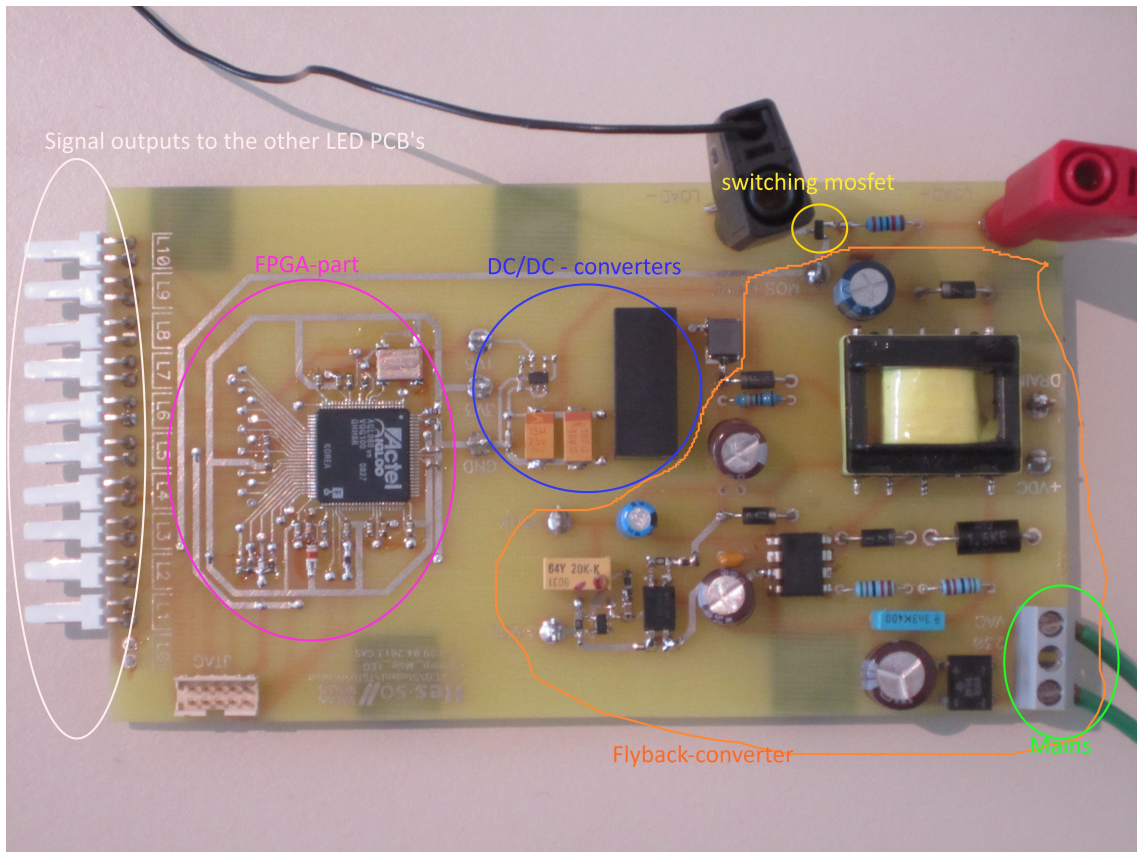
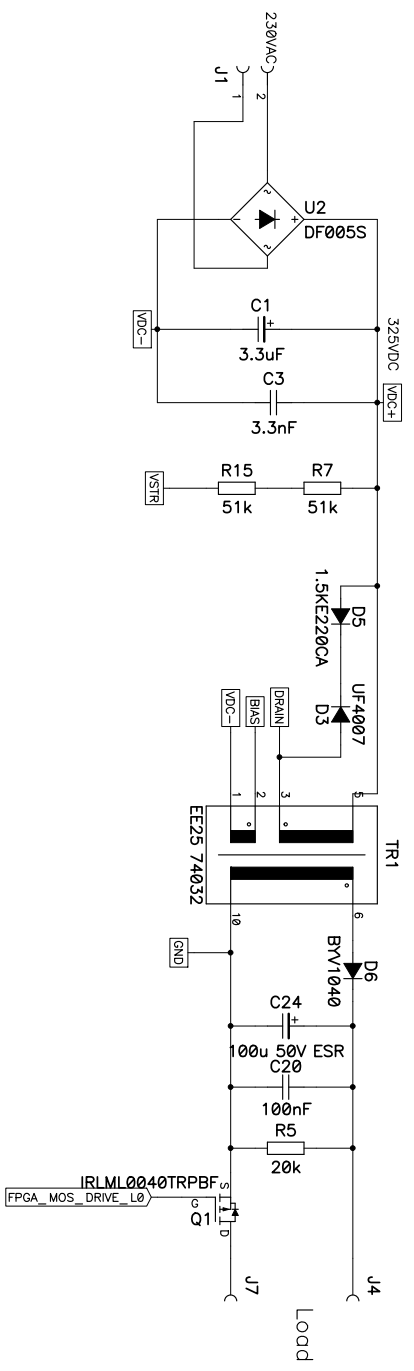
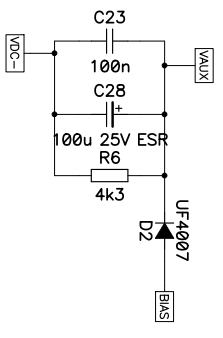


Figure H.10: PCB sender schematic 2, with FPGA mounted

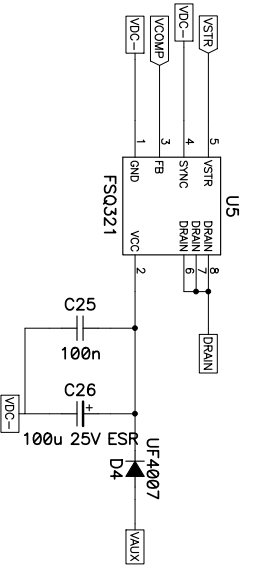
RECTIFIER + FLYBACK + VLC MOS



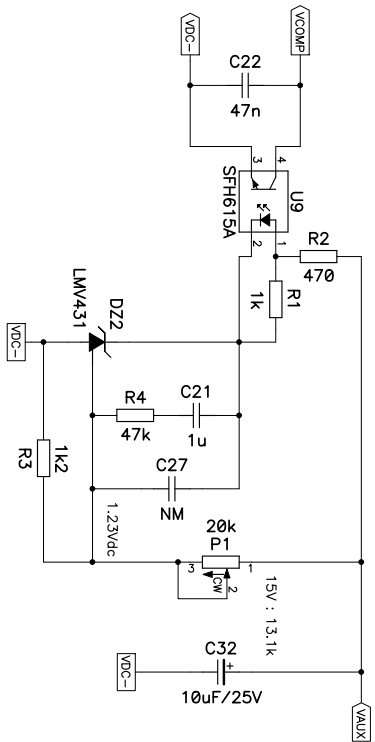
AUXILIARY



POWER MOS CHIP

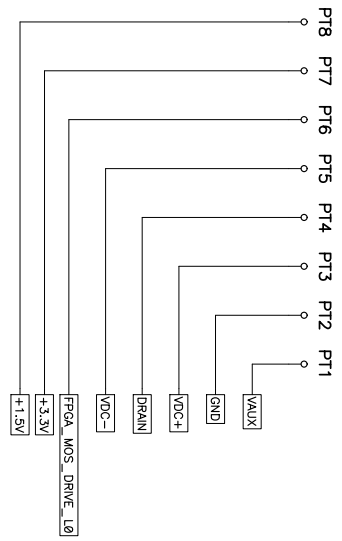


VLC LED DRIVER CIRCUIT		DES	21.05.2013	Rob Holvoet
Thesis		REV	V1.0	
HAUTE ECOLE VALAISANNE		Sheet1		
		REV	1/4	{path} Ortwerp_Main_LED_GAS.sch

