Development of a software-defined radio for the 2.4 GHz band

Bachelor's Thesis

DEGREE IN ELECTRONIC SYSTEMS ENGINEERING
Escola tècnica superior d’Enginyeria de Telecomunicacions de Barcelona
ETSETB

VICTOR PECANINS MARTÍNEZ
Author

MATTHIAS SCHULZ
Supervisor, Technische Universität Darmstadt

MATTHIAS HOLLICK
Advisor, Technische Universität Darmstadt

ALBERT AGUASCA
Advisor, Universitat Politècnica de Catalunya

August 2016
Abstract

Modern Software-Defined Radios (SDR) are a valuable educational tool, but their cost often discourages experimentation with a large number of nodes. A novel low-cost design capable of transmitting and receiving signals was brought into practice by assembling the first prototype. The design uses USB 3.0 to transfer sample streams to and from a computer in a real-time, streaming approach where required RAM size and latency are minimized. An asynchronous FIFO with almost-full and almost-empty flags was implemented in Verilog for the ICE40 FPGA, acting as a bridge between USB 3.0 and ADC/DAC. Interfacing with the USB chip was also implemented in the mentioned FPGA. The resulting device was able to transmit and receive low bandwidth signals generated by software on the 2.4 GHz band. Each part of the original design was tested, identifying possible hardware improvements for the next revision of the project.

Zusammenfassung

Resum

Actualment, la Ràdio Definida per Software (SDR) ha demostrat ser una eina útil per a l’educació, però el seu elevat cost dificulta poder experimentar amb un gran nombre de nodes. Un nou disseny per a una SDR de baix cost capaç de transmetre i rebre senyals ha estat dut a la pràctica muntant el primer prototip. El disseny fa servir USB 3.0 per a transferir seqüències de mostres des dels ordinadors en temps real, minimitzant la quantitat de RAM requerida i la latència. Una cua (FIFO) asíncrona amb senyals de quasi-ple i quasi-buit ha estat implementada en Verilog per a la FPGA ICE40. La cua actua com a pont entre USB 3.0 i els conversors digital/analògic. La comunicació amb el chip USB 3.0 també ha estat implementada amb aquesta FPGA. El dispositiu és capaç de transmetre i rebre senyals d’ample de banda reduït generades per software en la banda de 2.4 GHz. Cada part del disseny original ha estat comprovada, identificant possibles millores per a la propera revisió del projecte.

Resumen

Actualmente, la Radio Definida por Software ha demostrado ser una herramienta útil para la educación, pero su elevado coste dificulta poder experimentar con un gran número de nodos. Un nuevo diseño para una SDR capaz de transmitir y recibir señales ha sido llevado a la práctica montando el primer prototipo. El diseño usa USB 3.0 para transferir secuencias de muestras desde un ordenador en tiempo real, minimizando la cantidad de RAM necesaria y la latencia. Una cola (FIFO) asíncrona con señales de casi-lleno y casi-vacio ha sido implementada en Verilog para la FPGA ICE40. La cola actúa como puente entre USB 3.0 y los conversores digital/analógico. La comunicación con el chip USB 3.0 ha sido también implementada con ésta FPGA. El dispositivo es capaz de transmitir y recibir señales de ancho de banda reducido generadas por software en la banda de 2.4 GHz. Cada parte del diseño original ha sido comprobada, identificando posibles mejoras para la próxima revisión del proyecto.
# Table of Contents

Abstract iii  
List of Figures vii  
List of Tables ix  
Acknowledgements x  

1 Introduction 1  
1.1 Statement of purpose 2  
1.2 Project background 2  
1.3 Project specifications 2  
1.4 Project plan 3  
  1.4.1 Updated Gantt diagram 4  
1.5 State of art 5  

3 Method 6  
3.1 System overview 6  
3.2 Power supply 7  
3.3 Clock generator 8  
3.4 Universal Serial Bus (USB) 3.0 Interface 10  
  3.4.1 Driver API and GPIO issues 11  
3.5 Field-Programmable Gate Array (FPGA) 14  
  3.5.1 Start-up sequence 15  
  3.5.2 Peripheral configuration 17  
  3.5.3 Asynchronous First-In, First-Out (FIFO) 17  
  3.5.4 Loopback test 19  
  3.5.5 Sample streaming 21  
3.6 Analog Frontend 23  
3.7 Radio Frequency (RF) Transceiver 25  
3.8 Linux application 26  

4 Results 28
# Table of Contents

4.1 Test bench setup .................................................................................. 28  
4.2 Test procedure and discussion of results ............................................ 29  
4.3 Conclusion and future tasks ................................................................. 33  

5 Budget ..................................................................................................... 34  

6 Glossary .................................................................................................... 37  

7 References ................................................................................................. 39  

8 Annex ........................................................................................................ 40  
8.1 Programming Si5351 with Raspberry PI ............................................ 40  
8.2 Programming ICE40 FPGA with Raspberry PI .................................... 40  
8.3 Raspberry PI in the IceCube environment ......................................... 41  
8.4 Reading and writing to the FT600 ....................................................... 43  
8.5 Scripts for transmitting and receiving data ....................................... 46  
8.6 Proposed hardware improvements ..................................................... 50
# List of Figures

1. Implementations of digital radio ................................................. 1
2. Work packages ........................................................................ 3
3. Hardware block diagram ............................................................ 6
4. Finished prototype with parts indicated ....................................... 6
5. Si5351 block diagram ................................................................. 8
6. Mounted Clock Generator .......................................................... 9
7. Mounted USB Interface ............................................................... 12
8. FT600 timing diagrams ............................................................... 13
9. Mounted FPGA ........................................................................... 15
10. Startup sequence state diagram ................................................ 15
11. Startup timing diagram ............................................................... 16
12. Generic Ring Oscillator ............................................................... 17
13. Asynchronous FIFO block diagram ........................................... 18
14. FIFO pointer generation detail ................................................ 19
15. Loopback schematic ................................................................. 20
16. Schematic for sample streaming ............................................... 21
17. USB Side Finite State Machine (FSM) state diagram .................... 22
18. Analog Frontend Side FSM state diagrams ................................ 22
19. Mounted Analog-to-Digital converter (ADC)/Digital-to-Analog converter (DAC) ......................................................... 25
20. RF Transceiver simplified block diagram .................................... 25
21. Mounted RF Transceiver ............................................................ 26
22. Schematic of Software-Defined Radio (SDR) driver application .... 27
23. Test bench with parts indicated ................................................ 28
24. Test bench schematic ............................................................... 29
25. Seemoo-SDR Transmit (TX) test ............................................... 30
26. Transmitted baseband signal .................................................... 31
27. Distortion at the DAC output signal .......................................... 31
28. Seemoo-SDR Receive (RX) test ............................................... 32
29. Cost per unit breakdown .......................................................... 36
30. Raspberry PI Inter-Integrated Circuit (I2C) connections ............... 40
31  Raspberry PI Serial Peripheral Interface (SPI) connections  . . . . . . . . 41
32  Bug: Unrequired resistive divider  . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
33  Corrected: Direct connection to VBUS  . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
34  Isolated square pads detach from the board due to applied force  . . . . . . . . . 52
35  Interlaced tracks to solder a pin header  . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Project specifications</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Specifications of popular SDR platforms</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Summary of power supplies used in the system</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Power budget</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Summary of clock signals used</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>USB throughput comparison</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>Toolchain comparison for a test design</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>ADC/DAC key electrical characteristics</td>
<td>24</td>
</tr>
<tr>
<td>9</td>
<td>Bill of materials</td>
<td>34</td>
</tr>
<tr>
<td>10</td>
<td>Production costs</td>
<td>35</td>
</tr>
<tr>
<td>11</td>
<td>Engineering cost estimation</td>
<td>35</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to express my sincere gratitude to all the people who had made this project possible: To my family and friends for their unconditional support, to my advisor Matthias Schulz who gave me the opportunity to collaborate in this project, to Carsten Bruns, who worked in the antenna matching and the configuration of peripherals and made the collaboration an enjoyable experience, to the teachers on my home university who taught me everything I know about electronics, and to the co-workers at SEEMOO department who made me feel like in home during my visit.

Thesis Statement

\textit{pursuant to § 22 paragraph 7 of APB TU Darmstadt}

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. In the submitted thesis the written copies and the electronic version are identical in content.
1 Introduction

In an increasingly connected world, wireless communications play an important role in everyday life. Especially since the development of wireless computer networks and mobile phones in the last two decades, devices and techniques for transmitting digital signals have evolved, driven by the market demand on increasing data rates, lowering power consumption and production cost. This is a trend very likely to persist in the forthcoming years.

Usually the implementation of these techniques is based on RF Integrated Circuits (ICs), which provide all the electronics needed to transfer data from an antenna to a microprocessor (Figure 1a). These include analog (amplifiers, oscillators, mixers, filters, data converters) and digital circuits (for encoding/decoding signals according to the specifications of the protocol). This approach reduces development costs, allows to use the radio module as a reusable black box, freeing system designers from the complex task of RF design and standard compliance.

SDRs (Figure 1b) pursue a different vision. The goal is to have a transparent interface allowing the software to have direct control on the radio waves. Only the mandatory processing is made in hardware, i.e. frequency up- and down-conversion and digital to analog conversion. The sample streams are processed directly in the software layer.

Although this approach is not computationally- or cost-efficient for commercial mass-market communications, the flexibility of these devices and their ability of being reconfigured for virtually every communication protocol makes them a valuable instrument for research and experimentation.

However, since SDRs are a high-end product targeted to research facilities, their price is often prohibitive for individual users. There are currently low-cost SDRs [1] only capable of receiving. In some cases, such as education in practical communications and security, it would be useful to have a low-cost SDR capable of both transmitting and receiving. To serve this purpose, a new SDR has been developed.

Figure 1: Implementations of digital radio
1.1 Statement of purpose

The goal of this project is to demonstrate that using state-of-art commodity components and a minimalistic design approach, an SDR capable of transmitting and receiving arbitrary signals in the IEEE 802.11 (WiFi) band can be developed, with a reduced cost compared to other similar devices.

Special care has been taken in choosing the components so that the entire design can be assembled by hand, so it can be easily modified for experimentation. Its low-cost design makes it suitable for hobbyists and education in university, specially for laboratory demonstrations in communications or electrical engineering.

1.2 Project background

This project began with the work of Matthias Schulz, advisor of this thesis, who designed the schematic and the printed circuit board. My contribution begins with the assembly and test of the first hardware prototype and reporting modifications done to the initial design and bug fixes. As the work progressed, digital design of the FPGA in Verilog took most of my development time, especially in the part to transfer data between the USB interface and the ADC/DAC. Additionally, my work also included the software to transfer signals between the computer and the device.

