**ETH**

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Network-enabled digital lock box for trapped-ion experiments

QuanTech Workshop

G. Bisson, M. Glantschnig, D. Hagmann, P. Tirler

Sunday 15th May, 2022

**Abstract**

Experiments with trapped ions require lasers which are stable in frequency, amplitude and phase. To achieve this, one commonly uses active feedback, i.e. one measures the deviation of a quantity from its desired value and feeds this error back to the system. To tune this feedback loop, one commonly uses a PID (Proportional, Integral, Derivative) controller, which is also referred to as a lock box.

The aim of this QuanTech project was to develop a replacement for the digital lock box *EVIL* that had been used at the Trapped Ion Quantum Information group over the past decade and which has become deprecated since. We evaluated different options and eventually opted for a semi-custom solution based on the commercial mixed signal Red Pitaya STEMlab 125-14 board. Special care was taken to ensure backward compatibility with the previous solution, both hardware and software wise.

To this end, we designed a custom printed circuit board which we called *Bichannel Lockbox On One Device (BLOOD)* that can be deployed in the same Eurocard racks that are used for the EVIL, i.e. has the same form factor and power supplies. This PCB also comes with the possibility to change the gain and offset of the analog inputs and outputs digitally. It runs custom software, for which we combined the open-source *PyRPL* project, a software/firmware stack designed for controlling AMO experiments on the Red Pitaya with Python, with *DEVIL*, the software currently used to control the EVILs in the TIQI group.

During the course of this project we built a prototype of the BLOOD and we verified that the soft- and hardware work together in the way we intended. However, to make it fully usable in the lab, some further improvements are needed.

## Acknowledgements

We want to thank Prof. Jonathan Home and the whole Trapped Ion Quantum Information (TIQI) group at ETH for the opportunity we had to join their group for this project. In particular, we want to thank our supervisors Martin Stadler, Nick Schwegler and Vlad Negnevitsky for their close support in planning and execution throughout the project, without which this would not have been possible. We also owe thanks to all other group members who helped us throughout the last semester, including, among others, Peter Clements, Matt Grau, Ilia Sergachev and Robin Oswald.

# Contents


**1 Introduction** **5**
- 1.1 Control in Trapped-Ion experiments . . . 5
  - 1.1.1 The PDH lock . . . 5
- 1.2 State of the Art . . . 6
  - 1.2.1 Hardware . . . 6
  - 1.2.2 Software . . . 6
- 1.3 Reason for upgrade . . . 6
- 1.4 Project goals . . . 6

**2 Execution** **8**
- 2.1 The Printed Circuit Board . . . 8
  - 2.1.1 Schematics . . . 8
  - 2.1.2 Layout . . . 13
  - 2.1.3 Soldering . . . 13
- 2.2 Software/Firmware . . . 14
  - 2.2.1 Testing of PyRPL . . . 14
  - 2.2.2 Software-Firmware Stack . . . 17
  - 2.2.3 Adaptation of the PyRPL Firmware . . . 17
  - 2.2.4 Modified `pyrpl_tiqi` . . . 18
  - 2.2.5 Adaptation of the DEVIL server & client . . . 19
  - 2.2.6 Operating System on STEMlab . . . 20

**3 Results** **22**
- 3.1 Assembly . . . 22
- 3.2 PCB Tests . . . 22
  - 3.2.1 Input & Output Sections . . . 23
  - 3.2.2 Communication between RedPitaya and ICs on the PCB . . . 24
- 3.3 Open-loop transfer function . . . 24
- 3.4 PDH lock of laser to a cavity . . . 25
- 3.5 Outlook . . . 27
  - 3.5.1 Open issues . . . 27
  - 3.5.2 Conclusion . . . 27

**References** **28**

**A Specification Table** **30**

**B Cost-Calculation** **31**

**C Repositories** **31**

**D Verilog module for gain & offset control** **32**

**E Pipelining of PyRPL RTL code** **34**

**F Handling the BLOOD Server** **34**

**G Documentation for BLOOD-Devil client** **35**
- G.1 Integration of PyRPL client into Devil client . . . 35
- G.2 The `BloodRegister` class . . . 36
- G.3 The `BloodChannel` class . . . 37
- G.4 Initialization of Server after power-up . . . 37
- G.5 The `BloodServer` class . . . 38