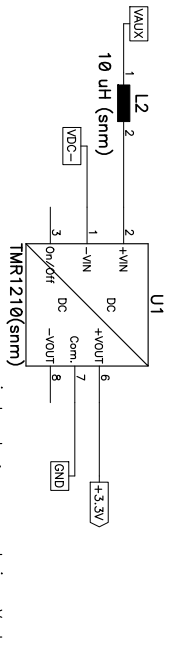


FEEDBACKLOOP

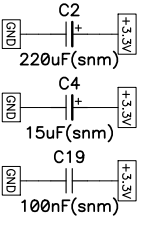
Measure points



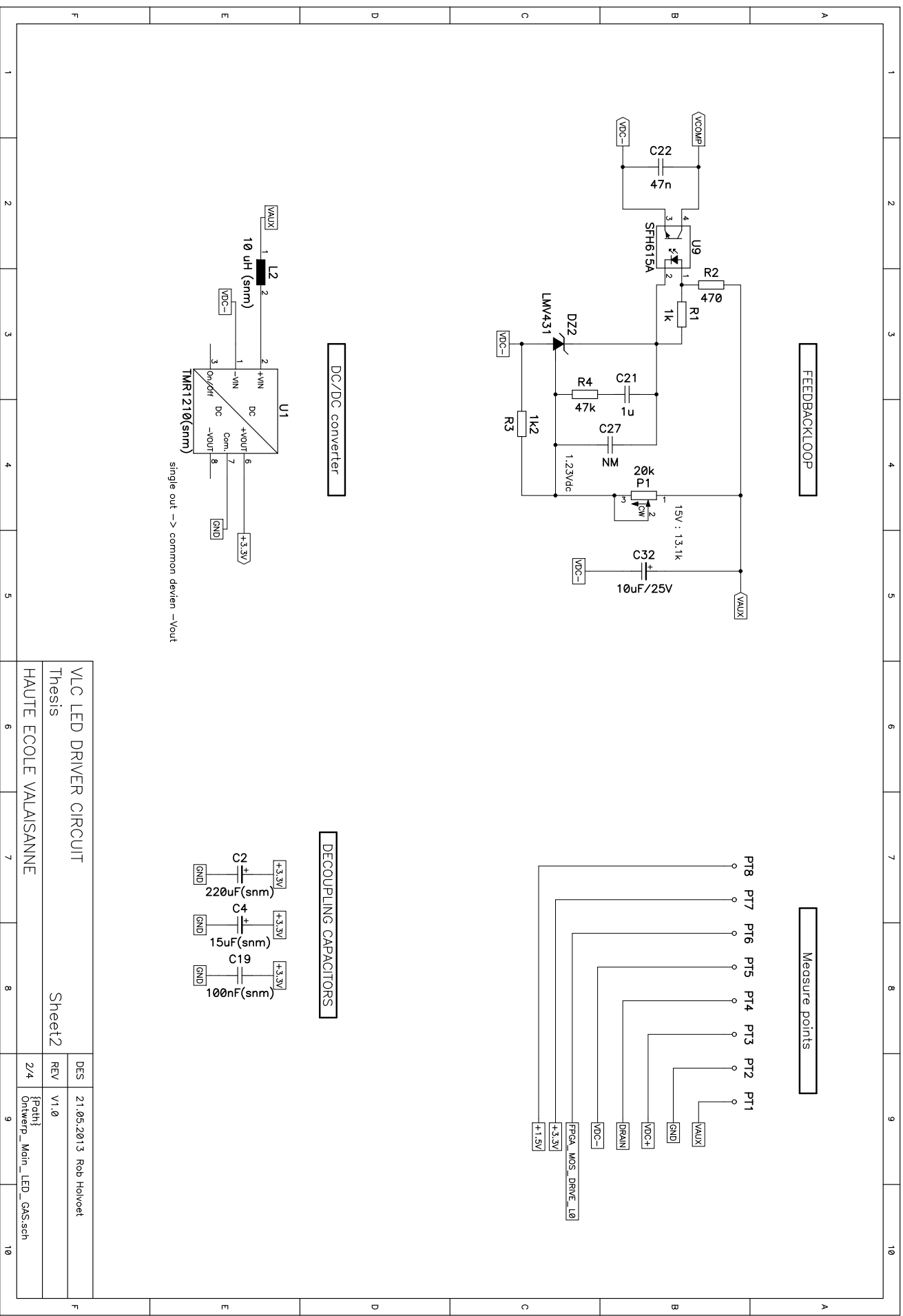
DC/DC converter

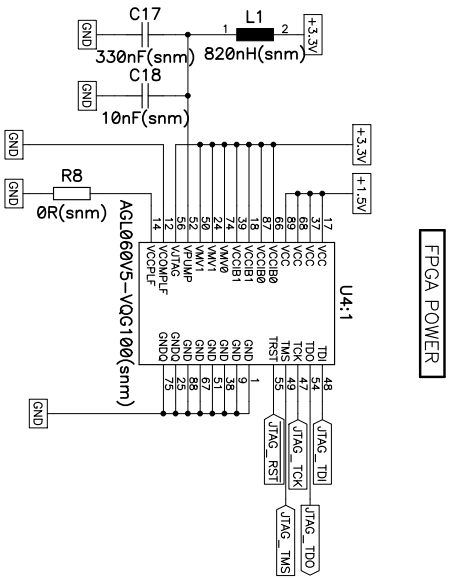


DECOUPLING CAPACITORS

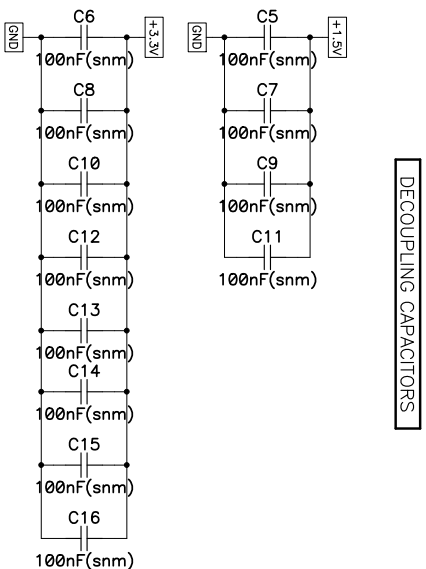
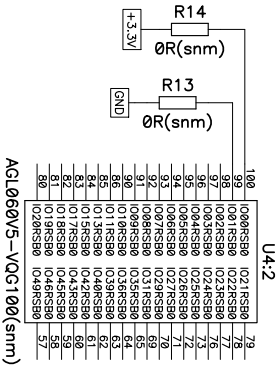
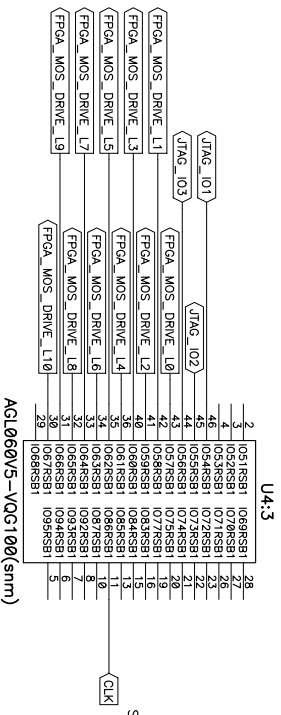


VLC LED DRIVER CIRCUIT		DES		21.05.2013		Rob Holvoet	
Thesis		REV		V1.0			
HAUTE ECOLE VALAISANNE		Sheet2		2/4		{path} Ortwerp_Main_LED_GAS.sch	