The project also took benefit from the collaboration of Carsten Bruns, who designed and implemented the logic for configuring the chips using the FPGA, also the calculation of component values for antenna impedance matching and fixing hardware bugs. Both of us did the work separately, regularly merging the modifications in the whole design.

1.3 Project specifications

This project intends to be a low-cost experimentation platform for students, so the first requirement is the cost of the components must not surpass 50 €. The SDR should also be entirely mounted by hand, so chips using the Ball-Grid Array (BGA) package are not allowed. Additionally, BGA chips would require a multilayer Printed Circuit Board (PCB), which would also increment the costs. Only Surface Mounted Devices (SMD) and Quad-Flat No-Lead (QFN) chips will be used. The use of RF coaxial connectors would also increase the cost unnecessarily, so the SDR will include an integrated antenna. The electrical specifications of the developed device are reviewed in Table 1.

<table>
<thead>
<tr>
<th>Operating modes</th>
<th>TX/RX, Half-duplex</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF range</td>
<td>2.4 - 2.5 GHz</td>
</tr>
<tr>
<td>Sample Rate</td>
<td>22 MHz</td>
</tr>
<tr>
<td>Alias-free bandwidth</td>
<td>20 MHz</td>
</tr>
<tr>
<td>Power supply</td>
<td>USB 3 port</td>
</tr>
</tbody>
</table>

Table 1: Project specifications
1.4 Project plan

The project has been structured in four different work packages addressing the different tasks that have been carried out, as shown in Figure 2. The work packages were developed in parallel, as many of the tasks among work packages were related between each other.

The first work package consists in electronic prototyping and circuit assembly. The second work package involves digital logic circuit design and programmable circuits. The third package contains the tasks of testing, benchmarking and interpretation of results. The fourth work package includes all the documentation tasks that have been done throughout the development of the project.

The project has been developed over a period of five months. Two milestones were used to keep track of the project status: The Project Critical Review at the middle of the semester and the Final Presentation after the project was finished.

![Figure 2: Work packages](image-url)
1.4 Project plan

1.4.1 Updated Gantt diagram
1.5 State of art

In order to quantitatively compare our design with the current products, here we present a list of some popular SDRs and their specifications (Table 2).

**HackRF**

HackRF was one of the first SDR devices with open hardware. With origins in the amateur community, this single-board SDR is the cheapest device capable of transmitting and receiving over a wide band of frequencies. However, it is still too expensive for experimenting with a large number of nodes.

**USRP**

The Universal Software Radio Peripheral (USRP) is a modular SDR, consisting in a motherboard and two daughterboards. The motherboard includes the power supply, FPGA, reference oscillators, ADC/DAC and Ethernet Interface. Daughterboards include different transceivers and power amplifiers depending on the desired frequency band.

**RTL-SDR**

The RTL-SDR project [1] is based on inexpensive DVB-T sticks with the chip RTL2832U. The driver was reverse-engineered to work as an RX-Only SDR. Supported by the online community, this platform provides a low-cost introduction to SDR. Unfortunately, it is not capable to work with WiFi signals.

**WARP**

The Wireless Open-Access Research Platform [2] is a state-of-art SDR designed specifically for wireless networks, with support for Multiple-In Multiple-Out (MIMO). It is a fully programmable device capable of operating with or without a host computer.

<table>
<thead>
<tr>
<th>Name</th>
<th>RF Range MHz</th>
<th>Bandwidth MHz</th>
<th>Capabilities</th>
<th>Price USD</th>
</tr>
</thead>
<tbody>
<tr>
<td>HackRF One</td>
<td>1 – 6000</td>
<td>20</td>
<td>TX/RX</td>
<td>314.99</td>
</tr>
<tr>
<td>USRP N210 (w/SBX)</td>
<td>0 – 6000</td>
<td>40</td>
<td>TX/RX</td>
<td>1,810.00</td>
</tr>
<tr>
<td>WARP v3</td>
<td>2400 – 5000</td>
<td>100</td>
<td>TX/RX</td>
<td>4,900.00</td>
</tr>
<tr>
<td>RTL-SDR</td>
<td>24 – 1766</td>
<td>2.4</td>
<td>Only RX</td>
<td>20.00</td>
</tr>
<tr>
<td>This project</td>
<td>2400 – 2500</td>
<td>20</td>
<td>TX/RX</td>
<td>&lt;50.00</td>
</tr>
</tbody>
</table>

Table 2: Specifications of popular SDR platforms
3 Method

The experimental part of this project consists in assembling a working prototype of the design. After the prototype is completed and all the parts are functional, the resulting system will be characterized.

3.1 System overview

The complete circuit of the SDR has been divided into different functional modules (Figure 3) that perform individual tasks. Each subsystem is fully assembled and tested before proceeding with the next one.

In the following pages, the process of assembling the prototype is depicted. The report begins with an evaluation of the power supply requirements, then assembling the clock generator, FPGA, data converters and finally the RF transceiver, until all the components are assembled and the prototype can be tested as a whole (Figure 4) Problems encountered at each stage and the proposed solutions are discussed.

![Figure 3: Hardware block diagram](image)

![Figure 4: Finished prototype with parts indicated](image)
3.2 Power supply

The power supply for the system is provided by the host computer through the USB port. This provides a regulated supply of 5V, with a maximum current draw of 500 mA if using USB 2.0, which can be pushed up to 900 mA when using USB 3.0.

The 5 V supplied by the USB port are processed by integrated linear regulators in order to produce the necessary power supply for each subsystem (Table 3)

In order to avoid signals from digital circuits leaking into the RF circuitry, separate power supplies are used for the digital and analog parts of this circuit. In addition, a separate ground plane is used for digital and analog circuits to help preventing interference. The two ground planes are separated by a low-value inductor (choke), which filters the possible high frequency interfering signals.

The amount of power delivered by each linear regulator is limited. To ensure a reliable operation, the current consumed by each subsystem must be less than the maximum permitted. These parameters can be found in the respective datasheets, and have been collected into a table to check, at least in theory, that these limits are respected in the design.

In the case of the FPGA, it has four separate power inputs for the different input/output banks, plus one 1.2 V input for the logic core. The power consumption in the FPGA is highly dependant on the complexity of the design and the switching frequency. The programming tool reports an estimation of the power consumption of the FPGA as a whole, but does not specify the power consumed within each bank. We have assumed all the power is splitted in equal parts for the four banks, which gives us an approximation useful for computing the total power consumption.

For all the components, it is assumed that they are operating in the mode which is more power consuming. For the ADC/DAC, both converters are assumed to be ON, for the transceiver, it is assumed to betransmitting at maximum power, for the USB interface , it is assumed to be working continuously in SuperSpeed mode.

Remarkably, as seen in Table 4 the net VCCIO has been found to draw more current than the maximum permitted, mostly because the nominal current consumption of the USB interface working at SuperSpeed is 185 mA [5]. In future revisions of this design, a linear regulator with more output strength may be considered. A detailed list including this and other proposed hardware modifications is available in section 8 (Annex).

<table>
<thead>
<tr>
<th>Net name</th>
<th>Voltage (V)</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBUS</td>
<td>5</td>
<td>Main power input</td>
<td>USB</td>
</tr>
<tr>
<td>VDD</td>
<td>3.3</td>
<td>Analog circuits</td>
<td>TPS795</td>
</tr>
<tr>
<td>VCCIO</td>
<td>3.3</td>
<td>Digital circuits</td>
<td>MAX8882 A</td>
</tr>
<tr>
<td>VCC_FPGA</td>
<td>1.2</td>
<td>FPGA Core voltage</td>
<td>LP5907 U16</td>
</tr>
<tr>
<td>VDD_MAX5864</td>
<td>3.0</td>
<td>ADC/DAC Analog</td>
<td>LP5907 U17</td>
</tr>
<tr>
<td>OVDD</td>
<td>2.5</td>
<td>ADC/DAC Digital</td>
<td>MAX8882 B</td>
</tr>
</tbody>
</table>

Table 3: Summary of power supplies used in the system
### 3.3 Clock generator

Each subsystem needs a clock input that serves as a timing reference for the sequential digital and analog circuits. The frequency of the reference clock is different for each subsystem (Table 5). As each circuit is powered at a different voltage, the acceptable amplitude for the clock signal also differs.

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Frequency (MHz)</th>
<th>Signal amplitude (Vp-p)</th>
<th>Si5351 output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analog Front-End</td>
<td>22</td>
<td>3.0</td>
<td>CLK7</td>
</tr>
<tr>
<td>USB Interface</td>
<td>30</td>
<td>3.3</td>
<td>CLK3</td>
</tr>
<tr>
<td>RF Transceiver</td>
<td>40</td>
<td>3.3</td>
<td>CLK5</td>
</tr>
<tr>
<td>FPGA</td>
<td>22</td>
<td>1.2</td>
<td>CLK1</td>
</tr>
</tbody>
</table>

Table 5: Summary of clock signals used

The clocks on Table 5 are generated with a Si5351 integrated circuit "Programmable Clock Generator". From a single 27 MHz precision quartz crystal, up to 8 clock outputs with different frequency and amplitude can be generated. Figure 5 shows the block diagram of a single clock generator path inside the Si5351.

The input stage (Figure 5) generates the reference clock, from which the other clocks are generated. The reference can be an external quartz crystal, an external clock, or an Voltage-Controlled Oscillator (VCO).

![Figure 5: Si5351 block diagram](image-url)
3.3 Clock generator

The reference frequency should be accurate and stable, as all the other clocks in the system are referred to it. Most notably, the Local Oscillator (LO) frequency the RF transceiver is also referenced to this clock, which determines the frequency accuracy in transmitting and receiving. In this project, a quartz crystal is used as reference clock source.

The first stage (Figure 5) consists of two selectable Phase-Locked Loop (PLL) that generate a higher, intermediate frequency in the range between 600 and 900 MHz [3]. One of these PLL can be configured to generate a variable frequency for spread-spectrum applications, but this feature is not used in this project.

The second stage consists of fractional frequency dividers, that turn the signal generated by the PLL into several clocks with the desired frequencies. Thus, the frequency of the PLL has to be roughly the Least Common Multiple (LCM) of the desired clock frequencies. The parameters can be generated automatically with a manufacturer provided application [3] or manually [4].