G.6 Channel Detection process . . . 38
G.7 Changes in the GUI . . . 38
G.8 transferfunctionplot.py . . . 38
G.9 Server only registers used between client and server . . . 38
G.10 Communication between server `DummyChannel` and client `BloodChannel` . . . 40

**H Detailed list of proposed code changes and missing features** **42**
H.1 General Features . . . 42
H.1.1 Entire System . . . 42
H.1.2 Server . . . 42
H.1.3 Client . . . 42
H.2 Fixes for the Client . . . 42
H.2.1 `main.py` . . . 43
H.2.2 `bloodserver.py` . . . 43
H.2.3 `bloodchannel.py` . . . 43
H.2.4 `customwidgets.py` . . . 44
H.3 Server . . . 44
H.3.1 `DummyChannel.cpp` . . . 44
H.4 Bitstream . . . 44
H.5 `pyrpl_tiqi` . . . 44

**I Modifications that make the BloodClient incompatible with the EVIL** **45**

# 1 Introduction

## 1.1 Control in Trapped-Ion experiments

Trapped Ions are one of the leading platforms in the endeavour to implement different forms of Quantum Computing, from Universal QC to Quantum Simulation. The logic states are encoded in the energy-levels of single ions, trapped alone or with a manageable number of others in an electric (or sometimes also magnetic) field. The electronic energy levels of ions couple to the electromagnetic field surrounding the atom, and this coupling is used to prepare, manipulate and read out the state of an ion qubit. Practically, this means that the qubits can be controlled using laser light. Depending on the exact operation, the frequency, intensity and phase of this laser must be very precise, and feedback control is used to ensure any noise or drift is mitigated. Outside factors such as temperature and pressure fluctuations from sound waves negatively affect the laser's stability. As the linewidth of some energy levels used in trapped ion experiments can be very narrow, frequency stability is crucial, and additional layers of control have to be introduced to ensure this stability can be guaranteed even in a lab environment that includes temperature drifts or researchers working and moving in the vicinity of the experiment.

### 1.1.1 The PDH lock

As an example, we discuss a way to stabilize the frequency of a laser making use of active feedback control. The basic idea is to measure the frequency difference between the laser and a very stable reference frequency. A common choice for this is the resonance frequency of a high-finesse cavity, which can be engineered to be sufficiently stable provided it is situated in a temperature controlled environment isolated from sound. A laser incident onto the cavity will only experience minimal reflection if its frequency matches a multiple of the resonance frequency. By measuring the reflection one can implement feedback control, changing the laser frequency until it reaches resonance. As the intensity of the reflected light is symmetric around the minimum, its phase must be used to determine the sign of the frequency error. This can practically be extracted by creating two side bands, symmetrically around the resonance. Their combined reflection signal will now depend on their relative phase shift, and thus also give away the direction of the detuning, which makes feedback control possible. This technique is called, after its inventors, the Pound-Drever-Hall (PDH) technique. [8] A schematic setup of an experimental implementation can be seen in Figure 1.

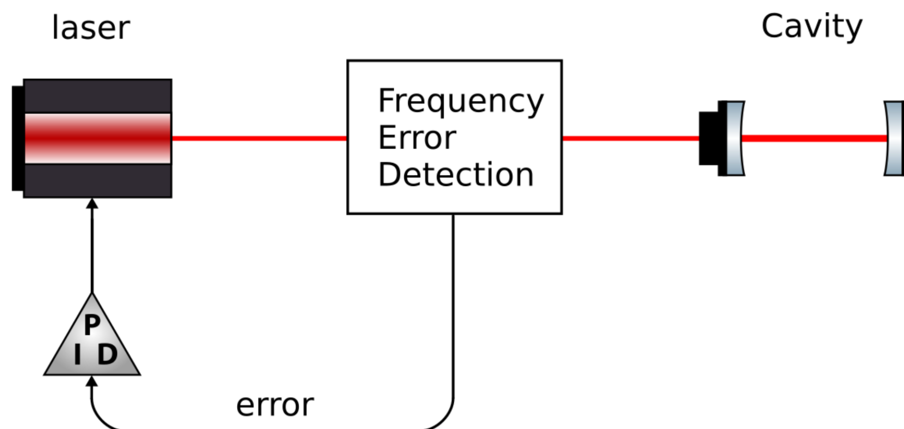



Figure 1: Example schematic of a PDH lock. In this case, the frequency of the laser is modified by adjusting the current through the laser diode or its temperature, dependent on the output of a PID controller that takes the demodulated reflection as an input.