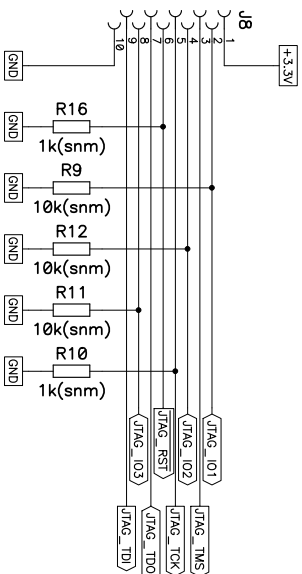


FPGA I/O



DECOUPLING CAPACITORS

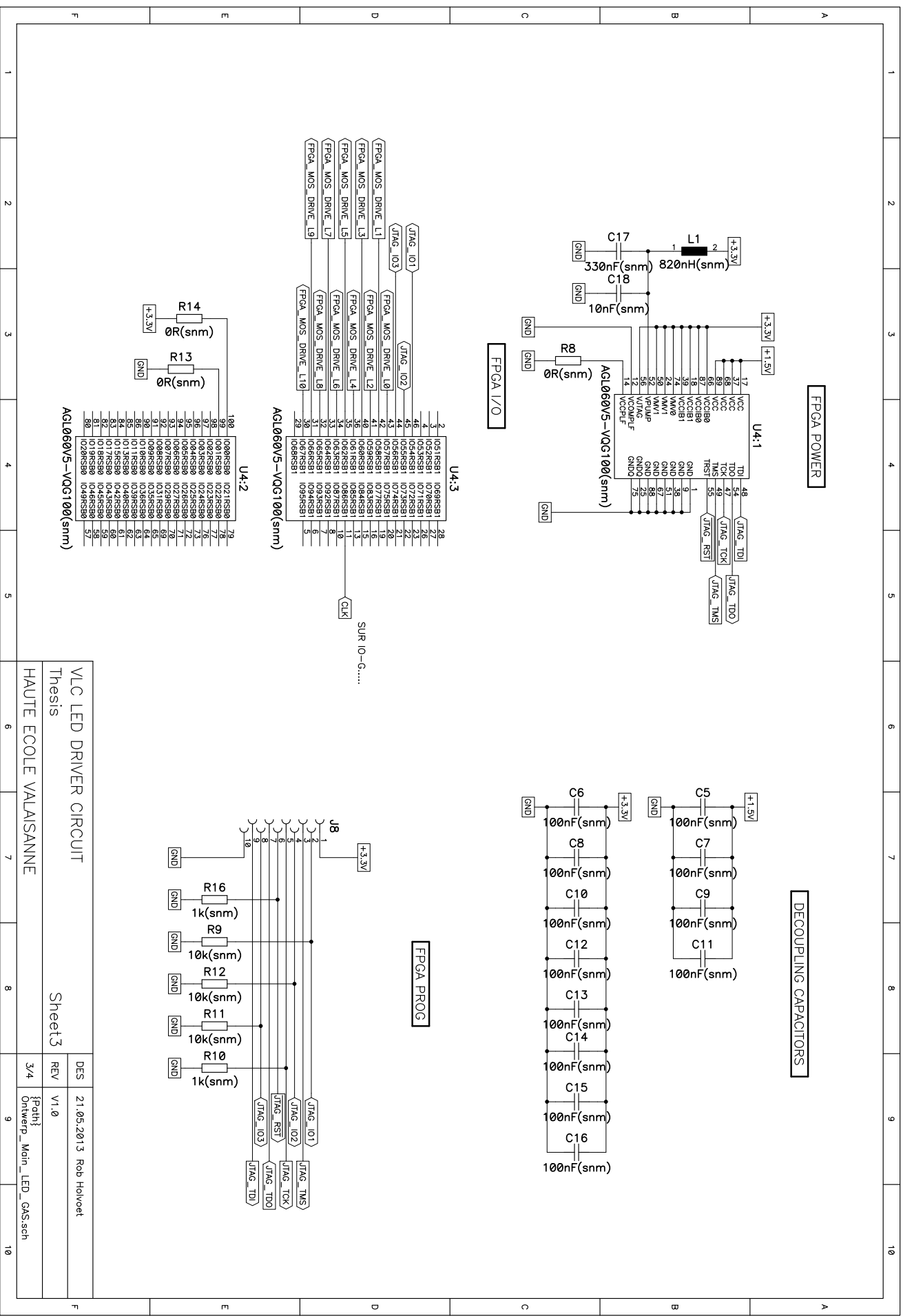
FPGA PROG

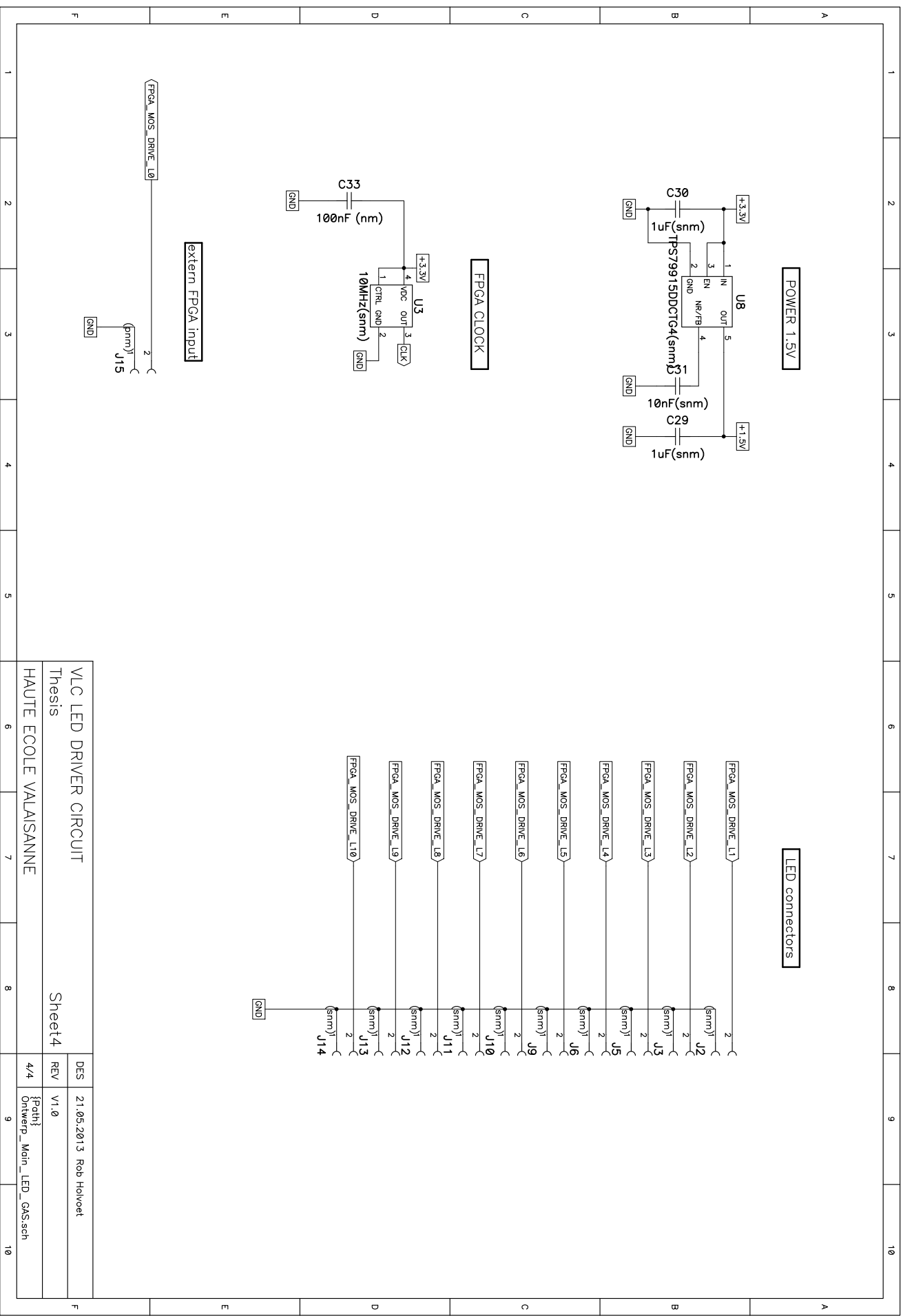


VLC LED DRIVER CIRCUIT
Thesis
HAUTE ECOLE VALAISANNE

Sheet3

DES 21.05.2013 Rob Holvoet
REV V1.0
{path}
Ortwerp_Main_LED_GAS.sch





VLC LED DRIVER CIRCUIT
Thesis

Sheet4

HAUTE ECOLE VALAISANNE

DES	21.05.2013	Rob Holvoet
REV	V1.0	
4/4	{path}	Ortwerp_Main_LED_GAS.sch

Appendix I

Code

I.1 VHDL Code

I.1.1 ADC

ADC controller

```
-- This architecture controls the ADC: it sends the signals  
-- CS_n and SCLK to the ADC and it receives the serial data  
-- back
```

```
LIBRARY ieee;  
USE ieee.numeric_std.all;  
USE ieee.std_logic_1164.all;
```

```
ARCHITECTURE RTL OF adc121S101Controller IS
```

```
    signal clock_i_CS: std_ulogic; -- CS  
    signal logic_voltage : std_ulogic_vector(adcBitNb+3 downto 0);  
    -- 16 bits in 1 conversion (3 leading zeros + 12 bits from  
    -- sample + 1 zero)  
    signal enable : std_ulogic; -- enable if rising edge of clock_div  
    signal ADC_i : std_ulogic_vector(adcBitNb-1 downto 0);  
    -- parallel representation of sample  
    signal voltage_i : natural RANGE 0 TO 4096;  
    -- integer representation of sample for visibility  
    signal counter_dds_rising : unsigned(ddsBitNb-1 downto 0);  
    -- counter we use for DDS  
    signal counter_dds_rising_i : unsigned(ddsBitNb-1 downto 0);  
    signal sclk_i : std_ulogic;  
    signal sclk_i2 : std_ulogic;
```

```

constant cs_high : integer := 6; — determines during
— how many periods of SCLK 'CS' should stay high
constant cs_low : integer : adcBitNb+4; — determines
— during how many periods of SCLK 'CS' should stay low
constant step : unsigned(ddsBitNb-1 downto 0) :=
to_unsigned(341, ddsBitNb);
— step we use for DDS: we have a 10 bit counter (1024
— different values)
— frequency divider by three:  $f = (341/1023) * f(\text{clk}) =$ 
—  $(1/3) * f(\text{clk})$ 

```

BEGIN

```

— increments the step to the counter on the rising edge of
— the clock
— the counter counts: 0,341,682,1023,341,...
— if the counter is 1023, we add 342. Otherwise the
— counter counts: 0,341,682,1023,340,681,...
process (reset, clock)
begin
    if (reset='1') then
        counter_dds_rising <= (others => '0');
    elsif rising_edge(clock) then
        if counter_dds_rising = to_unsigned(1023, ddsBitNb)
        then
            counter_dds_rising <= counter_dds_rising +
            step + 1;
        else
            counter_dds_rising <= counter_dds_rising +
            step;
        end if;
    end if;
end process;

— delay of half a clock period by triggering on the falling edge
— of the clock
process (reset, clock)
begin
    if(reset='1') then
        counter_dds_rising_i <= (others => '0');
    elsif falling_edge(clock) then
        counter_dds_rising_i <= counter_dds_rising;
    end if;
end process;

```

```
sclk_i <= counter_dds_rising(counter_dds_rising 'high) and
counter_dds_rising_i(counter_dds_rising 'high);
-- by taking the MSB of the counter, we can generate a divider by
-- three with a duty cycle of 50 %.

-- enable pulse of 1 clock period
process (reset, clock)
begin
    if(reset='1') then
        sclk_i2 <= '0';
    elsif rising_edge(clock) then
        sclk_i2 <= sclk_i;
    end if;
end process;

enable <= sclk_i and (not sclk_i2);
-- generates a pulse of one clock period on the rising edge of
-- SCLK

-- cs_n
process (reset, clock)
    variable counter_low : integer range 0 to (adcBitNb+4);
    variable counter_high : integer range 0 to 6;
begin
    if (reset='1') then
        clock_i_cs <= '1';
        counter_low := 0;
        counter_high := 0;
    elsif rising_edge(clock) then
        if (enable='1') then
            if (counter_low < cs_low) then
                clock_i_cs <= '0';
                counter_low := counter_low+1;
                counter_high := 0;
            else
                if counter_high < (cs_high-1) then
                    clock_i_cs <= '1';
                    counter_high := counter_high + 1;
                else
                    counter_low := 0;
                end if;
            end if;
        end if;
    end if;
end process;
```

```

        end if;
end process;

CS_n <= clock_i_cs;

-- serial to parallel conversion
process (reset , clock)
    variable SCLK_counter : integer range -1 to (adcBitNb+3);
begin
    if (reset = '1') then
        SCLK_Counter := adcBitNb+3;
        logic_voltage <= (others => '0');
        ADC_i <= (others => '0');
        voltage_i <= 0;
    elsif rising_edge(clock) then
        if (clock_i_CS = '0') then
            if (enable='1') then
                logic_voltage(SCLK_counter) <= SDATA;
                SCLK_Counter := SCLK_Counter-1;
            end if;
        elsif (clock_i_CS = '1') then
            SCLK_Counter := adcBitNb+3;
            ADC_i <= logic_voltage(logic_voltage'high-3
                downto 1);
            -- serial to parallel after each conversion
            voltage_i <= to_integer(unsigned(ADC_i));
            -- display as integer
        end if;
    end if;
end process;

ADC <= ADC_i;
voltage <= voltage_i;
sclk <= sclk_i;

END ARCHITECTURE RTL;

```

I.1.2 Ethernet

Dual port RAM

— This architecture represents a dual port RAM. We write to RAM
 — via port A and read from RAM via port B

```
USE std.textio.all;
```

```
ARCHITECTURE bhv OF bramDualportWritefirst IS
```

```
  — Define ramContent type
```

```
  type ramContentType is array(0 to (2**addressBitNb)-1) of  
    bit_vector(dataBitNb-1 DOWNT0 0);
```

```
  — Declare ramContent signal
```

```
  signal ramContent: ramContentType;
```

```
BEGIN
```

```
  — Port A
```

```
  — We write the samples in RAM via port A
```

```
  process(clockA)
```

```
  begin
```

```
    if clockA'event and clockA='1' then
```

```
      if enA = '1' then
```

```
        if writeEnA = '1' then
```

```
          ramContent(to_integer(unsigned(addressA))) :=  
            to_bitvector(dataInA, '0');
```

```
        end if;
```

```
      end if;
```

```
    end if;
```

```
  end process;
```

```
  — Port B
```

```
  — when we want to send a frame, we read the data from RAM via port B
```

```
  process(clockB)
```

```
  begin
```

```
    if clockB'event and clockB='1' then
```

```
      dataOutB <= to_stdulogicvector(ramContent(to_integer  
        (unsigned(addressB))));
```

```
    end if;
```

```
  end process;
```

```
END ARCHITECTURE bhv;
```

Controller

```

-- This architecture controls the data we put in a frame
-- when we want to send. When it gets a pulse 'sendFrame',
-- the RAM is read and send. It also determines the UDP port,
-- the destination IP address and the destination MAC address.