Lastly, in the output stage, each clock can still be divided by an integer (R dividers). After that, the signals are buffered by power amplifiers to be able to drive peripherals. Each pair of amplifiers can be powered by a different supply, effectively allowing to have clock signals with different amplitudes. All the buffer amplifiers must be connected to a power supply to ensure the Si5351 is working correctly. These connections are done through low-value inductors (chokes) to avoid the generated clock signals to leak into the power supply lines.

At the beginning of this project, the registers were programmed using a Raspberry PI running a Python script capable of parsing the register map generated by the Clock Builder and send them through the I2C interface. The I2C cables are visible in Figure 6. More information about this procedure is available in section 8.

In the final prototype, the clock registers are programmed by the FPGA in the startup process as detailed in section 3.5.2
3.4 USB 3.0 Interface

Transmission of baseband samples and control signals between the SDR and the host computer is done through an USB 3.0 interface. This new version of the well-known USB specification recently developed adds extra speed to the data transfers. This throughput is needed to meet the bandwidth requirements of the ADC/DAC.

WiFi signals of interest have a bandwidth of 20 MHz. To minimize aliasing, the ADC/DAC has a bandwidth of 22 MSps. The samples are 8 bits for the I channel and 8 bits for the Q channel, a total width of 16 bits per sample. Then, the required ADC/DAC throughput is given by Equation 1

\[
22 \text{ MS/s} \cdot 16 \text{ bits/S} = 352 \text{ Mbits/s}
\]

It is desirable that those sample streams are handled continuously by the computer, in a real-time, streaming approach. Otherwise, a large amount of Random-Access Memory (RAM) should be used to cache the signals in the board, adding an extra cost to the hardware. Therefore, the throughput of the communication interface must be greater or equal than 1.

<table>
<thead>
<tr>
<th>Version</th>
<th>Theoretical Throughput (Mbits/s)</th>
<th>Expected Throughput (Mbits/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB 1.1 (Full Speed)</td>
<td>12</td>
<td>-</td>
</tr>
<tr>
<td>USB 2.0 (High Speed)</td>
<td>480</td>
<td>280</td>
</tr>
<tr>
<td>USB 3.0 (Super Speed)</td>
<td>5000</td>
<td>3200</td>
</tr>
</tbody>
</table>

Table 6: USB throughput comparison

As can be seen in Table 6, USB 3.0 is the only version capable of meeting the throughput requirements reliably. Therefore, it is justified to use this standard, even taking into account that it represents an extra cost.

The USB functionality is provided by IC FT600Q "Superspeed USB FIFO" [5] This integrated circuit implements the USB 3.0 protocol and physical layer, providing a 16-bit parallel synchronous bus interface suitable to be easily connected to an FPGA.

The chip operates on the FIFO principle: Data received from the host computer is stored in a queue in the internal memory of the chip, when the FPGA requests the data, it is delivered in the same order as the computer sent it. Similarly, the chip provides bi-directional transfers buffering the data coming from the FPGA and delivering it to the host computer when requested.

Two operation modes are available: Single-channel (245) mode provides one bidirectional communication channel. Multi-channel (600) mode provides up to four bidirectional pipes. The mode is selected by the host computer using the driver Application Program Interface (API). The timing diagram and waveforms are different for each mode. As the size of the internal RAM is fixed, it is divided in equal segments to allocate the buffer for each channel. Therefore, the queue length is smaller as the number of channels increases.
3.4 USB 3.0 Interface

In this project, the Single-channel mode was used, in order to take full advantage of the internal RAM of the FT600.

The parallel bus consists of 16 bidirectional data lines, 2 bidirectional byte enable lines and handshaking signals. The 16 data lines are divided into two bytes. The byte enable \texttt{BE[1:0]} indicates whether both bytes are valid or only one. In this project, both bytes are always used to carry I and Q samples respectively, so the value of these lines is not relevant. They are both pulled \texttt{HIGH} when writing to FT600.

Data lines are bidirectional and are used both when reading and writing to the USB FIFO, effectively providing half-duplex transfers. When the FPGA (or another bus master) reads from the FIFO, the data and byte enable pins are driven by FT600 (Figure 8a). The control line \texttt{OE_N} should be pulled \texttt{LOW} by the master in order to enable FT600 to drive these lines. When the bus master writes to the FIFO, data and byte lines are high-impedance inputs read by the FT600. \texttt{OE_N} must be pulled \texttt{HIGH} in this case.

When FT600 has received data from the host computer and its internal buffer is not empty, it pulls \texttt{LOW} the \texttt{RXF_N} control line, as seen in Figure 8a. Then, bus master detects data is available for reading, and it instructs FT600 to start outputting data by asserting \texttt{LOW} signals \texttt{RD_N} and \texttt{OE_N}. In this condition, data stored in internal buffer will appear on the bus, 16 bits each clock cycle. The data stream may be interrupted either by FT600 (pulling \texttt{RXF_N = HIGH}) or by the bus master (pulling \texttt{RD_N, OE_N = HIGH}) \textsuperscript{1}

Similarly, FT600 signals it has space available in the internal buffer by pulling \texttt{LOW} the \texttt{TXE_N} line (Figure 8b). Then, the bus master pulls \texttt{LOW} the \texttt{WR_N} line and \texttt{HIGH} the \texttt{OE_N} line, indicating data will be written to the FT600. This data will be transmitted through the USB interface in packets of 1024 bytes.

Data bits change near the falling edge of the clock. Therefore, they should be sampled by the bus master at the rising edge, in order to allow them to settle down into the correct value.

Bus clock is generated internally by FT600 using its own PLL, referred to a 30 MHz clock generated by the clock generator. The frequency of the bus clock is selectable by the host computer using the FT600 USB API, and it can take values of 40, 50, 66 or 100 MHz. \textsuperscript{2}

3.4.1 Driver API and GPIO issues

The manufacturer provides a driver for the chip as a C library, and example code as well [6]. The driver is available for Windows and has partial support for Linux. It implements procedures to transmit and receive data buffers through USB interface, initializing the device and configuring parameters such as bus clock frequency or number of channels.

In addition, FT600 incorporates two General-purpose Input/Output (GPIO) pins, which can be configured as inputs or outputs and controlled through the USB driver. The initial idea was to use these pins to switch between a \textit{Normal Mode}, where the SDR would be

\textsuperscript{1}Stream interruption by bus master is not expressly documented in the datasheet [5], however this desirable behavior is observed in the tests.

\textsuperscript{2}40 MHz and 50 MHz clock not documented in the datasheet, but available in the API source code.
transferring data samples between USB interface and ADC/DAC, and a *Configuration Mode*, where the data would be interpreted as instructions for configuring the peripherals on the board (such as tuner LO frequency, amplifier gain, filter bandwidth and so on).

However, the GPIO functionality has never worked, by the time of our tests (July 2016), neither on Windows (v. 1.1.0.0) or Linux (v. 0.4.4) drivers. After contacting with the manufacturer, it was confirmed that this feature was not implemented yet, as the driver is still on Beta phase. Therefore, as a temporal solution, configuration of peripherals is done by introducing an escape sequence in the data stream (3.5.2).

![Mounted USB Interface](image)

**Figure 7: Mounted USB Interface**

Test wires for connection to the logic analyzer can be observed at the right.

---

3More recent drivers, version 0.4.10 and 1.2.0.5 were made available shortly after the development of the prototype finished. These drivers advertise GPIO function has been implemented.
3.4 USB 3.0 Interface

(a) FPGA reading data from FT600
Stream stopped by FT600

(b) FPGA writing data to FT600.
Stream stopped by FPGA

Figure 8: FT600 timing diagrams
Using 245 single-channel mode.
3.5 FPGA

The core processing unit of the project is an FPGA. In this project, the model ICE40HX1K is used. This integrated circuit consists of a reconfigurable array of sequential and combinational logic circuits, that can be configured to control the rest of the peripherals and to arbitrate data transfers.

The design implemented in the FPGA is done in terms of logic elements (such as gates, registers, finite state machines). In the process called Design Flow the high-level circuit description in Verilog goes under various steps until it is loaded to the chip (Synthesis, Mapping, Place & Route, Bitmap Generation).

In synthesis, the combinational and sequential circuits are simplified and expressed (Mapping) in terms of Technology Primitives [7], that are physically built in an array-like structure. Then, in Place & Route, each primitive is assigned a position in the array by an optimization algorithm, taking into account the delay caused by interconnections. The configuration of all blocks in the array is encoded into a Bitmap.

IceCube and Project IceStorm

Interestingly, two different software toolchains are available implementing the design flow for ICE40. The first one is the commercial tool supplied by the manufacturer, Lattice Ice Cube. Alternatively, there is a fully open-source toolchain, Project Ice Storm [8].

The commercial toolchain is used most of the time to develop this project. An uncomplete design was also successfully compiled with Project Ice Storm. As it is an open project under active development, not all features are yet supported. To be able to compile the complete design with Ice Storm, it would be possible to collaborate with the project implementing the required features (i.e. Manual placement of primitives). However, this task was decided to be beyond the scope of this project.

<table>
<thead>
<tr>
<th>Performance parameter</th>
<th>IceStorm</th>
<th>Lattice Ice Cube</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthesis time (s)</td>
<td>2.52</td>
<td>14.0</td>
</tr>
<tr>
<td>Placer time (s)</td>
<td>1.07</td>
<td>21.3</td>
</tr>
<tr>
<td>Router time (s)</td>
<td>0.54</td>
<td>13.0</td>
</tr>
<tr>
<td>No. of logic cells</td>
<td>351/1280</td>
<td>397/1280</td>
</tr>
<tr>
<td>Max. clock frequency (MHz)</td>
<td>17.58</td>
<td>77.0</td>
</tr>
</tbody>
</table>

Table 7: Toolchain comparison for a test design

Programming the FPGA

Once the bitmap is generated by the toolchain, it must be transferred to the FPGA through SPI. For prototyping, this bitmap must be loaded each time the board is powered on. A Raspberry PI was used on this purpose. The bitmap is uploaded to the
Raspberry PI when ICE Cube finishes compiling, and a script is called remotely to transfer the bitmap through SPI connected to the board (Figure 9), as detailed in section 8. In production, it can be programmed permanently into FPGA Read-Only Memory (ROM).

![Figure 9: Mounted FPGA](image)

Two main tasks are done by the FPGA: Configuration of the rest of peripherals (clock generator, ADC/DAC and Transceiver) and transfer data between USB interface and ADC/DAC. These two operations are separated in Configuration Mode and Normal mode. Additionally, the FPGA runs a **Start-up sequence** when it is powered up.

### 3.5.1 Start-up sequence

As soon as the board is connected to a power source, the FPGA runs a boot-like initialization sequence (Figure 10), where the peripherals are configured to provide minimum functionality and communication through USB interface.