## 1.2 State of the Art

### 1.2.1 Hardware

In the Trapped Ion Quantum Information (TIQI) lab, almost all tasks related to feedback based laser stabilization are performed using the "Electronically variable interactive lockbox", or "EVIL". It was designed by Ludwig de Clercq and Vlad Negnevitsky around 2012. It consists of a custom PCB with two analog input and output channels and the commercial Papilio One 500K FPGA board [3]. The signal processing is done digitally within the FPGA using a firmware which was also developed specifically for this device. The EVIL needs $\pm 15$ V and GND supplies and its enclosure is designed to fit standard 19' racks.

A more detailed list of specifications can be found in Appendix A, together with a comparison of the device we have designed during this project.

### 1.2.2 Software

The current EVIL boards are controlled via a server that runs on a Raspberry Pi which is connected via USB to the EVIL boards. This server announces itself to a client running on a PC in the lab, from which the experiment is then controlled. Written by David Nadlinger [13] in 2015, this client-server setup supports a multi-client logic, that allows for multiple clients to connect to multiple servers simultaneously. Any parameter changes done by one client are synchronized to all others in real-time.
The client is written in Python and can be used directly from source or via a very simple executable for Windows. It comes with a simple GUI for experiment control, relying on the Python binding *PyQT* [18] of the Qt framework [21].
The server on the other hand is written in C++, and uses the *boost.asio* library [2] for asynchronous operations management. Communication over the network is implemented using the ZeroMQ library [26].

## 1.3 Reason for upgrade

The EVIL has been successfully used for many projects since it was introduced in 2015, but has become outdated since: The Papilio One 500K, which carries the FPGA, is no longer available. A total redesign of the EVIL would be necessary because the whole design is tailored around this part.

This also opens up the opportunity to improve upon some shortcomings of the EVIL. The offset and gain of the analog input stage can be adjusted by potentiometers on the PCB which are only reachable by unmounting the EVIL from the rack. Further points of improvement are the low 10-bit resolution of the ADC in the input stage followed by the few DSP units available on the Xilinx XC3S500E FPGA. The EVIL also only has a USB interface. To control the EVIL remotely over the Lab-network it is connected via USB to a Raspberry Pi which is then connected to the lab network [14]. It would be favorable to omit the Raspberry Pi by directly running the server on the EVIL. This requires a CPU which is able to run Linux.

Xilinx' Zynq SoC devices meet these demands, as they feature both ARM processing cores, networking interfaces as well as a much larger programmable logic with more DSP units on one chip.

## 1.4 Project goals

Drawing from years of experience using the DEVIL in the TIQI lab, our supervisors Martin Stadler, Nick Schwegler and Vlad Negnevitsky helped us to set the following goals:

- A system on chip with integrated FPGA and CPU which is capable of running Linux and the server. This approach provides a much simpler interface between FPGA and CPU while improving latency and throughput. This would most probably mean using a Xilinx Zynq chip as the central element of the lock box.
- A direct Ethernet connection, so that the server can communicate directly with the lab network.
- Clocking the ADCs with an crystal oscillator clock instead of an FPGA generated clock would lead to a better sampling and signal quality.

- A control loop bandwidth comparable to the old EVILs, which requires a latency below 1 µs, with a target of 400 ns
- An input voltage range from 200 mV to 6 V and an output range from 2 to 10 V.
- The new device should be rack-mountable and require the same power supplies as the EVIL.
- Digitally controlled amplifier gains and offset voltages, adjustable directly via the control software (the client).