```

ARCHITECTURE RTL OF udpApplDiscardController IS

```

    signal ramCounter: unsigned(addressB'range);
    signal functionSelected : std_ulogic;

    type discardStateType is (
        idle, waitGrant, send1, send, waitFifo, waitSendEnd);
    signal discardState: discardStateType;
    signal begin_counter: std_ulogic; -- determines
-- where we should start reading (begin or halfway RAM)
    constant halfRam : unsigned(addressB'range) := shift_left
        (resize("1",addressB'length),addressB'length-1);
    constant max : natural := 10000000; -- determines how
-- long the LED blinks when a UDP packet is received

```

BEGIN

```

functionSelected <= '1' when unsigned(udpPortIn) = portId
                    else '0';
isSelected <= functionSelected;
-- When there is a packet for UDP port 'portId', our block is selected.

readEn <= functionSelected;
-- When there is a packet for us, we can read it.

-- When the FPGA receives a valid UDP packet on port 'portId',
-- a led will blink.
process(clock,reset)
    variable ledCounter : natural;
begin
    if reset = '1' then
        ledUdp <= '0';
        ledCounter <= 0;
    elsif rising_edge(clock) then
        if udpDataValid = '1' and ledCounter = 0 and
            udpPortIn = std_ulogic_vector(to_unsigned(portId,
            udpPortIn'length)) then

```

```

        ledUdp <= '1';
        ledCounter := ledCounter + 1;
    elsif ledCounter = max then
        ledUdp <= '0';
        ledCounter := 0;
    elsif ledCounter > 0 then
        ledCounter := ledCounter + 1;
    end if;
end if;
end process;

-- read RAM content for data to send
countRamAddresses: process(reset, clock)
begin
    if reset = '1' then
        ramCounter <= (others => '0');
    elsif rising_edge(clock) then
        if ramCounter = 0 then
            if discardState = send1 then
                ramCounter <= ramCounter + 1;
            end if;
        else
            if ramCounter < frameLength-1 then
                if discardState = send then
                    if txFull = '0' then
                        ramCounter <= ramCounter + 1;
                    else
                        ramCounter <= ramCounter - 1;
                    end if;
                elsif discardState = waitFifo then
                    if txFull = '0' then
                        ramCounter <= ramCounter + 1;
                    end if;
                end if;
            else
                ramCounter <= (others => '0');
            end if;
        end if;
    end if;
end process countRamAddresses;

-- when we want to send a frame, 'begincounter' is inverted.
-- 'begincounter' determines which part of the RAM is send.
process(clock, reset)

```

```
begin
    if reset = '1' then
        begin_counter <= '0';
    elsif rising_edge(clock) then
        if sendFrame = "01" or sendFrame = "10" then
            begin_counter <= not begin_counter;
        end if;
    end if;
end process;
```

— discard FSM

```
discardSequencer: process(reset, clock)
begin
    if reset = '1' then
        discardState <= idle;
    elsif rising_edge(clock) then
        case discardState is
            when idle =>
                if sendFrame = "01" or sendFrame = "10" then
                    discardState <= waitGrant;
                end if;
            when waitGrant =>
                if writeGranted = '1' then
                    discardState <= send1;
                end if;
            when send1 =>
                discardState <= send;
            when send =>
                if ramCounter = 0 then
                    discardState <= waitSendEnd;
                elsif txFull = '1' then
                    discardState <= waitFifo;
                end if;
            when waitFifo =>
                if txFull = '0' then
                    discardState <= send;
                end if;
            when waitSendEnd =>
                if sendFrame = "00" then
                    discardState <= idle;
                end if;
        end case;
    end if;
end process;
```



```

end process discardSequencer;

-- write to UDP
writeRequest <= '1' when
    (discardState = waitGrant) or
    (discardState = send1)      or
    (discardState = send)      or
    (discardState = waitFifo)
else '0';

txWr <= not txFull when discardState = send
else '0';

-- If 'begincounter' = '0', we read from the first part of
-- the RAM. Otherwise, the second part is read.
addressB <= std_ulogic_vector(ramCounter) when begin_counter
    = '0' else std_ulogic_vector(ramCounter+halfRam);

destUdpPort <= std_ulogic_vector(to_unsigned(portId ,
    destUdpPort'length));
udpPortOut <= std_ulogic_vector(to_unsigned(portId ,
    udpPortOut'length));

-- If we send a UDP packet to port 'portId' from the PC, the
-- FPGA will send to the IP address and MAC address of the PC.
process(clock , reset)
begin
    if reset = '1' then
        destIpAddress <= sourceIpAddress;
        destMacAddress <= sourceMacAddress;
    elsif rising_edge(clock) then
        if udpPortIn = std_ulogic_vector(to_unsigned(
            portId , udpPortIn'length)) and udpDataValid = '1'
        then
            destIpAddress <= sourceIpAddress;
            destMacAddress <= sourceMacAddress;
        end if;
    end if;
end process;

-- we bring 'begin_counter' outside , so we can see on the
-- oscilloscope whether the FPGA alternately reads from
-- the first and the second part of RAM (this signal should

```

```

-- be a square wave with a duty cycle of 50 %).
beginc <= begin_counter;

```

```

END ARCHITECTURE RTL;

```

ADC to RAM

```

-- This architecture receives the 12 bit samples from the ADC
-- controller and stores them in RAM

```

```

ARCHITECTURE adc_to_ram OF adc_to_ram IS

```

```

    constant halfRam : unsigned(addressA'range) := shift_left
    (resize("1",addressA'length),addressA'length-1);
    constant fullRam : unsigned(addressA'range) := (others =>
    '0');
    signal cs_n_i : std_ulogic;
    signal cs_n_i2 : std_ulogic;
    signal writeEnA_i : std_logic;
    signal addressA_i : unsigned(addressBitNb-1 downto 0);

```

```

BEGIN

```

```

enA <= '1';

```

```

-- delay 'cs_n' 1 clock period
process(clock,reset)
begin
    if reset='1' then
        cs_n_i <= '0';
    elsif rising_edge(clock) then
        cs_n_i <= cs_n;
    end if;
end process;

```

```

-- delay 'cs_n' another clock period
process(clock,reset)
begin
    if reset='1' then
        cs_n_i2 <= '0';
    elsif rising_edge(clock) then
        cs_n_i2 <= cs_n_i;
    end if;
end process;

```

```

writeEnA_i <= cs_n and not(cs_n_i2);
-- writeEnA_i is high for 2 clock periods,
-- in the first period we write the first byte of the sample,
-- in the second period we write the second byte of the sample.

-- writes samples to RAM
process(clock,reset)
    variable counter : integer;
begin
    if reset='1' then
        addressA_i <= (others => '0');
        dataInA <= (others => '0');
        counter := 0;
    elsif rising_edge(clock) then
        if (writeEnA_i = '1') then
            if counter = 0 then
                dataInA <= ('0','0','0','0') & ADC
                    (ADC'high downto 8);
                -- put 4 zeros before sample to store 16
                -- bits (2 bytes) in RAM
                counter := 1;
                addressA_i <= addressA_i + 1;
            elsif counter = 1 then
                dataInA <= ADC(7 downto 0);
                counter := 0;
                addressA_i <= addressA_i + 1;
            end if;
            if (addressA_i = halfRAM) then
                sendFrame<="01";
                -- when we are halfway the RAM, the content
                -- of the first part is send
            elsif addressA_i = fullRam then
                sendFrame<="10";
                -- when we are at the end of the RAM, the
                -- content of the second part is send
            else
                sendFrame<="00";
            end if;
        end if;
    end if;
end process;

-- control signals to dual port RAM

```

```
addressA <= std_ulogic_vector(addressA_i-1);  
writeEnA <= writeEnA_i;
```

```
END ARCHITECTURE adc_to_ram;
```

I.1.3 Sequence generation for LEDs

```

ARCHITECTURE gen OF bitgen IS
  signal bitled1_sig : std_logic_vector((L-1) downto 0);
  constant L : integer := L_array'length;
BEGIN

P1 : process (reset, clock)
begin
  if(reset = '1') then
    bitled1_sig <= L_array;
  elsif rising_edge(clock) and clock_div100k = '1' then
    --generating max length LFSR's for different register lengths
    if L = 4 then
      -- different LFSR for the different LEDS(number corresponds with led)
      case lednumber is
        when 1 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
          (bitled1_sig(L-1) xor bitled1_sig(L-2));
        when others => bitled1_sig <= (bitled1_sig((L-2) downto 0) &
          bitled1_sig(L-1));
      end case;

    elsif L = 5 then
      -- different LFSR for the different LEDS(number corresponds with led)
      case lednumber is
        when 1 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
          (bitled1_sig(L-1) xor bitled1_sig(L-3));
        when 2 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
          (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
          bitled1_sig(L-3) xor bitled1_sig(L-4));
        when 3 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
          (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
          bitled1_sig(L-3) xor bitled1_sig(L-5));
        when others => bitled1_sig <= (bitled1_sig((L-2) downto 0) &
          bitled1_sig(L-1));
      end case;

    elsif L = 6 then
      -- different LFSR for the different LEDS(number corresponds with led)
      case lednumber is
        when 1 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
          (bitled1_sig(L-1) xor bitled1_sig(L-2));
        when 2 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
          (bitled1_sig(L-1) xor bitled1_sig(L-2) xor