The startup sequence is initiated whenever the board is connected to a computer providing power through USB, when the FPGA has been just programmed through SPI, or when an external reset is induced by pulling LOW the pin `fpga_gpio[4]`, which is connected to the signal `ext_nreset`. A push button has been wired to induce external reset for convenience (not shown in Figure 9)

![Figure 10: Startup sequence state diagram](image)
Right after the signal `ext_nreset` transitions from LOW to HIGH (Figure 11[a]), the internal clock is enabled and a countdown produces a delay of approximately 4.3 ms, allowing external devices to start correctly. This delay is produced by the startup component.

After that, `cfg_nreset` is pulled HIGH (Figure 11[b]), activating the I2C master to load a predefined register map to Si5351. The clock generator is configured to produce a 30 MHz clock, needed by the USB interface. A second delay of about 174 ms is produced to allow I2C to finish writing registers and ensure a steady clock is provided to FT600 before it is initialized by pulling HIGH the `ft600_nreset` signal (Figure 11[c]).

A short delay of about 17 us is needed to wait for FT600 to initialize. Finally, the internal logic is activated by pulling HIGH the `nreset` signal (Figure 11[d]). The SDR is now ready to transfer data (Normal mode) or configure peripherals on the orders of USB interface (Configuration mode).

### The problem of Clock Generator

In the first steps of startup sequence, the Si5351 clock generator is still not configured, so it does not provide any clock to FPGA. Yet, some clock signal is needed to run the internal FSM and I2C master in order to configure the registers.

To solve this problem, an internal clock signal is generated by means of a ring oscillator (Figure 12). This oscillator takes advantage of logical feedback and propagation delay of the internal FPGA gates to generate a periodic signal. No additional hardware is needed to generate this clock.

The frequency of such ring oscillator is mainly determined by the number of NOT gates in the chain and the input-to-output delay of the gates [9], given by the expression in Equation 2

\[
f = \frac{1}{2N\tau}
\]

where \( f \) is the oscillation frequency in Hz, \( \tau \) is the propagation delay in seconds and \( N \) accounts for the total number of gates.

Propagation delay is subject to temperature and pressure variations, also depends on the supply voltage. Because of this, ring oscillators are rarely used in open-loop designs (that is, outside a PLL). Nevertheless, I2C interface does not require a tight frequency specification, making the solution suitable for this task.
3.5 FPGA

A chain with an odd number of NOT gates generates negative feedback suitable for oscillation.

In this design, 48 Look-Up Tables (LUTs) from the FPGA were allocated for the buffer chain of the ring oscillator. These LUT were placed manually in a fixed position, to prevent the synthesis tool to move them randomly in the placement process, altering the frequency each time the design is synthesized. With 48 LUT, the frequency obtained was approximately 30 MHz, this clock is then divided to provide clock signal for the I2C interface.

3.5.2 Peripheral configuration

While the device is in normal operation, it may be necessary to change the configuration of the peripherals, e.g. to change the tuner frequency. This is done when the SDR is in Configuration Mode.

The device enters Configuration Mode when a specific escape character is received from the USB interface. The escape character is 0xFFFF. This value was selected because it corresponds to the maximum output value of the DAC, and it is very unlikely to occur in actual sampled signals. This value is not interpreted as a regular sample, but as an instruction that signals the next few bytes must be interpreted as configuration commands. By doing so, the DAC dynamic range is slightly reduced, as only 255 out of 256 codes are available. The performance of the prototype should not be affected by this.

In future versions of the design, it should be possible to enter Configuration mode also by pulling LOW the pin ft600_gpio[0]. This would allow to enter Configuration mode without using escape sequences. However, as the support for GPIO on FT600 is limited, this functionality is not yet implemented.

3.5.3 Asynchronous FIFO

The conceptually simple task of transferring data between the USB interface and the analog frontend holds several implementation challenges.

First, the number of bits of each bus is different. To solve this issue the effective number of bits is reduced, as discussed in 3.6

Second challenge is the two peripherals operate in different clock domains. Transfers with the USB interface are in bursts at a 100MHz clock. However, a continuous sample stream at 22Msps should be guaranteed to the analog frontend. Moreover, it is desirable that such transfers are as close as possible to real-time, with minimum latency.
The standard approach to this requirement is using an **Asynchronous FIFO**. This component can be thought as a ring buffer with independent read and write clocks.

Transferring data between two clock domains has been shown to be problematic [10]. Especially for generating empty and full flags, signals in different clock domains must be compared. Care must be taken on synchronizing this signals to the respective clocks, otherwise metastability can happen, leading to very hard to detect failures.

For this reason, a third-party FIFO design is used as reference. Several modifications have been done to the original code [10], to provide extra functionalities: Support for almost empty and almost full signals, support for parallel ICE40 block RAM primitives [7], and support for initialization of FIFO contents (this last feature is not used now, but can be useful in further designs).

The critical part of FIFO design is generation of **full** and **empty** signals. These signals are important not only for the external components, but are also used in order to avoid underrun or overrun conditions. The **full** and **empty** signals are both generated when the read and write address pointers are equal. These pointers are generated in different clock domains, so they must be synchronized in order to compare them. The pointers are converted to Gray code prior to synchronization, to avoid race conditions caused by different propagation times on each bit. This is a common technique in asynchronous FIFO design, which relies on the fact that successive Gray codes only differ in one bit [10].

In order to distinguish between **full** and **empty** conditions, the read and write counters are one bit longer than the actual address bus from RAM. This extra bit indicates when the write pointer has wrapped around the circular buffer. When the RAM addresses are equal, if the extra bits from read and write pointers are equal, **empty** signal is asserted. If the extra bits are different, **full** condition is asserted.
3.5 FPGA

Figure 14: FIFO pointer generation detail

The almost_full and almost_empty signals have been implemented as suggested in [10], by having two additional counters, that are anticipated a few positions in respect to the actual read and write address. The "almost" signals are generated by comparing the anticipated pointers with the actual pointers.

An important characteristic of the FIFO is the maximum clock frequency at which it can be safely operated. This maximum frequency is related to the propagation delay of signals in combinatorial logic blocks. The more complex a logic block is, the more levels of gates in series it has, therefore the propagation delay is longer and the frequency lower. It is desirable to simplify the logic circuits to operate at a higher frequency, or at the same frequency but more reliably.

The logic for generating almost_empty, almost_full flags is in fact the most complex block in the FIFO system, since it involves a Gray-to-binary converter and a binary comparator. This logic is effectively limiting the maximum frequency of the entire system. Lattice ICECube tools are generally locating the critical path in these blocks.

In future improvements of this design, it would be worth to develop some solution to this bottleneck. For example, implementing a digital less-than-or-equal comparator that works directly on Gray pointers, without having to convert them before into binary code.

3.5.4 Loopback test

The asynchronous FIFO component alone is not enough to transfer data between two peripherals. Some control logic is required in order to match the timing diagram of the USB interface. On this purpose, a FSM has been designed.

Before dealing with the design of a complete data flow between the USB interface and the Frontend, a Loopback Test has been developed. The purpose of this design is reading data from the USB interface, storing it in the internal FIFO, and when requested, writing it back to the same interface.

This is useful for checking the reliability of the communication between these two compo-
nents, not having to worry about synchronization of different clock domains. In fact, this Loopback FSM will be used with little modification in the next section, for the complete design.

The FSM is governed by the request signals TXE_N, RXF_N from FT600 and signals rempty, wfull from the FIFO. If signal TXE_N = LOW and rempty = LOW, the FSM starts transferring data from internal FIFO to USB interface. Similarly, when signals RXF_N = LOW, wfull = LOW, data is written into the internal FIFO. When both read/write request signals are LOW, then signals wfull, rempty are used to decide, giving priority to transferring data from USB interface to the internal FIFO. By doing this, it is prioritized that the SDR will transmit as soon as the host computer starts sending data.

In the preliminary tests, it was discovered a bug in the FSM that caused problems with the first and last bytes. This is because ICE40 block RAM primitive only outputs valid data after an edge in read clock. This causes an additional cycle of latency, meaning that the byte that has been read is not the one pointed by the read pointer, but the previous one. This causes problems only when reading the first word.

To solve this, an additional flip-flop is used to indicate the current data at the RAM read port is invalid, and it has to be refreshed before sending it to the USB interface. The data is invalidated when the FSM starts writing to the FIFO when it was empty.

Figure 15: Loopback schematic
3.5.5 Sample streaming

After having completely understood and tested the communication between USB Interface and FPGA, the next step is designing a data path between the USB Interface and the ADC/DAC. For this, two FIFOs were needed, for transmitting and receiving.

When transmitting, the DAC FIFO takes the samples coming in bursts from the USB Interface and delivers them as a continuous stream to the DAC. Similarly, when receiving the ADC FIFO takes data from the ADC and delivers it to the USB Interface. Remarkably, in both cases the read and write clock for the FIFOs have different frequencies.

In the USB side, the FSM is a variation of the design used in the Loopback test. The main difference is that now there are two different FIFOs, also the mechanism for changing from Idle to Push or Pull is slightly different. It only makes sense that the USB interface writes to the DAC FIFO, and reads from the ADC FIFO. Therefore, signals dac_winc, dac_wfull, dac_almost_full are connected to the DAC FIFO, and adc_rinc, adc_rempty, adc_almost_empty are connected to the ADC FIFO.

In the Loopback, wfull=0 and rempty=0 signals are used to exit from the Idle mode and start Push or Pull respectively. But if, for example, the ADC is pushes one sample into the FIFO, the USB will detect the FIFO is not empty, and will read this byte. However, the FIFO will be empty again and the USB will stop reading having read only a single byte. This would lead to a lot of delay caused by mode changes and render the FIFO useless, as it would be empty the most of the time.

By using almost-empty and almost-full signals, the USB interface has time to process the requests and sending the data before the analog side has to stop. So the FIFOs are in fact accepting samples in one side while delivering them to the other side. Solving this problem was crucial for ensuring a glitch-free streaming operation.

---

**Figure 16: Schematic for sample streaming**
By contrast, in the USB FSM, the respective \textit{almost\_*} signals are used for entering Push or Pull, and \textit{empty, wfull} are used for going back to Idle mode. This is to guarantee a minimum length in the transfers and improve efficiency.