To match these expectations, we decided to base our solution on the readily available RedPitaya STEMlab 125-14 multipurpose boards [23], and design a custom carrier PCB around it to match the input/output specifications and add the analog gain and offset control. The STEMlab features a 14-bit ADC and DAC, a Xilinx Zynq 7010 SoC and provides most of the features we require, including Ethernet connectivity. The custom PCB can then be designed such that all other goals are met. The solution we went for was inspired by a lockbox designed by the 'Atoms - Photons - Quanta' research group at TU Darmstadt [17]. Starting from their design we modified it using multiplexers and digital potentiometers to allow for controlling the analog gain and offset using the multi-purpose I/O pins on the extension connector of the RedPitaya.

Sticking with the current naming tradition within the TIQI group, we propose to call our device **BLOOD** - **B**ichannel **L**ockbox **O**n **O**ne **D**evice. We also came up with a logo for the BLOOD (Fig. 2).

Having decided for the type of solution we want to propose early in the project, most of the work will go into designing the carrier board, and writing the software and firmware necessary to provide the required features.

Figure 2: The official logo of the BLOOD.

# 2 Execution

## 2.1 The Printed Circuit Board

In this subsection we present the design of the printed circuit board. Firstly, the schematics will be illustrated, then the layout and finally the soldering process.

### 2.1.1 Schematics

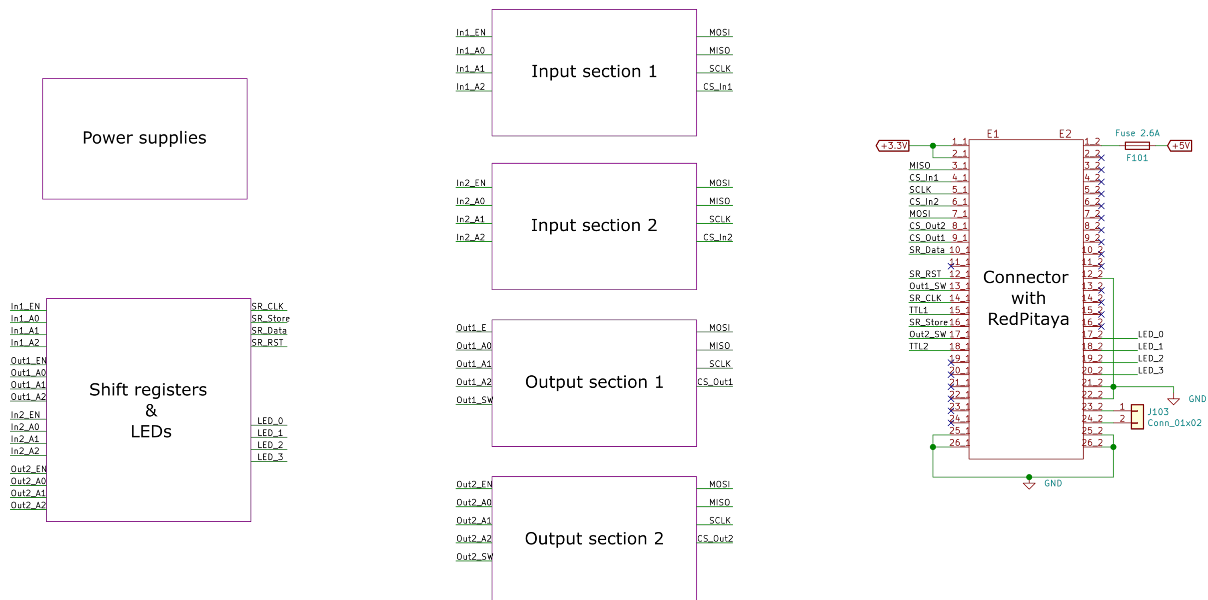



Figure 3: Main blocks of the schematics and their interconnections, adapted from the main page of the KiCad schematics.

Our schematics are based on the open source project *RedPitaya-IntStab* [24] which uses KiCad, a CAD software for drawing schematics and layouting PCBs.
In Figure 3 you can see the main blocks that compose our design:

Power supplies: The block contains the connections to the external power supply which provides ±15 V and ground. It also contains the voltage converter circuits to generate the required voltages for the active components on the carries.