```

```

        bitled1_sig(L-3) xor bitled1_sig(L-6));
    when 3 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
        (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
        bitled1_sig(L-4) xor bitled1_sig(L-5));
    when others => bitled1_sig <= (bitled1_sig((L-2) downto 0) &
        bitled1_sig(L-1));
end case;

elsif L = 7 then
-- different LFSR for the different LEDS(number corresponds with led)
case lednumber is
    when 1 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
        (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
        bitled1_sig(L-3) xor bitled1_sig(L-4));
    when 2 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
        (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
        bitled1_sig(L-3) xor bitled1_sig(L-6));
    when 3 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
        (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
        bitled1_sig(L-4) xor bitled1_sig(L-6));
    when 4 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
        (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
        bitled1_sig(L-4) xor bitled1_sig(L-7));
    when 5 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
        (bitled1_sig(L-1) xor bitled1_sig(L-3) xor
        bitled1_sig(L-4) xor bitled1_sig(L-5));
    when 6 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
        (bitled1_sig(L-1) xor bitled1_sig(L-2));
    when 7 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
        (bitled1_sig(L-1) xor bitled1_sig(L-4));
    when 8 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
        (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
        bitled1_sig(L-3) xor bitled1_sig(L-4) xor
        bitled1_sig(L-5) xor bitled1_sig(L-6));
    when 9 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
        (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
        bitled1_sig(L-3) xor bitled1_sig(L-4) xor
        bitled1_sig(L-6) xor bitled1_sig(L-7));
    when others => bitled1_sig <= (bitled1_sig((L-2) downto 0) &
        bitled1_sig(L-1));
end case;

elsif L = 8 then
-- different LFSR for the different LEDS(number corresponds with led)

```

```

case lednumber is
  when 1 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
    (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
     bitled1_sig(L-3) xor bitled1_sig(L-8));
  when 2 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
    (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
     bitled1_sig(L-4) xor bitled1_sig(L-6));
  when 3 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
    (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
     bitled1_sig(L-6) xor bitled1_sig(L-7));
  when 4 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
    (bitled1_sig(L-1) xor bitled1_sig(L-3) xor
     bitled1_sig(L-4) xor bitled1_sig(L-5));
  when 5 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
    (bitled1_sig(L-1) xor bitled1_sig(L-3) xor
     bitled1_sig(L-4) xor bitled1_sig(L-6));
  when 6 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
    (bitled1_sig(L-1) xor bitled1_sig(L-3) xor
     bitled1_sig(L-4) xor bitled1_sig(L-7));
  when 7 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
    (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
     bitled1_sig(L-3) xor bitled1_sig(L-4) xor
     bitled1_sig(L-5) xor bitled1_sig(L-7));
  when 8 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
    (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
     bitled1_sig(L-3) xor bitled1_sig(L-4) xor
     bitled1_sig(L-7) xor bitled1_sig(L-8));
  when others => bitled1_sig <= (bitled1_sig((L-2) downto 0) &
    bitled1_sig(L-1));
end case;
elsif L = 9 then
-- different LFSR for the different LEDS(number corresponds with led)
case lednumber is
  when 1 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
    (bitled1_sig(L-1) xor bitled1_sig(L-5));
  when 2 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
    (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
     bitled1_sig(L-3) xor bitled1_sig(L-8));
  when 3 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
    (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
     bitled1_sig(L-4) xor bitled1_sig(L-5));
  when 4 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
    (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
     bitled1_sig(L-5) xor bitled1_sig(L-6));

```

```

when 5 =>    bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-5) xor bitled1_sig(L-9));
when 6 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-6) xor bitled1_sig(L-8));
when 7 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-3) xor
bitled1_sig(L-4) xor bitled1_sig(L-6));
when 8 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-3) xor
bitled1_sig(L-5) xor bitled1_sig(L-8));
when 9 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-4) xor
bitled1_sig(L-5) xor bitled1_sig(L-7));
when 10 =>    bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-3) xor bitled1_sig(L-4) xor
bitled1_sig(L-5) xor bitled1_sig(L-7));
when 11 =>    bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-3) xor bitled1_sig(L-4) xor
bitled1_sig(L-5) xor bitled1_sig(L-9));
when others => bitled1_sig <= (bitled1_sig((L-2) downto 0) &
bitled1_sig(L-1));
end case;

elsif L = 10 then
-- different LFSR for the different LEDS(number corresponds with led)
case lednumber is
when 1 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-4));
when 2 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-3) xor bitled1_sig(L-6));
when 3 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-4) xor bitled1_sig(L-5));
when 4 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-4) xor bitled1_sig(L-8));
when 5 =>    bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-5) xor bitled1_sig(L-10));

```



```

when 6 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-6) xor bitled1_sig(L-9));
when 7 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-7) xor bitled1_sig(L-9));
when 8 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-3) xor
bitled1_sig(L-4) xor bitled1_sig(L-6));
when 9 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-3) xor
bitled1_sig(L-4) xor bitled1_sig(L-9));
when 10 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-3) xor
bitled1_sig(L-6) xor bitled1_sig(L-7));
when 11 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-3) xor
bitled1_sig(L-7) xor bitled1_sig(L-8));
when others => bitled1_sig <= (bitled1_sig((L-2) downto 0) &
bitled1_sig(L-1));
end case;
elsif L = 11 then
-- different LFSR for the different LEDS(number corresponds with led)
case lednumber is
when 1 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-3));
when 2 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-3) xor bitled1_sig(L-5));
when 3 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-3) xor bitled1_sig(L-7));
when 4 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-3) xor bitled1_sig(L-10));
when 5 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-4) xor bitled1_sig(L-6));
when 6 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-4) xor bitled1_sig(L-11));
when 7 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-5) xor bitled1_sig(L-9));

```

```

when 8 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-5) xor bitled1_sig(L-10));
when 9 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-6) xor bitled1_sig(L-7));
when 10 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-8) xor bitled1_sig(L-9));
when 11 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-9) xor bitled1_sig(L-10));
when others => bitled1_sig <= (bitled1_sig((L-2) downto 0) &
bitled1_sig(L-1));
end case;
elsif L = 12 then
— different LFSR for the different LEDS(number corresponds with led)
case lednumber is
when 1 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-3) xor bitled1_sig(L-9));
when 2 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-3) xor bitled1_sig(L-11));
when 3 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-5) xor bitled1_sig(L-7));
when 4 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-2) xor
bitled1_sig(L-6) xor bitled1_sig(L-9));
when 5 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-3) xor
bitled1_sig(L-4) xor bitled1_sig(L-10));
when 6 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-3) xor
bitled1_sig(L-8) xor bitled1_sig(L-9));
when 7 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-4) xor
bitled1_sig(L-5) xor bitled1_sig(L-8));
when 8 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-4) xor
bitled1_sig(L-6) xor bitled1_sig(L-7));
when 9 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
(bitled1_sig(L-1) xor bitled1_sig(L-5) xor

```

```

        bitled1_sig(L-7) xor bitled1_sig(L-8));
    when 10 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
        (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
        bitled1_sig(L-8) xor bitled1_sig(L-9));
    when 11 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
        (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
        bitled1_sig(L-9) xor bitled1_sig(L-10));
    when others => bitled1_sig <= (bitled1_sig((L-2) downto 0) &
        bitled1_sig(L-1));
    end case;
elseif L = 13 then
-- different LFSR for the different LEDS(number corresponds with led)
    case lednumber is
        when 1 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
            (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
            bitled1_sig(L-3) xor bitled1_sig(L-6));
        when others => bitled1_sig <= (bitled1_sig((L-2) downto 0) &
            bitled1_sig(L-1));
    end case;
elseif L = 14 then
-- different LFSR for the different LEDS(number corresponds with led)
    case lednumber is
        when 1 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
            (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
            bitled1_sig(L-3) xor bitled1_sig(L-13));
        when others => bitled1_sig <= (bitled1_sig((L-2) downto 0) &
            bitled1_sig(L-1));
    end case;
elseif L = 15 then
-- different LFSR for the different LEDS(number corresponds with led)
    case lednumber is
        when 1 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
            (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
            bitled1_sig(L-3) xor bitled1_sig(L-5));
        when others => bitled1_sig <= (bitled1_sig((L-2) downto 0) &
            bitled1_sig(L-1));
    end case;
elseif L = 16 then
-- different LFSR for the different LEDS(number corresponds with led)
    case lednumber is
        when 1 => bitled1_sig <= bitled1_sig((L-2) downto 0) &
            (bitled1_sig(L-1) xor bitled1_sig(L-2) xor
            bitled1_sig(L-4) xor bitled1_sig(L-13));
        when others => bitled1_sig <= (bitled1_sig((L-2) downto 0) &

```

```
        bitled1_sig(L-1));
    end case;
    end if;
end if;
end if;
end process P1;

bitled1 <= bitled1_sig(L-1);
END ARCHITECTURE gen;
```

I.2 Python Code

I.2.1 Ethernet sender

```
# This Python script sends a UDP packet to the FPGA.
```

```
import socket

UDP_IP = '169.254.111.1' # IP address FPGA
UDP_PORT = 9 # destination UDP port

HOST = socket.gethostname(socket.gethostname()) # get host
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((HOST, 0)) # bind socket to host, choose source UDP port (0)

print "HOST:", HOST

data = "Hello_world_VLC_Communication_Correlation_filter_Python_Ethernet,\
_test_message_to_send_to_FPGA"
# data to send
sock.sendto(data, (UDP_IP, UDP_PORT)) # send to destination
print "send_done"

sock.close() # close socket
```