In the frontend side, there are two separate FSM for ADC/DAC. The ADC FSM simply starts pushing samples to the ADC FIFO when \textit{adc\_almost\_full}=0, to make sure there is enough space available, and keeps pushing until \textit{adc\_full}=1. The DAC FSM works very similar, starts pulling data from the DAC FIFO when \textit{dac\_almost\_empty}=0 until \textit{dac\_empty}=1. Additionally, the DAC FSM also controls the value of the \textit{RXTX} signal, indicating to the transceiver chip when it has to activate TX circuitry.
As there are two different FSMs on the ADC/DAC side, this means the chip can actually be transmitting and receiving at the same time (full duplex). In fact, the ADC/DAC chip and the transceiver are designed to work in full duplex, and the only factor that could limit the SDR to operate in half-duplex would be the USB interface, because the parallel bus can only transmit in one direction at a time.

Even with this limitation it may be possible to achieve an overall full-duplex design, for that, the USB interface should switch direction fast enough to maintain both streams. From the throughput calculations we can see this is possible in theory. It may be interesting to experiment with this in the future, however this operation mode has not been extensively tested.

The mechanism of using flags for signalling when the data in the FIFO is invalid has been skipped from this explanation, but it works the same way here as in the loopback test.

3.6 Analog Frontend

The analog frontend is the bridge between the digital and analog parts of the circuit. The integrated circuit MAX5864 was selected to occupy this role. The chip includes two DACs, to convert a stream of In-Phase/Quadrature (IQ) samples into a dual-channel baseband analog signal, and two ADCs to convert data in the opposite way. Additionally, the chip has also a precision voltage reference generator and an SPI interface to selectively turn on and off the data converters.

The SPI interface is governed by the FPGA. As with the clock generator, the ADC/DAC register can be programmed using configuration escape sequences. In this prototype, the most usual configuration is to turn on both the ADCs and DACs, as power consumption is not critical.

Digital signals on the data converters are registered by a single clock signal. The frequency of this clock determines the sample rate of the ADCs and DACs. The maximum sample rate for the MAX5864 is 22 MHz, and there are several chips from the same family that are compatible and provide more bandwidth. The actual sample rate can be changed arbitrarily by changing the frequency of the clock in the clock generator.

In each clock cycle, two samples are converted. Those are the I samples and the Q samples. The I samples are registered at the falling edge of the clock, and the Q samples are registered at the rising edge. This scheme is known as Double Data Rate (DDR), and the FPGA has dedicated primitives to pass between these signals and the corresponding double width parallel, single clocked signals that are used in the FIFO.

The samples of the analog frontend are internally packed into 16-bit words in the FIFO. For the ADC, the 8 Most-Significative Bits (MSBs) represent the I data and the 8 Least-Significative Bits (LSBs) are the Q data. The same applies for the DAC, but only 8 MSBs out of 10 bits are used, assigning zero to the two LSBs.

This approach allows to take advantage of the entire dynamic range of the DAC. However, by setting to zero the two LSBs, the precision is degraded, thus reducing the maxi-
3.6 Analog Frontend

minimum achievable Signal-to-Noise Ratio (SNR). Ideally, the SNR reduction is given by the well-known formula for the quantization noise (Equation 3)

$$SNR_{reduction} = 6.02N_{\text{bits}} \approx 12dB$$ (3)

The theoretical result in Equation 3 is useful to have a quantitative idea of the impact of zero-padding in the maximum achievable SNR, but in practice, the measured noise and distortion levels were much higher due to hardware issues.

On the analog side, signals are in differential mode, to minimize the effect of RF radiation and interference. The transceiver chip MAX2830 works also with differential signals, so the connection between two chips can be simplified.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Unit</th>
<th>ADC</th>
<th>DAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common mode voltage</td>
<td>V</td>
<td>1.65</td>
<td>1.29 - 1.5</td>
</tr>
<tr>
<td>Differential full-scale voltage</td>
<td>mV</td>
<td>± 512</td>
<td>± 400</td>
</tr>
<tr>
<td>Input resistance</td>
<td>kΩ</td>
<td>245</td>
<td></td>
</tr>
<tr>
<td>Input capacitance</td>
<td>pF</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Recommended load resistance</td>
<td>kΩ</td>
<td>70</td>
<td></td>
</tr>
</tbody>
</table>

Table 8: ADC/DAC key electrical characteristics

The electrical conditions must be satisfied for a proper performance of the ADC/DAC. For the ADC, there is no problem as the input impedance is fairly high, and the transceiver will be able to drive it. The common-mode voltage at the ADC should be as close as possible to 1.65V. The transceiver can set a common-mode voltage configured by the SPI interface.

However for the DAC, only signals below 1 MHz were produced. As the frequency of the signal increased, distortion and attenuation appeared in the signal. The origin of the DAC problem was not known in the time this thesis was finished. The believed cause of the problem is that the DAC output is not strong enough to drive the transceiver TX input correctly. In fact, the transceiver TX input has an impedance of 20 kΩ, while the DAC is designed to drive a load of no less than 70 kΩ. The manufacturer was contacted for support, and the use of a buffer amplifier was advised. In future generations of this design, a buffer should be included at the output of the DAC.
3.7 RF Transceiver

The RF Transceiver chip MAX2830 is the core radio subsystem of the board. It integrates all the necessary blocks to do frequency translation between RF and baseband, including but not limited to mixers and PLL. The internal blocks have variable parameters that can be configured through SPI interface. The baseband inputs and outputs are differential, the same as the ADC/DAC analog pins.

In the transmitter chain (Figure 20) the baseband signal is first filtered by a low-pass antialiasing filter. This filter can be understood as a reconstructing filter to attenuate the steps of the DAC. Then, the filtered signal is feed into a frequency upconverter, using two mixers, one for each channel. These mixers generate a signal in the Wifi band. The resulting signal drives a power amplifier, whose output is connected to the antenna when the device is transmitting.

Figure 19: Mounted ADC/DAC

Figure 20: RF Transceiver simplified block diagram
Adapted from datasheet [12]
3.8 Linux application

In the receiver chain, the signal captured by the antenna is fed into a low-noise amplifier. Then, it enters a frequency down-converter to produce a baseband signal. The baseband signal is filtered with a lowpass filter to remove the image frequencies. A Received Signal Strength Indicator (RSSI) block is used to detect the power of the received signals. Finally, an adjustable gain can be applied to the baseband signal to match the full dynamic range of the ADC.

Both up- and down-converter are using the same LO signal, therefore the device transmits and receives in the same band. To avoid interference, the antenna is only connected to one of the TX or RX blocks at a time. The antenna switch is controlled by the RXTX pin.

The local oscillator consists of a PLL that generates a signal on the 2.4-2.5 GHz range, based on the reference frequency provided by the clock generator. The frequency is tuned by changing the value of the PLL divider through SPI interface.

To simplify setting the values of the registers and configuring the transceiver, a Graphical User Interface (GUI) application was developed by another student who is also working on this project.

3.8 Linux application

The manufacturer provides a C++ based API to transfer data streams and configure the FT600 USB interface. Using the API directly would mean that the program should include the code for generating and processing signal streams. This would be an unnecessary effort, because there are already good tools for doing that, for example: GNU Radio, Matlab, Octave...

The goal of the application or driver is to encapsulate the FT600 API and provide a standard way of transferring streams that is compatible with the rest of the software. This is accomplished in Linux with the use of special files, also known by fifos or named pipes. These special files can be handled as regular files by the application, using the standardized system calls `open()`, `close()`, `read()`, `write()`. Instead of writing to the filesystem, these files transfer data between processes in Linux Operating System (OS).
As named pipes are unidirectional, two pipes are created: one for transmitting data, the other one for receiving. In the application, there are two threads running in parallel, one reading from the TX pipe and writing to the FT600, the other one reading from FT600 and writing to the RX pipe. External applications can write to TX pipe and read from RX pipe. Additionally, a third thread measures the number of bytes transmitted each second and displays the throughput in the terminal.

Additionally, the application can be used not only to transfer data samples to the SDR, but also to configure peripherals, sending an escape sequence followed by the appropriate command.

The relevant code for the reader and writer thread, demonstrating how to use the FT600 API is available on the Annex of this document.

Figure 22: Schematic of SDR driver application
Results

In addition to the individual tests for each subcircuit that were carried out during the prototype assembly, the system was tested as a whole and its capacities were evaluated. The global features of the SDR were tested, namely transmitting and receiving data, and configuring the peripheral chips.

4.1 Test bench setup

The setup for evaluating the board was assembled in a regular communications laboratory, no special shielding or isolation against interfering signals was used. Apart from the prototype SDR board, the test bench included a laptop, a Raspberry PI, an USRP, and an Ethernet Switch. An additional mixed-signal oscilloscope was used for debugging and monitoring signals on the board (Figure 23).

![Figure 23: Test bench with parts indicated]

The Seemoo-SDR board is permanently attached to a Raspberry PI through an SPI interface. This is for programming the FPGA each time the SDR board is powered on. The prototype is connected through an USB 3 cable to a regular laptop. The laptop used Ubuntu Linux 15.04, and had an USB 3.0 port operating at SuperSpeed mode.

The laptop was used to program the FPGA, operate the prototype SDR through the USB 3.0 port, and operate the USRP to test the prototype. Both Raspberry PI and USRP were connected to the laptop via Ethernet, but the laptop only had a single Ethernet port. This made necessary to use an Ethernet Switch.

The laptop was connected to the wall power through its respective charger throughout all the tests. This was made to ensure a proper ground connection to all the devices and to ensure enough power can be delivered through USB port. If the laptop was disconnected from the charger, some problems were observed in the USB interface of the prototype, the origin of this behavior has to be investigated in the future.
Two tests were carried out: First, transmitting a signal with the Seemoo-SDR and receiving it with the USRP; second, transmitting with the USRP and receiving with Seemoo-SDR. The Seemoo-SDR and the USRP radio signals could be connected using a coaxial cable or wireless using a 2.4 GHz antenna. In both cases the tests were satisfactory.

### 4.2 Test procedure and discussion of results

The prototype was able to transmit and receive signals in the Wifi band, although the bandwidth requirements planned from the beginning were not met. The features of data streaming and peripheral configuration were working as expected.

#### Seemoo-SDR as a transmitter

The prototype board was connected to an USB 3.0 port, and the Raspberry PI was powered up. The Lattice ICE Cube environment was initiated, and the FPGA was programmed by the process depicted in the Annex. In order to check that the FPGA has been correctly programmed, the 30 MHz clock output from Si5351 to USB interface was monitored with an oscilloscope. If the signal is not present, the Raspberry PI and the SDR board must be disconnected from power supply and connected again.

To check the USB interface has been correctly initialized, the linux tool `lsusb` has to show the *Future Devices International Ltd, SuperSpeed FIFO Bridge* device. If the device is not shown, but the 30 MHz clock is present, then the SDR board has to be reset a couple of times by pressing the hard reset button.