Connector with RedPitaya: On the right side of Fig. 3 there are all the pins that come from the two extension connectors (E1 and E2) of the RedPitaya. These pins have a few different functions. First of all, the RedPitaya can be powered by applying 5 V to pin 1 of E2. 3.3V are provided by the RedPitaya through pin 1 and 2 of E1. The pins SCLK, MISO, MOSI, CS_In1,2 and CS_Out1,2 are used to communicate with the digital potentiometers in the input and output sections. The pins SR_Data, SR_RST, SR_CLK, SR_Store are used to control the two 8-bits shift registers in the "Shift registers & LEDs" block. Out1_SW and Out2_SW are used to control the output switches that enable and disable the two outputs. TTL1 and TTL2 are intended to be used as trigger inputs or outputs. Lastly on E2 LED_0-3 are used to control the 4 LEDs in the "Shift registers & LEDs" block.

Input and Output Sections: In the centre of figure 3 there are four blocks, two are input blocks and two are output blocks. The input blocks contain the components that act on the input signals before they enter the RedPitaya. The output blocks contain the components that condition the signals that come out of the RedPitaya.

Shift Registers & LEDs: This last block serves two purposes:

- Fan out 4 signals to 16 using two 8-bit shift registers inside that take as inputs the SR_Data, SR_RST, SR_CLK, SR_Store signals from the RedPitaya and output 16 signals (all the signals on the left side of the block in figure 3) which are used to control the 4 multiplexers, one in every input/output block.
- Provide visual feedback in the lab using 4 LEDs which are visible from the front panel of the BLOOD and can be turned ON and OFF through the signals LED_0-3 coming from the RedPitaya.

**Power Supply section**

As written above, in this section of the schematics there are the components that convert the +/-15V provided to the PCB from the laboratory into the voltages necessary for the other components of the PCB. Figure 4 shows a simplified version of the KiCad schematics.