I.2.2 Main script

```
# This Python script receives UDP packets from the FPGA. These packets
# contain the samples taken by the ADC. The samples are stored in an
# array and we do some calculations to print the samples in a graph
# as seen on the oscilloscope. Before receiving the data, we first send
# a packet to the FPGA, so he knows our MAC address and IP address.
# We also filter the received data with each of the bit sequences. After
# this correlation, we use some code to find the position. Finally, we
# plot the position in a graph.
# We import the file 'functions.py' where we defined the functions we
# need.
# We also import the file 'functions_test.py' where we defined some
# functions for testing purposes.

import functions
import functions_test
import matplotlib.pyplot as plt
from scipy.signal import lfilter
```

```
import numpy as np

UDP_IP = '169.254.111.1' # IP address FPGA
UDP_PORT = 9 # destination UDP port

functions.sendUdp(UDP_IP,UDP_PORT) # send UDP packet to FPGA

UDP_IP = '' # get packets from all IP addresses
UDP_PORT = 9 # bind to UDP port 9

sequence = [0,1,0,0,1,0,1,1,0,1] # 10 bit start sequence of the LFSRs

sequence1_gen = functions.generate_seq1(sequence) # sequence of 1023 bits
sequence2_gen = functions.generate_seq2(sequence) # sequence of 1023 bits
sequence3_gen = functions.generate_seq3(sequence) # sequence of 1023 bits
sequence4_gen = functions.generate_seq4(sequence) # sequence of 1023 bits
sequence5_gen = functions.generate_seq5(sequence) # sequence of 1023 bits
# calculate sequences from start sequence
# bit sequences sent by the LEDs (time between two bits is 5 us)

samplesPerBit = 5 # how many samples we take for 1 bit

sequence1_samples = functions.sequence_gen(sequence1_gen, samplesPerBit)
sequence2_samples = functions.sequence_gen(sequence2_gen, samplesPerBit)
sequence3_samples = functions.sequence_gen(sequence3_gen, samplesPerBit)
sequence4_samples = functions.sequence_gen(sequence4_gen, samplesPerBit)
sequence5_samples = functions.sequence_gen(sequence5_gen, samplesPerBit)
# generates new bit sequences so the time between two bits is 1 us
# instead of 5 us, because time between ADC samples is also 1 us

packets = 100
# determines on how many data (packets) we want to do the calculations

numberOfCalculations = 1
# how many times we want to run the while loop

while numberOfCalculations >= 1:
    numberOfCalculations = numberOfCalculations - 1
    array_packets = functions.receiveUdp(packets,UDP_IP,UDP_PORT)
    # this function receives n packets and gives the received data back
    # in a list

    array_analog, array_real = functions.bit_to_real_and_osc_val \
    (array_packets)
```

```
# this function calculates the samples (2 bytes) and calculates the
# corresponding analog values as seen on the oscilloscope.

array_peaksout = functions.getpeaksout(array_analog,0.07)
# this function filters out the unwanted peaks

time=functions.timegen(len(array_real),1e6)
# determine time between samples: samplefrequency = 1 MHz

# plot data vs. time (as seen on the oscilloscope)
#functions.grafosc(time,array_peaksout)

# we used this function to see what happens if we filter the bit
# sequence on itself (auto correlation)
#functions_test.autocorr(sequence1_samples)

# this function plots the received signal and the ideal signal under
# each other so we can see an eventual time shift
#functions_test.received_vs_ideal(array_peaksout, sequence1_samples)

# correlation filters
y1=lfilter(sequence1_samples,1,array_peaksout)
y2=lfilter(sequence2_samples,1,array_peaksout)
y3=lfilter(sequence3_samples,1,array_peaksout)
y4=lfilter(sequence4_samples,1,array_peaksout)
y5=lfilter(sequence5_samples,1,array_peaksout)

# plot filtered data vs. time
# functions.grafcorr(time,y1,1)
# functions.grafcorr(time,y2,2)
# functions.grafcorr(time,y3,3)
# functions.grafcorr(time,y4,4)
# functions.grafcorr(time,y5,5)

# calculate the mean of the peaks of the filtered signal
corr1 = functions.mean_peaks_after_correlation(y1,samplesPerBit, \
len(sequence1_gen))
corr2 = functions.mean_peaks_after_correlation(y2,samplesPerBit, \
len(sequence2_gen))
corr3 = functions.mean_peaks_after_correlation(y3,samplesPerBit, \
len(sequence3_gen))
corr4 = functions.mean_peaks_after_correlation(y4,samplesPerBit, \
len(sequence4_gen))
corr5 = functions.mean_peaks_after_correlation(y5,samplesPerBit, \
```

```
len(sequence5_gen))

print "Correlatie_1:", corr1
print "Correlatie_2:", corr2
print "Correlatie_3:", corr3
print "Correlatie_4:", corr4
print "Correlatie_5:", corr5

coeff1 = [5.668, -0.05907, 0.0003756, -1.253e-6, 2.028e-9, -1.268e-12]
coeff2 = [4.173, -0.0234, 9.693e-5, -2.265e-7, 2.575e-10, -1.125e-13]
coeff3 = [5.024, -0.04922, 0.0003086, -1.015e-6, 1.607e-9, -9.747e-13]
coeff4 = [4.379, -0.02752, 0.0001286, -3.525e-7, 4.842e-10, -2.605e-13]
coeff5 = [4.706, -0.03478, 0.0001661, -4.151e-7, 4.93e-10, -2.238e-13]
# coefficients derived from the fitting with a 5th order polynom on the
# measurements (correlation value versus radius)

radius1 = functions.getRadius(corr1,coeff1)
radius2 = functions.getRadius(corr2,coeff2)
radius3 = functions.getRadius(corr3,coeff3)
radius4 = functions.getRadius(corr4,coeff4)
radius5 = functions.getRadius(corr5,coeff5)
# find radiuses from each of the 5 LEDs starting from the correlation
# value and the coefficients of the 5th order polynom

radius = [radius1] + [radius2] + [radius3] + [radius4] + [radius5]
# put the 5 radiuses in a list

led = np.array([[2,2.4],[2,4.4],[3.5,3.4],[5,4.4],[5,2.4]])
# gives the position of the 5 LEDs in x-y coordinates (in meters)

functions.displd(led, radius)
# shows the position of the 5 LEDs and shows circles around the LEDs
# with the calculated radiuses

position = functions.middlePoint(radius,led)
# calculates the position of the receiver starting from the
# intersections of the different circles

plt.show()
# show all the plots

print "done"
```


I.2.3 Functions

```
# In this file we define all the functions we use in the main script to
# make our code more readable.

import matplotlib.pyplot as plt
import matplotlib.patches as patch
import numpy as np
import socket
import math

def sendUdp(IPaddress , UDPport):
    UDP_IP = IPaddress # IP address FPGA
    UDP_PORT = UDPport # destination UDP port

    HOST = socket.gethostbyname(socket.gethostname()) # get host
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((HOST, 0)) # bind socket to host, choose source UDP port (0)

    print "HOST:", HOST

    data = "Hello_world_VLC_Communication_Correlation_filter_Python_\
    \_\_\_\_Ethernet_,_test_message_to_send_to_FPGA"

    # data to send
    sock.sendto(data , (UDP_IP, UDP_PORT)) # send to destination
    print "send_done"

    sock.close() # close socket

def receiveUdp(packets , UDP_IP , UDP_PORT):
    sock = socket.socket(socket.AF_INET, # Internet
                        socket.SOCK_DGRAM) # UDP (datagram)
    sock.bind((UDP_IP, UDP_PORT)) # bind socket to right interface

    array = bytearray(1024) # bytearray to store the samples

    array_packets = [0] * packets * 1024
    # list where we put all the data

    # receive packets and store data in a list
    for x in range(0, packets):
        print "wait_message"
        data , addr = sock.recvfrom_into(array, 1024) # buffer size is
```

```
# 1024 bytes
print "sender:", addr # print source IP address and source
# UDP port
array_packets[x*1024: 1024*(x+1)] = array

sock.close() # close socket

return array_packets

def grafosc(x,y) :
    x = x[0:len(y)]
    plt.figure(1,figsize=(20,10))
    plt.plot(x,y)
    plt.title('Signal_receiver')
    plt.xlabel('time_(s)')
    plt.ylabel('signal_(V)')
    plt.grid()
# plt.savefig('7bit1_1led.png')

def grafcorr(x,y,nr) :
    x = x[0:len(y)]
    plt.figure(figsize=(20,10))
    plt.plot(x,y)
    plt.title('Signal_after_correlation_filter_' + str(nr))
    plt.xlabel('time_(s)')
    plt.ylabel('signal_(V)')
    plt.grid()
# plt.savefig('7bit1_1led_corr.png')

def timegen(length , samplefreq):
    time = map(float , [0] * length)
    time[0] = 0.0

    # time between samples : 1/samplefreq
    for x in range(0,(length-1)):
        time[x+1] = time[x] + (1/samplefreq)
    return time

def bit_to_real_and_osc_val(bitarray):
    array2 = [0] * len(bitarray)

    for x in range(0, len(bitarray)):
        array2[x] = bitarray[x] # convert bytearray to list of integers
```

```

array_samples = [0] * (len(bitarray)/2)

for x in range(0,(len(bitarray)/2)):
    array_samples[x] = array2[2*x]*pow(16,2) + bitarray[2*x+1]
    # calculate the samples (2 bytes) and store them in another list

array_real = map(float , array_samples) # make real
array_analog = array_real

for x in range(0,len(bitarray)/2):
    array_analog[x] = (array_real[x]/4095)*3.3
    # calculate the corresponding analog values as seen on the
    # oscilloscope

return array_analog , array_real

def sequence_gen(sequence , samplesPerBit):
    counter = 0
    counter2 = 0
    # 5 samples per bit
    sequence1_samples = [0] * samplesPerBit * len(sequence)
    for x in range (0, samplesPerBit * len(sequence)):
        sequence1_samples[x] = sequence[counter2]
        counter = counter + 1
        if counter == samplesPerBit:
            counter2 = counter2 + 1
            counter = 0

    return sequence1_samples

def getpeaksout(peakarray , diffval):
    for x in range(1,len(peakarray)-1):
        diff1=peakarray[x] - peakarray[x-1]
        diff2=peakarray[x] - peakarray[x+1]
        if (abs(diff1) > diffval) and (abs(diff2) > diffval) and \
            abs(diff2 + diff1) > (2*diffval) :
            if abs(peakarray[x] - peakarray[x-1]) > \
                abs(peakarray[x] - peakarray[x+1]):
                peakarray[x] = peakarray[x+1]
            else:
                peakarray[x] = peakarray[x-1]

mean_array = np.mean(peakarray)
#calculate mean of samples