Once the USB device is correctly detected by the laptop, the linux driver providing pipe access to FT600 has to be started in a terminal. The special files `./tx` and `./rx` will be created in the working directory, to write and read from the FT600 respectively.
4.2 Test procedure and discussion of results

The GUI application for programming the peripherals should be started at this point. The access to the board will use the created pipes. The Analog Frontend chip should be configured in Transceiver Mode, for activating both ADC and DAC. For the RF Transceiver chip, the `init_regs` command should be executed first, then `set_mode` with `SHDN_N=1` to put the chip into active mode. Then, we can set the transmitter frequency to 2.412 GHz, or Wifi channel 1.

The USRP should be receiving at the same frequency in order to capture the transmitted signal. The USRP Hardware Driver (UHD) driver package includes an utility to show the Fast Fourier Transform (FFT) of the received signal in real time. To do so, execute the command `uhd_fft -f 2412M -s 20M`. The proper antenna input should be selected in the GUI.

Then, a signal can be transmitted by simply writing the samples to the special file `./tx`. The frequency plot of the transmitted signal will appear on the UHD screen.

In Figure 26 we can see a 150kHz sine wave produced by the DAC. We can observe the signal is free from glitches. Even though the FIFO is continuously receiving data bursts, the DAC is producing a continuous signal and the RXTX is set accordingly. This shows the streaming mechanism implemented in the FPGA is working as expected.

In Figure 25 we see the 150KHz single tone modulated at 2.412 GHz, and received correctly by the USRP. The SNR is around 40 dB, and the first overtone is 25 dB lower in amplitude than the original signal.

There is a problem when transmitting signals with a higher frequency. For example in ?? a 5MHz signal is transmitted, the DAC attenuates the signal and adds distortion, which degrades the SNR considerably. The origin of this problem is believed to be the output strength of the DAC, and the board should be modified to include a buffer amplifier in the next revision of this design.

![Figure 25: Seemoo-SDR TX test](image-url)
4.2 Test procedure and discussion of results

Figure 26: Transmitted baseband signal
The frequency of the sine wave is 150kHz. From top to bottom:
Differential DAC I outputs, RXTX pin,
DAC FIFO flags: Empty, almost-empty, full, almost-full

Figure 27: Distortion at the DAC output signal
The frequency of the sine wave is 5MHz.
Distortion is observed and the amplitude of the signal is significantly reduced.
Additive noise is also observed.
4.2 Test procedure and discussion of results

Seemoo-SDR as a receiver

The performance of the prototype at receiving signals was also demonstrated. With the same setup as for transmitting, the UHD FFT utility must be turned off, and the UHD signal generator utility should be started. That can be done with the command `uhd_siggen -f 2412M -s 20M -x 150000`. This will generate a 150kHz sine signal modulated at 2.412 GHz.

The signal can be received by reading from the special file `./rx`. The raw samples can be processed by an external application. In this case, we used an Octave/Matlab script to plot the samples and calculate the FFT.

The received signal was almost free from distortion, but had a considerable amount of added noise. This could be caused by the thermal noise in the transceiver, or by leakage of clock signals into the ADC path. To fix this, the sensitivity of the receiver could be adjusted, by changing the registers that control the Low-Noise Amplifier (LNA) gain and filter bandwidth.

![Figure 28: Seemoo-SDR RX test](image-url)

Received samples and FFT
4.3 Conclusion and future tasks

This project began as a theoretical design. During these months, we have produced a first prototype capable of transmitting and receiving signals in the Wifi band, with limited bandwidth. Although the bandwidth and distortion requirements targeted from the beginning were not met, the prototype serves as a proof-of-concept, demonstrating the initial design can be brought to reality.

It has also been demonstrated that the original design will need modifications in the future development, and those modifications have been recognized.

The most time-consuming task of this project has been logic design. At the beginning, I had to learn the Verilog language from scratch, as I worked before only with VHDL. The syntax of both languages are different, but the concepts behind are very similar and that made it a matter of adapting my knowledge while learning new things.

Another time-consuming task was the physical assembly of the prototype. Electrical failures such as bad contacts or short circuits, or soldering QFN chips were very hard to debug given the small size of the circuit features (often less than half a millimeter). Sometimes, making a modification in one part of the circuit apparently caused another unrelated part to fail. My advice on these cases would be, to check proper connections and re-do solderings of the failing part, checking signals with oscilloscope when possible and being sure the supply voltages have the correct value.

In the future development of this project, the most critical task is fixing the bandwidth and distortion problems in the ADC/DAC. Then, a second revision of the board should be redesigned including the modifications on the annex.

Another interesting task would be reverse-engineering the FT600 USB driver. There are only a few chips on the market that support USB 3, and FT600 is by far the cheapest. Not only this, FT600 is the only chip available in a QFN package. Other chips use BGA packages, which need a PCB with more than two layers. The cost of a multi-layer PCB is significatively greater than a 2-layer PCB. Additionally, BGA cannot be soldered by hand, discouraging its use in hobby projects. For this reasons, FT600 could have applications in many other open-source or hobby designs, and it would be interesting to have an open-source driver to have full control over the chip.

Finally, it would be also quite interesting to use only open-source tool Ice Storm to compile the FPGA design. This would require computer-science skills, as the tool is under active development and many features still have to be implemented. The fact of using an open-source tool for logic design in an SDR would be unprecedented, and in accordance with the philosophy of this design.
5 Budget

The project has been developed from the beginning keeping always in mind the economic viability of the finished product. Although that this is the first prototype and the design has to be modified before reaching the market, a study about the costs of this project gives an idea about the final product.

The bill of materials (Table 9) shows the price of the most important components on the board. Passive components have been omitted, because their cost is very small in comparison to the price of integrated circuits. This is true specially for high production volumes. For a hobbyist wanting to mount this board by hand, the price of individual passive components may be significant.

<table>
<thead>
<tr>
<th>Part No.</th>
<th>Description</th>
<th>Units per board</th>
<th>€/unit* (1 u.)</th>
<th>€/unit* (1000 u.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICE40HX1K-VQ100</td>
<td>FPGA</td>
<td>1</td>
<td>4.42</td>
<td>3.60</td>
</tr>
<tr>
<td>Si5351A-B-GM</td>
<td>Clock</td>
<td>1</td>
<td>2.20</td>
<td>1.65</td>
</tr>
<tr>
<td>FT600Q-T</td>
<td>USB3 FIFO</td>
<td>1</td>
<td>8.22</td>
<td>6.20</td>
</tr>
<tr>
<td>MAX2830ETM+</td>
<td>Transceiver</td>
<td>1</td>
<td>6.96</td>
<td>4.52</td>
</tr>
<tr>
<td>MAX5864ETM+</td>
<td>Hi-Speed AFE</td>
<td>1</td>
<td>8.49</td>
<td>5.42</td>
</tr>
<tr>
<td>MAX8882EUTAQ</td>
<td>Power supply</td>
<td>1</td>
<td>2.71</td>
<td>1.64</td>
</tr>
<tr>
<td>LP5907MFX-3.0</td>
<td>Power supply</td>
<td>1</td>
<td>0.507</td>
<td>0.251</td>
</tr>
<tr>
<td>LP5907MFX-1.2</td>
<td>Power supply</td>
<td>1</td>
<td>0.507</td>
<td>0.251</td>
</tr>
<tr>
<td>TPS795</td>
<td>Power supply</td>
<td>1</td>
<td>2.64</td>
<td>1.57</td>
</tr>
<tr>
<td>ABM11-27.000MHZ</td>
<td>Crystal oscillator</td>
<td>1</td>
<td>1.19</td>
<td>0.845</td>
</tr>
<tr>
<td>GSB343133HR</td>
<td>USB3 Connector</td>
<td>1</td>
<td>0.964</td>
<td>0.788</td>
</tr>
<tr>
<td>2450BL15B100</td>
<td>Balun 1:2</td>
<td>1</td>
<td>0.32</td>
<td>0.11</td>
</tr>
<tr>
<td>LXES15AAA1-133</td>
<td>ESD protection</td>
<td>4</td>
<td>0.136</td>
<td>0.041</td>
</tr>
</tbody>
</table>

Table 9: Bill of materials

*Actual price may vary, shown here for guidance only.

The components used in the board have been selected so that they can be mounted by hand for students or hobbyists. In this case, the final user should only have to buy the components and a blank PCB. Alternatively, this board could also be assembled industrially and sold to the customer as a working device. In this last case, the cost of automated board assembly has to be added.

It is no surprise that the production costs are significantly reduced for mass production, as exemplified in Table 10. In both cases, however, the initial goal of designing an SDR for under 50 euro has been accomplished.
**Budget**

<table>
<thead>
<tr>
<th>Production Cost</th>
<th>1 unit</th>
<th>1000 units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components</td>
<td>39.264</td>
<td>26.886</td>
</tr>
<tr>
<td>PCB</td>
<td>6.01</td>
<td>0.74</td>
</tr>
<tr>
<td>PCB Assembly</td>
<td>2.66</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>45.274</td>
<td>30.286</td>
</tr>
</tbody>
</table>

Table 10: Production costs

*Actual price may vary, shown here for guidance only.

The project can also be turned into a commercial product. In order to make some profit, the finished product could be sold for 50 euro, that would mean to have a benefit of about 20 euro per board sold. Still with this margin, this product would beat the price of the rest of available SDR in the market, being effectively the cheapest device capable of transmitting and receiving.