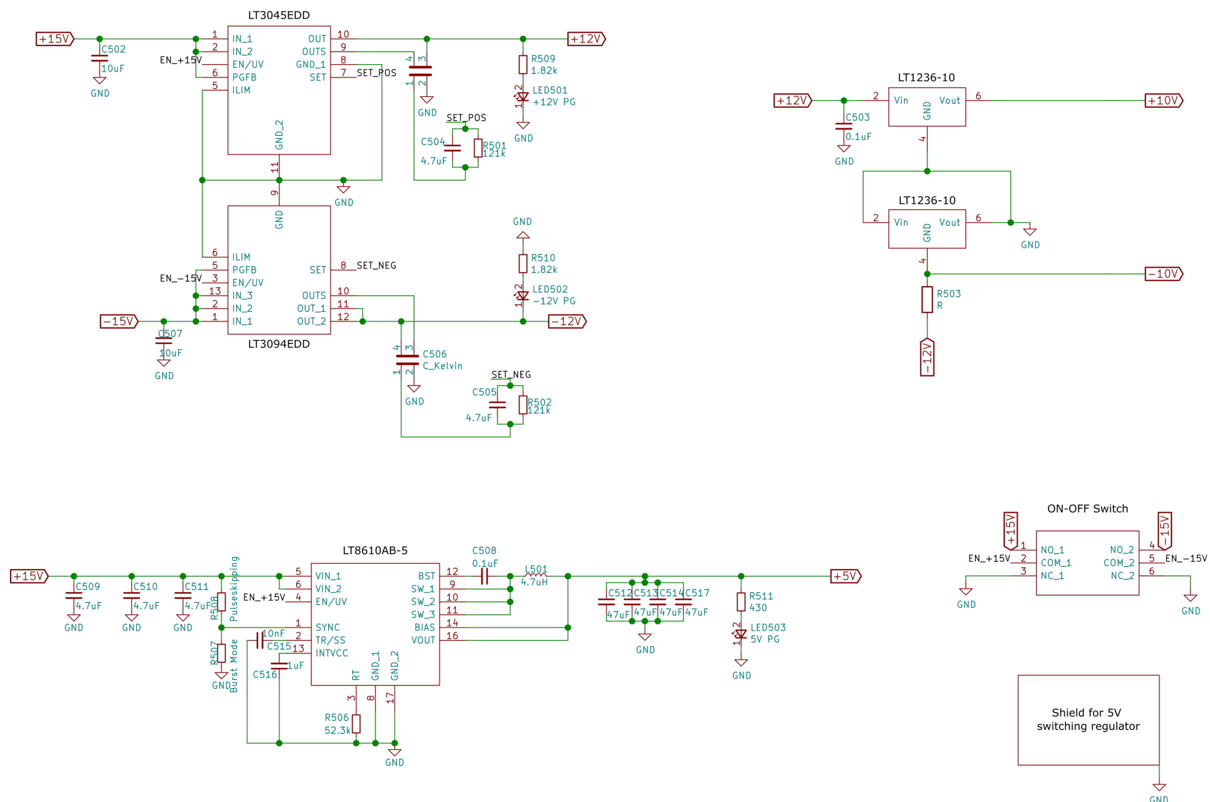



Figure 4: Schematics of the power supplies, adapted from the KiCad schematics.

Starting from the top left of the figure, the two components LT3045EDD and LT3094EDD are high performance low dropout linear regulators with ultralow noise and ultrahigh power supply rejection ratio (PSRR) architecture. The output of these devices is set to +/-12V and they can supply up to 500mA each [6]. These voltages are needed for supplying the OpAmps, digipots and multiplexers in the input and output sections.
On the bottom left of figure 4, the component LT8610AB-5 is a compact and high efficiency synchronous step-down switching regulator that has high maximum output currents of 3.5A [7]. This step down regulator provides 5V for powering the RedPitaya through the connector E2 and the LEDs that are on the PCB. In order to reduce the output ripple to less than 10mV we added four 47uF capacitors at the output. The switching frequency has been set to 800KHz through the resistor R506.
On the top right of the figure, the two LT1236-10 are precision references that combine ultralow drift and noise and high output accuracy [5]. The maximum output current that they can source and sink is up to 10mA. The low drift and noise and high output accuracy are important properties for these components because the +/-10V they supply is used by the digital potentiometers in the input/output sections to adjust the voltage offset. The voltage offset needs to be very stable in time for the PID controller to work

effectively.
At the bottom right of figure 4 there is the ON-OFF switch which acts on all the power supply chips at the same time. It turns them on by connecting the respective enable pins to the input voltage (+/-15V) and off by connection the enable pins to GND. The last component is a metal shield which covers the 5V switching regulator to reduce the switching noise that leaks out.
As a last remark we added 3 LEDs respectively at the +5V and +/-12V outputs which indicate whether the respective chip is ON or OFF.