```

```
for x in range(0, len(peakarray)):
    peakarray[x] = peakarray[x] - mean_array
    # subtract mean from each sample, so we have a signal around 0 volt.

return peakarray

def generate_seq1(sequence):
    length = len(sequence)
    range_lfsr = (pow(2, len(sequence)) - 1)
    sequence_nbit = [0] * range_lfsr
    # with an 8 bit LFSR, we can have maximum a sequence of 255 bits
    begin_seq = sequence
    # store start sequence
    for x in range(0, range_lfsr):
        sequence_nbit[x] = sequence[0] # first bit of LFSR is used to
        # drive the LED
        if length == 4: # sequences for 4-bit to 15-bit LFSR
            sequence = sequence[1: len(sequence)] + [(sequence[0]^ sequence[1])]
        elif length == 5:
            sequence = sequence[1: len(sequence)] + [(sequence[0]^ sequence[2])]
        elif length == 6:
            sequence = sequence[1: len(sequence)] + [(sequence[0]^ sequence[1])]
        elif length == 7:
            sequence = sequence[1: len(sequence)] + [(sequence[0]^ sequence[1] \
                ^ sequence[2] ^ sequence[3])]
        elif length == 8:
            sequence = sequence[1: len(sequence)] + [(sequence[0]^ sequence[1] \
                ^ sequence[2] ^ sequence[7])]
        elif length == 9:
            sequence = sequence[1: len(sequence)] + [(sequence[0]^ sequence[4])]
        elif length == 10:
            sequence = sequence[1: len(sequence)] + [(sequence[0]^ sequence[3])]
        elif length == 11:
            sequence = sequence[1: len(sequence)] + [(sequence[0]^ sequence[2])]
        elif length == 12:
            sequence = sequence[1: len(sequence)] + [(sequence[0]^ sequence[1] \
                ^ sequence[2] ^ sequence[8])]
        elif length == 13:
            sequence = sequence[1: len(sequence)] + [(sequence[0]^ sequence[1] \
                ^ sequence[2] ^ sequence[5])]
        elif length == 14:
            sequence = sequence[1: len(sequence)] + [(sequence[0]^ sequence[1] \
                ^ sequence[2] ^ sequence[12])]
```

```

    elif length == 15:
        sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
        ^ sequence[2] ^ sequence[4])]
    # calculate next value of the LFSR ('^' represents an xor function)
    # for an n bit LFSR
    if sequence == begin_seq:
        break # if we are back at the begin sequence, we jump out
        # the loop
sequence = sequence_nbit[0:x+1]
sequence.reverse() # flips the bit sequence for the correlation filter

return sequence # return bit sequence

def generate_seq2(sequence):
    length = len(sequence)
    range_lfsr = (pow(2,len(sequence))-1)
    sequence_nbit = [0] * range_lfsr
    begin_seq = sequence

for x in range(0,range_lfsr):
    sequence_nbit[x] = sequence[0]
    if length == 4:
        sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[2])]
    elif length == 5:
        sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
        ^ sequence[2] ^ sequence[3])]
    elif length == 6:
        sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
        ^ sequence[2] ^ sequence[5])]
    elif length == 7:
        sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
        ^ sequence[2] ^ sequence[5])]
    elif length == 8:
        sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
        ^ sequence[3] ^ sequence[5])]
    elif length == 9:
        sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
        ^ sequence[2] ^ sequence[7])]
    elif length == 10:
        sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
        ^ sequence[2] ^ sequence[5])]
    elif length == 11:
        sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
        ^ sequence[2] ^ sequence[4])]

```

```

elif length == 12:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[2] ^ sequence[10])]
elif length == 13:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[2] ^ sequence[5])]
elif length == 14:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[2] ^ sequence[12])]
elif length == 15:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[2] ^ sequence[4])]
if sequence == begin_seq:
    break
sequence = sequence_nbit[0:x+1]
sequence.reverse()

return sequence

def generate_seq3(sequence):
    length = len(sequence)
    range_lfsr = (pow(2, len(sequence))-1)
    sequence_nbit = [0] * range_lfsr
    begin_seq = sequence
    for x in range(0, range_lfsr):
        sequence_nbit[x] = sequence[0]
        if length == 4:
            sequence=sequence[1:len(sequence)]+[(sequence[1]^sequence[3])]
        elif length == 5:
            sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
            ^ sequence[2] ^ sequence[4])]
        elif length == 6:
            sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
            ^ sequence[3] ^ sequence[4])]
        elif length == 7:
            sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
            ^ sequence[3] ^ sequence[5])]
        elif length == 8:
            sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
            ^ sequence[5] ^ sequence[6])]
        elif length == 9:
            sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
            ^ sequence[3] ^ sequence[4])]
        elif length == 10:

```

```

        sequence=sequence [1: len ( sequence )]+[( sequence [0]^ sequence [1] \
        ^ sequence [3] ^ sequence [4])]
elif length == 11:
    sequence=sequence [1: len ( sequence )]+[( sequence [0]^ sequence [1] \
    ^ sequence [2] ^ sequence [6])]
elif length == 12:
    sequence=sequence [1: len ( sequence )]+[( sequence [0]^ sequence [1] \
    ^ sequence [4] ^ sequence [6])]
elif length == 13:
    sequence=sequence [1: len ( sequence )]+[( sequence [0]^ sequence [1] \
    ^ sequence [2] ^ sequence [5])]
elif length == 14:
    sequence=sequence [1: len ( sequence )]+[( sequence [0]^ sequence [1] \
    ^ sequence [2] ^ sequence [12])]
elif length == 15:
    sequence=sequence [1: len ( sequence )]+[( sequence [0]^ sequence [1] \
    ^ sequence [2] ^ sequence [4])]
if sequence == begin_seq:
    break
sequence = sequence_nbit [0: x+1]
sequence.reverse ()

return sequence

def generate_seq4 ( sequence ):
    length = len ( sequence )
    range_lfsr = ( pow ( 2, len ( sequence ) ) - 1 )
    sequence_nbit = [ 0 ] * range_lfsr
    begin_seq = sequence
    for x in range ( 0, range_lfsr ):
        sequence_nbit [ x ] = sequence [ 0 ]
        if length == 4:
            sequence=sequence [1: len ( sequence )]+[( sequence [1]^ sequence [2])]
        elif length == 5:
            sequence=sequence [1: len ( sequence )]+[( sequence [1]^ sequence [2] \
            ^ sequence [3] ^ sequence [4])]
        elif length == 6:
            sequence=sequence [1: len ( sequence )]+[( sequence [1]^ sequence [2] \
            ^ sequence [3] ^ sequence [4])]
        elif length == 7:
            sequence=sequence [1: len ( sequence )]+[( sequence [0]^ sequence [1] \
            ^ sequence [3] ^ sequence [6])]
        elif length == 8:
            sequence=sequence [1: len ( sequence )]+[( sequence [0]^ sequence [2] \

```

```

        ^ sequence[3] ^ sequence[4]])
elif length == 9:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[4] ^ sequence[5])]
elif length == 10:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[3] ^ sequence[7])]
elif length == 11:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[2] ^ sequence[9])]
elif length == 12:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[5] ^ sequence[8])]
elif length == 13:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[2] ^ sequence[5])]
elif length == 14:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[2] ^ sequence[12])]
elif length == 15:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[2] ^ sequence[4])]
if sequence == begin_seq:
    break
sequence = sequence_nbit[0:x+1]
sequence.reverse()

return sequence

def generate_seq5(sequence):
    length = len(sequence)
    range_lfsr = (pow(2,len(sequence))-1)
    sequence_nbit = [0] * range_lfsr
    begin_seq = sequence
    for x in range(0,range_lfsr):
        sequence_nbit[x] = sequence[0]
        if length == 4:
            sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[3])]
        elif length == 5:
            sequence=sequence[1:len(sequence)]+[(sequence[1]^sequence[2])]
        elif length == 6:
            sequence=sequence[1:len(sequence)]+[(sequence[1]^sequence[2])]
        elif length == 7:
            sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[2] \

```



```

        ^ sequence[3] ^ sequence[4]])
elif length == 8:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[2] \
    ^ sequence[3] ^ sequence[5])]
elif length == 9:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[4] ^ sequence[8])]
elif length == 10:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[4] ^ sequence[9])]
elif length == 11:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[3] ^ sequence[5])]
elif length == 12:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[2] \
    ^ sequence[3] ^ sequence[9])]
elif length == 13:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[2] ^ sequence[5])]
elif length == 14:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[2] ^ sequence[12])]
elif length == 15:
    sequence=sequence[1:len(sequence)]+[(sequence[0]^sequence[1] \
    ^ sequence[2] ^ sequence[4])]
if sequence == begin_seq:
    break
sequence = sequence_nbit[0:x+1]
sequence.reverse()

return sequence

```

```

def mean_peaks_after_correlation(filtered, samplesPerBit, lengthSequence):
    count_samples = len(filtered) # number of samples
    count_peaks = count_samples/samplesPerBit/lengthSequence
    # calculate number of peaks in filtered signal
    peaks = [0] * count_peaks
    # list where we can put the value of the peaks
    step = count_samples/count_peaks # decide the length of the intervals
    for x in range(0, count_peaks):
        peaks[x] = max(filtered[x*step:(x+1)*step])
        # in each interval, we look for the maximum value and store it in
        # a list