In order to get an idea of the value of the project, we can estimate the engineering and design costs by making an assumption about what would be the salary of an engineer working on this same project. By counting the amount of hours and summing the roles involved (Table 11)

<table>
<thead>
<tr>
<th>Worker</th>
<th>Role</th>
<th>Salary €/hour</th>
<th>Number of hours</th>
<th>Total €</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matthias</td>
<td>PCB design</td>
<td>12</td>
<td>160</td>
<td>1920</td>
</tr>
<tr>
<td>Matthias</td>
<td>Project management</td>
<td>15</td>
<td>100</td>
<td>1500</td>
</tr>
<tr>
<td>Carsten</td>
<td>FPGA logic design</td>
<td>12</td>
<td>100</td>
<td>1200</td>
</tr>
<tr>
<td>Carsten</td>
<td>Prototype assembling, debugging</td>
<td>11</td>
<td>100</td>
<td>1100</td>
</tr>
<tr>
<td>Carsten</td>
<td>Software development</td>
<td>12</td>
<td>100</td>
<td>1200</td>
</tr>
<tr>
<td>Carsten</td>
<td>Project management</td>
<td>15</td>
<td>100</td>
<td>1500</td>
</tr>
<tr>
<td>Victor</td>
<td>FPGA logic design</td>
<td>12</td>
<td>200</td>
<td>2400</td>
</tr>
<tr>
<td>Victor</td>
<td>Prototype assembling, debugging</td>
<td>11</td>
<td>200</td>
<td>2200</td>
</tr>
<tr>
<td>Victor</td>
<td>Software development</td>
<td>12</td>
<td>100</td>
<td>1200</td>
</tr>
<tr>
<td>Victor</td>
<td>Project management</td>
<td>15</td>
<td>100</td>
<td>1500</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td><strong>1260</strong></td>
<td><strong>15720</strong></td>
</tr>
</tbody>
</table>

Table 11: Engineering cost estimation

From Table 11 we can see, the estimated engineering cost is around 15720 € If the project is turned into a commercial product, and sold with a gain margin of 20 € per unit, a single 1000 unit batch would be sufficient to recover the engineering costs, the costs of the materials and still have some profit, as observed in Figure 29.

However, we have to keep in mind the design is still not completed, and probably the engineering costs in order to get a finished product would be higher.
Figure 29: Cost per unit breakdown
6 Glossary

ADC Analog-to-Digital converter. vii, ix, 2, 7, 10, 12, 15, 21–26, 30, 32, 33, 46, 52, 53

API Application Program Interface. 10, 11, 26, 27

BGA Ball-Grid Array. 2, 33

DAC Digital-to-Analog converter. vii, ix, 2, 7, 10, 12, 15, 17, 21–25, 30, 31, 33, 46, 52, 53

DDR Double Data Rate. 23, 53

FFT Fast Fourier Transform. 30, 32, 46

FIFO First-In, First-Out. v, vii, 10, 11, 17–23, 30, 31, 53

FPGA Field-Programmable Gate Array. v–vii, 2, 5–7, 9–11, 13–23, 28–30, 33, 40, 41, 51

FSM Finite State Machine. vii, 16, 19–23, 53

GPIO General-purpose Input/Output. 11, 12, 17, 40, 41

GUI Graphical User Interface. 26, 30

I2C Inter-Integrated Circuit. vii, 9, 16, 17, 40, 52

IC Integrated Circuit. 1, 10

IQ In-Phase/Quadrature. 23

LCM Least Common Multiple. 9

LNA Low-Noise Amplifier. 32

LO Local Oscillator. 9, 12, 26

LSB Least-Significative Bit. 23

LUT Look-Up Table. 17

MIMO Multiple-In Multiple-Out. 5

MSB Most-Significative Bit. 23

OS Operating System. 26

PCB Printed Circuit Board. 2, 33, 34, 50, 51

PLL Phase-Locked Loop. 9, 11, 16, 25, 26
QFN  Quad-Flat No-Lead. 2, 33

RAM  Random-Access Memory. 10, 11, 18, 20

RF  Radio Frequency. v, vii, 1, 2, 6, 7, 9, 24–26, 30

ROM  Read-Only Memory. 15

RSSI  Received Signal Strength Indicator. 26

RX  Receive. vii, 26, 27, 32

SDR  Software-Defined Radio. vii, 1, 2, 5, 6, 10, 11, 16, 17, 20, 23, 27–29, 33, 34, 40

SMD  Surface Mounted Devices. 2

SNR  Signal-to-Noise Ratio. 24, 30

SPI  Serial Peripheral Interface. viii, 14, 15, 23–26, 28, 40, 41

TX  Transmit. vii, 22, 24, 26, 27, 30

UHD  USRP Hardware Driver. 30, 32

USB  Universal Serial Bus. v, vii, ix, 2, 7, 10–13, 15–17, 19–23, 26, 28, 29, 33, 50, 51

USR P  Universal Software Radio Peripheral. 5, 28–30

VCO  Voltage-Controlled Oscillator. 8

WiFi  IEEE 802.11. 2, 5, 10
7 References


This annex includes the description of procedures and ideas specific to this project whose level of detail would not fit inside the main thesis text.

### 8.1 Programming Si5351 with Raspberry PI

At the beginning of the prototype assembly, the Si5351 clock generator was programmed using the I2C interface of a Raspberry PI model B. This technique was used until the I2C interface was implemented inside the FPGA.

The Si5351 pins SDA and SCL were connected directly to the I2C interface, with both pull-up resistors mounted in the SDR board. The respective pins of the FPGA were left unconfigured.

![Raspberry PI I2C connections](image)

The register map was generated using an utility provided by the manufacturer. This utility allows the user to select clock frequencies and various settings using a GUI and produces a text file describing the contents of the registers. A Python script was developed to read this text file and write the appropriate values through the I2C interface.

### 8.2 Programming ICE40 FPGA with Raspberry PI

During the development stages of this project, the FPGA had to be programmed each time the prototype was turned on. To do so and to transfer the bitmap to the chip, the SPI interface of a Raspberry PI was used. Additionally to the SPI signals, reset and chip select signals were controlled also by Raspberry PI using GPIO pins available on the connector header.
To transfer the bitmap and control the GPIO pins, a shell script had to be run in the Raspberry PI. The script was a modification of [13]

8.3 Raspberry PI in the IceCube environment

After the bitmap file is generated by Ice Cube environment, it has to be transferred to Raspberry PI via Ethernet, and the configuration script must be called to program the FPGA. When debugging the design, having to do all these steps manually is laborious. A method was developed so that Ice Cube automatically transfers the bitmap and programs the chip after compilation is successful.

In the end of the compilation process, Ice Cube calls internally the executable located at `sbt_backend/bin/linux/opt/bitmap` (path relative to the location of `iCECube2` executable). This executable has the task of generating the bitmap file if all the synthesis steps were successful.

The method consists on renaming the original `bitmap` executable to `bitmap_real`. Then, we will create a new executable script called `bitmap` in the same directory. We can put arbitrary code inside this script, and we can exploit the fact that the Ice Cube tool will execute this file in the end of the compilation process to customize the behavior.

The script basically calls the `bitmap_real` original executable first to produce the actual bitmap file, and later it has commands to transfer the generated file and executing the programming script remotely on the Raspberry PI

```
#!/bin/bash
#
# Wrapper script for the bitmap generator utility.
# This gets called automatically by iceCube2 IDE
# in the step of generating bitmap.
#
# With this wrapper, after generating the actual bitmap,
# the upload script is automatically called so the
# binary is programmed to the FPGA
#
```
This script must be called "bitmap" and placed in the >> sbt_backend/bin/linux/opt/ << directory. Give execution permission with chmod +x ./bitmap

The actual bitmap generator program also called "bitmap", must be renamed to "bitmap_real"

Running wrapper script for bitmap programming [SEEMOO]
Generating bitmap file...

Check the exit code of the bitmap generator, if it was successful, then proceed to program the FPGA

if [ $? -ne 0 ]; then
    Real bitmap generator failed! Exiting without programming!
    exit 1
else
    Obtain path of the bitmap file
    BITMAP_DIR=0

    while test $# -gt 0; do
        if [ "$1" == "--outdir" ]; then
            BITMAP_DIR="$2"
            shift
            shift
        else
            shift
        fi
    done

    BITMAP_FILE="$(ls $BITMAP_DIR/*.bin)"
    echo "Bitmap File: $BITMAP_FILE"
    echo "Real bitmap generator succeeded. Uploading to Raspberry Pi..."

    BIN_FILE=$BITMAP_FILE
    RPI_LOGIN=pi@192.168.2.80
    RPI_DEST=/home/pi

    scp $BIN_FILE $RPI_LOGIN:$RPI_DEST/seemoo_bitmap.bin
    echo "Upload successful. Executing remote programming script..."
    ssh $RPI_LOGIN "sudo $RPI_DEST/ice40.sh $RPI_DEST/seemoo_bitmap.bin"
    exit 0
fi
8.4 Reading and writing to the FT600

This is the C code of the developed driver to encapsulate the FT600 API and provide access using named pipes. Only the code for the reader, writer and measure threads is provided, to demonstrate relevant use of the FT600 API

**Writer thread**

```c
static void write_test(FT_HANDLE handle) {
    unique_ptr<uint8_t[]> buf(new uint8_t[wbuf_len]);
    unsigned int channel = 0;

    printf("Please open file %s for writing...\n", fifo_name_tx);
    file_tx= open(fifo_name_tx, O_RDONLY);
    printf("TX pipe is OPEN\n");

    while (!do_exit) {
        ULONG to_read = wbuf_len;
        long ok_read;
        ULONG to_write;
        ULONG ok_write;
        unsigned char * offset;

        ok_read = read(file_tx, buf.get(), to_read);
        if (ok_read < 0) {
            printf("TX: Broken pipe\n");
            close(file_tx);
            printf("Please open file %s for writing...\n", fifo_name_tx);
            file_tx= open(fifo_name_tx, O_RDONLY);
            printf("TX pipe is OPEN\n");
            continue;
        }

        to_write = ok_read;
        offset = buf.get();

        while (to_write > 0) {
            if (FT_OK != FT_WritePipeEx(
                    handle,
                    channel,
                    offset,
                    to_write,
                    &ok_write,
                    1000)) {
                printf("TX: FT600 Error! \n");
                close(file_tx);
                printf("Please open file %s for writing...\n", fifo_name_tx);
            }
        }
    }
}
```
8.4 Reading and writing to the FT600

```c
file_tx = open(fifo_name_tx, O_RDONLY);
printf("TX pipe is OPEN\n");
continue;
}
```

```c
tx_count += ok_write;
offset += ok_write;
to_write -= ok_write;
}
}
printf("Write test stopped\n");
```

Reader thread

```c
static void read_test(FT_HANDLE handle) {
    unique_ptr<uint8_t[]> buf(new uint8_t[rbuf_len]);
    unsigned int channel = 0;

    printf("Please open file %s for reading...\n", fifo_name_rx);
    file_rx = open(fifo_name_rx, O_WRONLY);
    printf("RX pipe is OPEN\n");

    while (!do_exit) {
        ULONG to_read = rbuf_len;
        ULONG ok_read;
        ULONG to_write;
        long ok_write;
        unsigned char * offset;

        if (FT_OK != FT_ReadPipeEx(
            handle,
            channel,
            buf.get(),
            to_read,
            &ok_read,
            1000)) {
            printf("RX: FT600 Error!\n");
            close(file_rx);
            printf("Please open file %s for reading...\n", fifo_name_rx);
            file_rx = open(fifo_name_rx, O_WRONLY);
            printf("RX pipe is OPEN\n");
            continue;
        }

to_write = ok_read;
offset = buf.get();
```
8.4 Reading and writing to the FT600

```c
while (to_write > 0) {

    ok_write = write(file_rx, offset, to_write);

    if (ok_write < 0) {
        printf("RX: Broken pipe\n");
        close(file_rx);
        printf("Please open file %s for reading...\n", fifo_name_rx);
        file_rx= open(fifo_name_rx, O_WRONLY);
        printf("RX pipe is OPEN\n");
        continue;
    }

    rx_count += ok_write;
    offset += ok_write;
    to_write -= ok_write;
}

printf("Read test stopped\r\n");
```