### Input Section

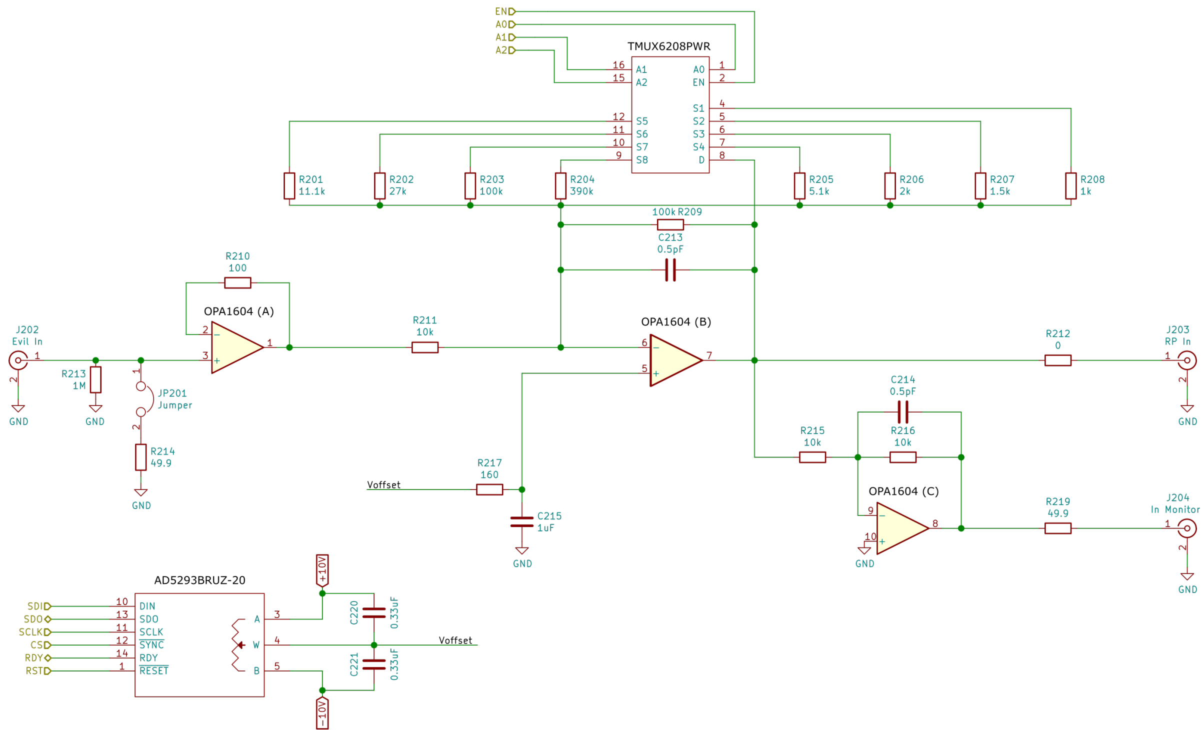


Figure 5: Schematics of the input section, adapted from the KiCad schematics.

On the device there are two analog signal inputs. Each one of these inputs goes through one of the two identical input sections on the PCB. In figure 5 there are the schematics which are a simplified version of the ones from KiCad.

The main features that we implemented in the input section are:

- Variable gain
- Variable offset
- The input impedance is 1 MΩ by default and can be changed to 50 Ω with a jumper (JP201)
- The analog input signal of the RedPitaya can be monitored from J204 without disturbing it

Variable gain: This feature is achieved thanks to the precision multiplexer TMUX6208PWR [11]. This component allows to select one of the eight different resistors (R201-R208) on the feedback path of the operational amplifier OPA1604 (B) which is in inverting configuration. The resistor is selected through the signals A2, A1, A0 which correspond to the binary address of the desired resistor, starting from 000 for the first one. For example to select the resistor number 3 (A2,A1,A0)=(0,1,1). The signals EN (EN enables or disables the MUX), A2, A1, A0 come from one of the two shift registers (see 2.1.1) which are in turn controlled by the RedPitaya.

| $R_{mux}$ [kΩ] | $G$ |
|---:|---:|
| 1 | 0.1 |
| 1.5 | 0.15 |
| 2 | 0.2 |
| 5.1 | 0.49 |
| 11.1 | 1 |
| 27 | 2.13 |
| 100 | 5 |
| 390 | 7.96 |
| Mux off | 10 |

Table 1: Table of all possible values that $G$ can have, according to Eq. 2.1.

Variable offset: This feature is achieved thanks to the digital potentiometer AD5293BRUZ-20 [4]. The resistance between the pins A and B of this component is 20 kΩ. The wiper can be adjusted in 1024 different positions between A and B. By inputting the voltage references +/-10V in A and B, the pin W can be set to a voltage that ranges from -10V to +10V. The position of the wiper is sent to the digipot through an SPI interface. The offset voltage coming from W (wiper) is inputted to the + pin of the OPAMP OPA1604 (B).
In this configuration the signal coming out of this opamp (B) is:

$$V_{out} = -GV_{in} + (G+1)V_{offset} \tag{2.1}$$

where $G = \frac{R_{feedback}}{R211}$ with $R_{feedback}$ equal to R209 in parallel with the resistor selected by the multiplexer. Tab 1 shows all the possible gain values that can be selected.

We simulated the whole circuit in PSpice for TI. We had to use a SPICE simulator from TI because a SPICE model for the OPA1602/1604 was only available from within Texas Instruments software. The TMUX6208PWR does not come with a SPICE model, so we used the model of an almost equivalent multiplexer, the ADG1408. The capacitor C213 was chosen such that the transfer function of the amplifier stage has maximum bandwidth and a flat response. We also used the PSpice simulation to determine the power consumption of the whole circuit.