```

```

    return np.mean(peaks[1:len(peaks)])
    # calculate mean of all the peaks

# radius is a list of the different found radiuses from the LEDs
# led is a n*2 matrix with the positions of the LEDs
def middlePoint(radius ,led ):
    rangeLeds = 2.5
    counterLeds = 0
    # we only do the calculations with the LEDs closer than 2.5m (most
    # accurate). If we don't have 3 leds to do the calculations with the
    # limit goes up with steps of 0.1m until we have 3 LEDs to do the
    # calculations with
    for x in range(0,len(radius)):
        if radius[x] < rangeLeds:
            counterLeds = counterLeds + 1
    while counterLeds < 3:
        rangeLeds = rangeLeds + 0.1
        counterLeds = 0
        for x in range(0,len(radius)):
            if radius[x] < rangeLeds:
                counterLeds = counterLeds + 1

    coordX=0
    coordY=0
    Point = [coordX ,coordY]
    counter=0
    N = [0] * int(math.pow(len(radius),2))
    A = [0] * int(math.pow(len(radius),2))
    B = [0] * int(math.pow(len(radius),2))
    C = [0] * int(math.pow(len(radius),2))
    delta = [0] * int(math.pow(len(radius),2))
    point = np.zeros((int(math.pow(len(radius),2)),2))
    #calculation of the coefficients for the calculation
    #of the intersection points and calculation of those points
    for k in range(0,len(radius)):
        for j in range(0,len(radius)):
            if j > k:
                x0 = led[k,0]
                x1 = led[j,0]
                y0 = led[k,1]
                y1 = led[j,1]
                r0 = radius[k]
                r1 = radius[j]
                if r0 < rangeLeds and r1 < rangeLeds :

```

```

if not ((x0-x1) == 0) and ((y0-y1) == 0):
    if (y0-y1) == 0:
        x=(math.pow(r1,2)-math.pow(r0,2)-math.pow(x1,2)+ \
            math.pow(x0,2))/(2*(x0-x1))
        N[counter] = 0
        A[counter] = 1
        B[counter] = -2*y1
        C[counter] = math.pow(x1,2)+math.pow(x,2)-2*x1*x+ \
            math.pow(y1,2)-math.pow(r1,2)
        delta[counter]=math.sqrt(abs(math.pow(B[counter],2)\
            -4*A[counter]*C[counter]))
        point[counter,0] = x
        point[counter,1] = (-B[counter]+delta[counter]) \
            /(2*A[counter])
        counter = counter+1
        N[counter] = 0
        A[counter] = A[counter-1]
        B[counter] = B[counter-1]
        C[counter] = C[counter-1]
        delta[counter] = delta[counter-1]
        point[counter,0] = x
        point[counter,1] = (-B[counter]-delta[counter]) \
            /(2*A[counter])
    else:
        N[counter] = (math.pow(r1,2)-math.pow(r0,2)- \
            math.pow(x1,2)+math.pow(x0,2)-math.pow(y1,2)+ \
            math.pow(y0,2))/(2*(y0-y1))
        A[counter] = math.pow(((x0-x1)/(y0-y1)),2)+1
        B[counter]=2*y0*(x0-x1)/(y0-y1)-2*N[counter]*(x0-x1)\
            /(y0-y1)-2*x0
        C[counter] = math.pow(x0,2)+math.pow(y0,2)+ \
            math.pow(N[counter],2)-math.pow(r0,2)-2*y0*N[counter]
        delta[counter]=math.sqrt(abs(math.pow(B[counter],2)\
            -4*A[counter]*C[counter]))
        point[counter,0] = (-B[counter]+delta[counter]) \
            /(2*A[counter])
        point[counter,1] = N[counter]-point[counter,0] \
            *(x0-x1)/(y0-y1)
        counter = counter+1
        N[counter] = N[counter-1]
        A[counter] = A[counter-1]
        B[counter] = B[counter-1]
        C[counter] = C[counter-1]
        delta[counter] = delta[counter-1]

```

```

        point[counter,0] = (-B[counter]-delta[counter]) \
            /(2*A[counter])
        point[counter,1] = N[counter]-point[counter,0] \
            *(x0-x1)/(y0-y1)
        counter = counter + 1
#get rid of the zeros in the array point
point = np.array(point)[0:counter]
#start calculation of midpoint from the calculated points

plt.plot(point[:,0],point[:,1], 'v',color='green')
deltaPoint = [0] * counter
middlePoints = np.zeros((int(counter/2),2))
#accumulation of the distances from 1 point to the others
for k in range(0,counter):
    deltaPoint[k] = 0
    for j in range(0,counter):
        deltaPoint[k] = deltaPoint[k]+math.sqrt(math.pow((point[k,0] \
            -point[j,0]),2)+math.pow((point[k,1]-point[j,1]),2))
# we only need the points with the smallest distances to be used for
# the calculations so we sort them and look with what points the are
# related
deltaPointSort = sorted(deltaPoint)

for k in range(0,counter/2):
    for j in range(0, counter):
        if deltaPointSort[k] == deltaPoint[j]:
            middlePoints[k,0] = point[j,0]
            middlePoints[k,1] = point[j,1]

plt.plot(middlePoints[:,0],middlePoints[:,1], 'v',color='orange')

# calculation of the geometric average
numberMPoint =0
for k in range(0,counter/2):
    if not ((middlePoints[k,0]+middlePoints[k,1]) == 0 ):
        numberMPoint = numberMPoint + 1
        coordX = coordX+ middlePoints[k,0]
        coordY = coordY+ middlePoints[k,1]

Point = np.array([coordX,coordY])/numberMPoint

plt.plot(Point[0],Point[1], "ro",ms=10,mfc="r",mew=2, mec="r")
text = "Position: ◡(" + str(round(Point[0],2)) + ", ◡" + str(round( \
Point[1],2)) + ")"

```

```
plt.text(7.5,1,text,fontsize=18) # plot coordinates of the position
# on the graph
plt.text(7,4,"windows",rotation=-90,fontsize=16)
# indicates where the windows are in the room
plt.hold(False)

return Point

def getRadius(corr, coeff):

    # calculates radius from a certain LED starting from the fifth order
    # polynom
    radius = coeff[0] + coeff[1] * corr + coeff[2] * \
    math.pow(corr,2) + coeff[3] * math.pow(corr,3) \
    + coeff[4] * math.pow(corr,4) + coeff[5] * math.pow(corr,5)

    return radius

def dispLED(led, radius):
    ledX = led[:,0]
    ledY = led[:,1]
    numberpoint = 300;
    coordX = [0] * numberpoint * 5
    coordY = [0] * numberpoint * 5
    #compute circle around each led
    pos = 0;
    for k in range (0,5):
        r = radius[k]
        inc = 2*math.pi / numberpoint
        for j in range(1,numberpoint):
            pos=pos+1
            alpha = j*inc
            coordY[pos] = ledY[k]+math.sin(alpha)*r
            coordX[pos] = ledX[k]+math.cos(alpha)*r

    fig=plt.figure(figsize=(20,10))
    ax=fig.add_subplot(111)
    plt.plot(ledX,ledY,'o')
    # plot position of LEDs
    plt.hold(True)
    #plt.plot(coordX,coordY,'.')
    rect=patch.Rectangle((0,0), 6.95, 7.25, fill = False)
    # draw a rectangle that represents the walls of the room
    ax.add_patch(rect)
```

```
plt.axis('equal')
plt.xlim((-1,11))
# set x range
plt.ylim((-1,8))
# set y range
plt.grid()
```

I.2.4 Functions testing

```
import functions
from scipy.signal import lfilter

# the code below was used to compare the received signal with the ideal
# signal so we could see an eventual time shift.
# We searched for the place with the most successive ones in the bit
# sequence of the received signal and of the ideal signal. Then we plot
# both under each other so we can see an eventual time shift

def received_vs_ideal(array_peaksout , sequence1_samples):

    synch = map(float , array_peaksout [0:len(sequence1_samples)])
    counter = 0
    maximum = 0
    for x in range(0, len(synch)):
        if synch[x] > 0:
            counter = counter + 1;
            if counter > maximum:
                maximum = counter
                index = x
        elif synch[x] < 0:
            counter = 0
    # find number of most successive ones (+ index) in received signal

    synch = synch[index:len(synch)] + synch[0:index]
    # shift the list

    synch2 = map(float , sequence1_samples)

    time=functions.timegen(len(sequence1_samples),1e6)

    counter2 = 0
    maximum2 = 0
    for x in range(0, len(synch2)):
        if synch2[x] == 1:
```

```
        counter2 = counter2 + 1;
        if counter2 > maximum2:
            maximum2 = counter2
            index2 = x
    elif synch2[x] == 0:
        counter2 = 0
# find number of most successive ones (+index) in ideal signal

synch2 = synch2[index2:len(synch2)] + synch2[0:index2]
# shift the list

functions.grafosc(time, synch)
# plot the received signal

sequenceke = synch2
for x in range(0, len(sequenceke)):
    if sequenceke[x] == 0:
        sequenceke[x] = -1
for x in range(0, len(sequenceke)):
    sequenceke[x] = sequenceke[x] * 0.2 - 0.5
# scale the ideal bit sequence and move it down so we can see the two
# bit sequences under each other

functions.grafosc(time, sequenceke)
# plot the ideal signal

# Auto correlation: we filter the bit sequence on itself. If we plot the
# filtered signal in a graph, we can see very high peaks.

def autocorr(sequence_samples):

    filt = sequence_samples[::-1] # flips the bit sequence
    for x in range(0, len(sequence_samples)):
        if sequence_samples[x] == 0:
            sequence_samples[x] = -1
    # replace all zeros by '-1'

    auto = map(float, sequence_samples)

    auto = auto + auto + auto
    # repeat the bit sequence three times so we can see 3 peaks

    y = lfilter(filt, 1, auto) # correlation filter
```

```
time=functions.timegen(len(auto),1e6)

functions.grafcorr(time,y,0)
# plot in a graph
```