Measure thread

```c
static void show_throughput(FT_HANDLE handle) {
    auto next = chrono::steady_clock::now() + chrono::seconds(1);
    (void)handle;

    while (!do_exit) {
        this_thread::sleep_until(next);
        next += chrono::seconds(1);

        int tx = tx_count.exchange(0);
        int rx = rx_count.exchange(0);

        if (tx > 1000) {
            printf("TX: %d KiB/s\t", tx/1000);
        } else {
            printf("TX: %d B/s\t", tx);
        }

        if (rx > 1000) {
            printf("RX: %d KiB/s\n", rx/1000);
        } else {
            printf("RX: %d B/s\n", rx);
        }
    }
```

45
8.5 Scripts for transmitting and receiving data

During the tests, the open source tool Octave (which uses the same language as MATLAB) was used to transmit and receive signals with SEEMOO-SDR using the named pipes interface. Two scripts were developed: The first one is able to generate a sine wave and reduce the number of bits to 8, to send them to the DAC. This script applies dithering to mitigate the effects of resampling. The second script reads samples from the ADC, calculates the FFT and shows the received signal in time and frequency domains.

generate_raw.m

```matlab
\%
\% Generate test sine wave raw files for SDR.
\%
\% Victor Pecanins <vpecanins@gmail.com>
\% SEEMOO SDR 2016
\%
\% To write to FIFO: cat signal.raw > tx
\%
\% Tested with octave
\%

clear all;
close all;

% Script options
% Show plots of generated signals
show_plot=1;

% Signal type [0=sin, 1=2sin, 2=sweep]
signal_type=0;

% Signal parameters here
Fs = 22000000; % Sample rate in Hz
L = 0.002; % Duration of the signal in seconds

% File to (over)write
filename = 'signal.raw';

% Number of samples
count = Fs*L;

% Time values
t = linspace(0, L, count);
```
8.5 Scripts for transmitting and receiving data

% Generate signals here
% Signals normalized between (-1, 1);
% 150 kHz is the max frequency where the input
% impedance of oscilloscope is 70K ohms (datasheet value)

if (signal_type==0)
  Fi = 550000; % Frequency of sine waves in Hz
  Fq = 550000; % For I and Q channels
  signal_i_n = sin(2*pi*Fi*t);
  signal_q_n = sin(2*pi*Fq*t);
  %signal_i_n = zeros(size(t));
  %signal_q_n = zeros(size(t));
elseif (signal_type==1)
  Fi = 150000;
  Fq = 150000;
  Fq2 = 3000000;
  signal_i_n = 1.0*(sin(2*pi*Fi*t) + sin(2*pi*Fq*t))/2;
  signal_q_n = 1.0*(sin(2*pi*Fq*t) + sin(2*pi*Fq2*t))/2;
elseif (signal_type==2)
% Sweep from Fa to Fb
  Fa = 100;
  Fb = 12000000;
  phi = Fa.*t + (Fb-Fa)/(2*L)*t.^2;
  signal_i_n = sin(2*pi*phi);
  signal_q_n = zeros(size(signal_i_n));
end

% Dithering noise
% This added noise makes discretization noise uniform
% aka minimize artifact tones generated by discretization
% Change 1 to 0 to see the effects in the spectrum.
  dither_noise = 1.*(rand(size(signal_i_n)) - 0.5);
  signal_i_n = signal_i_n + dither_noise/128;
  signal_q_n = signal_q_n + dither_noise/128;

% Signals in uint8 format
8.5 Scripts for transmitting and receiving data

```matlab
signal_i = 128 + round(128*signal_i_n);
signal_q = 128 + round(128*signal_q_n);

% Plot time series
% Only plot first samples
plot_int = 1:100;

if (show_plot == 1)
    % Plot time series, I and Q at the same graph
    figure;
    subplot(2,1,1);
    plot((plot_int), [signal_i(plot_int); signal_q(plot_int)], 'o-');
    axis([min((plot_int)),max((plot_int)), 0, 256]);
    title('Generated signal');

    % Compute required size for FFT
    Nfft = 2^ (ceil(log2(count)));

    % Compute FFTs
    % Remove offset of uint8, but preserve rounding artifacts
    fft_i = fft(signal_i-128, Nfft)/Nfft;
    fft_q = fft(signal_q-128, Nfft)/Nfft;

    % Frequency axis
    freqs = linspace(0, Fs, Nfft);

    % Plot I/Q Spectrum
    subplot(2,1,2);
    plot((freqs), 20*log10(abs([fft_i; fft_q])));
    axis([0, Fs, -100, 50]);
end;

% Interlace I/Q samples for writing
signal_out = zeros(2*count, 1);
signal_out(1:2:end) = signal_i;
signal_out(2:2:end) = signal_q;

% Remove possible escape sequences
signal_out(signal_out >= 255) = 254;

% Open binary file for writing
fid = fopen (filename, 'w');

% Write samples
count_w = fwrite(fid, signal_out, 'uint8');
printf('Wrote %d bytes.
', count_w);
```
8.5 Scripts for transmitting and receiving data

```matlab
% Close file
close(fid);

% Send file through fifo
$system(['cat signal.raw > tx']);
```

**show_raw.m**

```matlab

% Plot time series and spectrum from raw files
% Victor Pecanins <vpecanins@gmail.com>
% SEEMOO SDR 2016
% To read 1024 bytes from FIFO:
% dd if=rx skip=0 count=1024 bs=1 > signal.raw
% Tested with octave

clear all;
close all;

filename = 'signal2.raw';

system('dd if=rx skip=0 count=8 bs=4096 iflag=fullblock > signal2.raw');
figure;

while (1)
    % Open binary file
    fid = fopen(filename, 'r');

    % Read sample data
    [val, count] = fread(fid, Inf, 'uint8');

    % Separate odd and even samples
    signal_i = val(1:2:end);
    signal_q = val(2:2:end);

    % Only plot first samples
    plot_int = 1:500;
    %plot_int = [1:length(signal_i)];

    % Plot time series, I and Q at the same graph
    subplot(2,1,1);
    plot((plot_int), [signal_i(plot_int), signal_q(plot_int)], 'o-');
```
8.6 Proposed hardware improvements

During the development of the prototype, some bugs were found in the PCB design, which should be corrected in order to make the board work properly. In further development steps of this project, if a new PCB is designed, these modifications should be taken into account. Not all of these modifications are bugs, also there are some ideas for improving the current performance.

**VBUS connection in USB Interface**

In the original schematic, there is a resistive voltage divider between the power supply pin of the USB connector and the VBUS pin of the FT600 chip. This was found to be unrequired and should be replaced by a short, connecting directly the USB supply to VBUS.
8.6 Proposed hardware improvements

VCCIO power supply

The net VCCIO connects to the 3.3 power supply of the FT600 USB Interface, two IO banks of the FPGA and the clock generator. Only the FT600 is expected to draw 185 mA when working at SuperSpeed mode, while the maximum current the MAX8882 is only capable of delivering 160 mA. Although the prototype worked with this setup, it is advised to replace the regulator with one that can provide more output current.

USB Connector mechanical strength

In the original PCB the USB 3 connector shield is soldered to large copper pads to fix it on the board and preventing it from move when connecting or disconnecting the cable. However, the pads connected to the shield are isolated, being only copper squares in the board. When connecting and disconnecting a lot of times the cable, it was found that these squares detached from the PCB, due to the force applied.

In order to make the next revision design more reliable, it is advised not to have separate isolated squares, but to solder the shield directly to a big, continuous ground plane. Furthermore, the area beneath the connector could have a large number of vias connecting it.
8.6 Proposed hardware improvements

to a ground plane in opposite side of the board, to avoid the copper layer being detached due to mechanical force.

Figure 34: Isolated square pads detach from the board due to applied force

Test connectors for baseband signals

The original design has U.FL connectors in order to easily attach an oscilloscope or spectrum analyzer to the antenna line, and to some clock signals as well. In the following designs, it would be useful to include such connectors also to the ADC/DAC analog lines, to be able to see the baseband signals with an oscilloscope.

This could help debugging the problem of bandwidth limitation and distortion, and would also make the board more extensible, allowing the users to attach a custom transceiver for other frequency bands.

Test connectors for digital signals

The original PCB design includes specific places on the FT600 Parallel bus and the MAX5864 ADC and DAC digital pins where the tracks are not covered by a silk mask. The purpose of this is to solder tiny cables on the board to be able to monitor the signals with a logic analyzer. However, even if this is possible, it was found to be a very laborious task to solder single copper wires on the test pads, because the separation is less than half a milimiter.

In future designs, it may be interesting to change the shape of the tracks to connect a two-row pin header, with interlacing signals as has been done in the same original board for the I2C debug connector.
8.6 Proposed hardware improvements

Figure 35: Interlaced tracks to solder a pin header.
The same idea could be applied to the digital buses

ADC/DAC Clock phase offset

In the original design, the clock controlling the Frontend side of the FIFOs and the ADC/DAC FSMs, and the clock input of the ADC/DAC, are two different clock outputs of the clock generator, CLK1 and CLK7 respectively. The purpose of this is to have two clocks with the same frequency and different amplitude. It is also required that these two clocks have the same phase, or at least they must have a known phase offset.

The problem is that Si5351 can only control the relative phase of clocks CLK1 to CLK5. Experimental tests concluded that CLK6 and CLK7 have a random phase offset, different each time the chip is powered on. This is a big problem, because due to DDR, depending on the phase offset, I and Q signals may be switched randomly.

To solve this problem, both devices should be connected to the same clock source. In the prototype, the clock input of MAX8564 was connected using a flying wire to CLK1, showing the problem of I and Q channel switching was solved. However, this modification did not solve the problem of band limitation and distortion on the DAC, suggesting that this problem has the origin somewhere at the analog output of the DAC.

Alternatively, another clock signal could be used for the ADC/DAC, but this output must be one of the clocks between CLK1 and CLK5, which have control on the phase offset.