**Output Section**

As for the inputs, on the PCB there are two identical output sections. See figure 6 for the schematics which are an adaptation of the one from KiCad.
The working principles of the output sections are very similar to the ones of the inputs. The variable gain and offset are obtained in the same way. Other features implemented in these sections are:

- The output can be enabled or disabled from the RedPitaya and is disabled by default. This allows to enable the output only if the user is sure that the output voltage is in the accepted range of the equipment that is connected to the device.
- It is possible to select the voltage range of the output as only positive or only negative through a DIP-Switch.

Output enable and disable: this feature is provided by the use of the PhotoMOS switch AQY221N3MY [16] in series with the output. Pin 1 of this component is controlled by the RedPitaya board and, when it is high, it closes pins 3 and 4, enabling the output.

Voltage range and DIP-Switch: Two diodes (D401 and D402) connected between the output (J403) and GND can be selected with a DIP-Switch, optionally limiting the output to only positive or negative voltages. Furthermore, the DIP-Switch can also be used to bypass the PhotoMOS in case it is not working correctly.
As a last remark the output impedance of the device is approximately 50Ω.

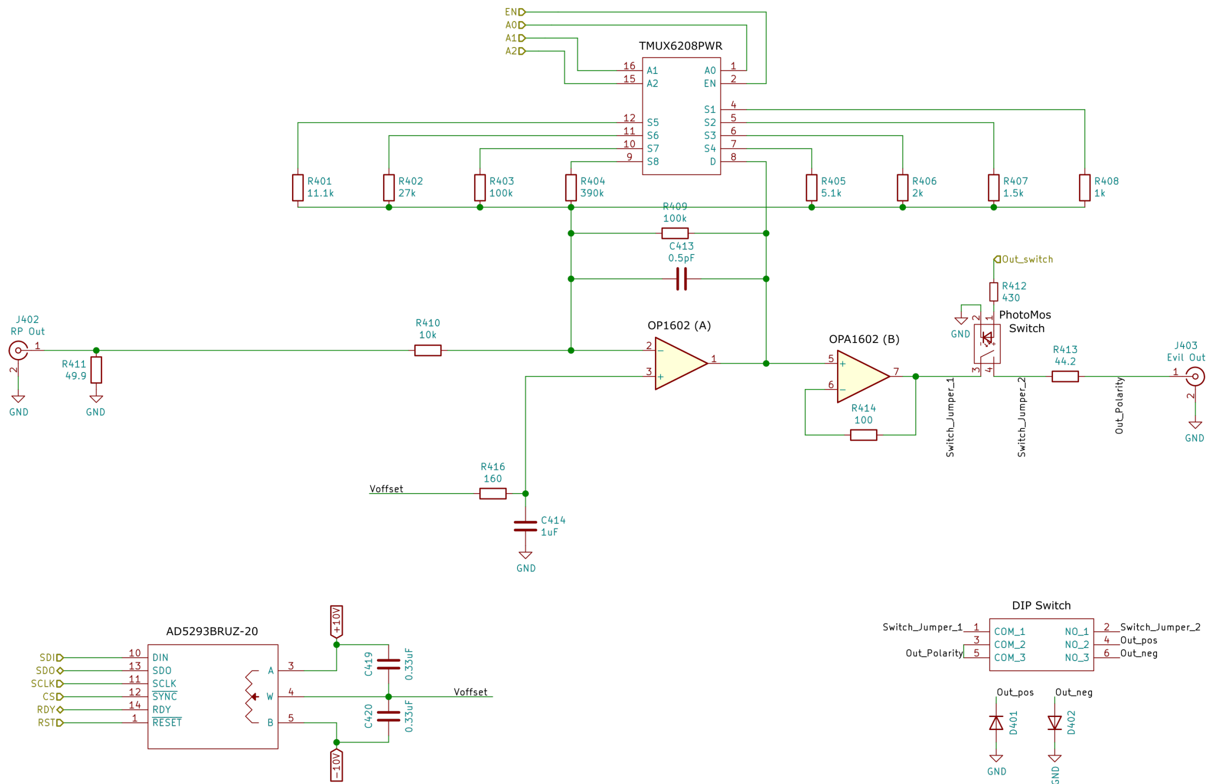



Figure 6: Schematics of the output section, adapted from the KiCad schematics.

### Shift registers and LEDs

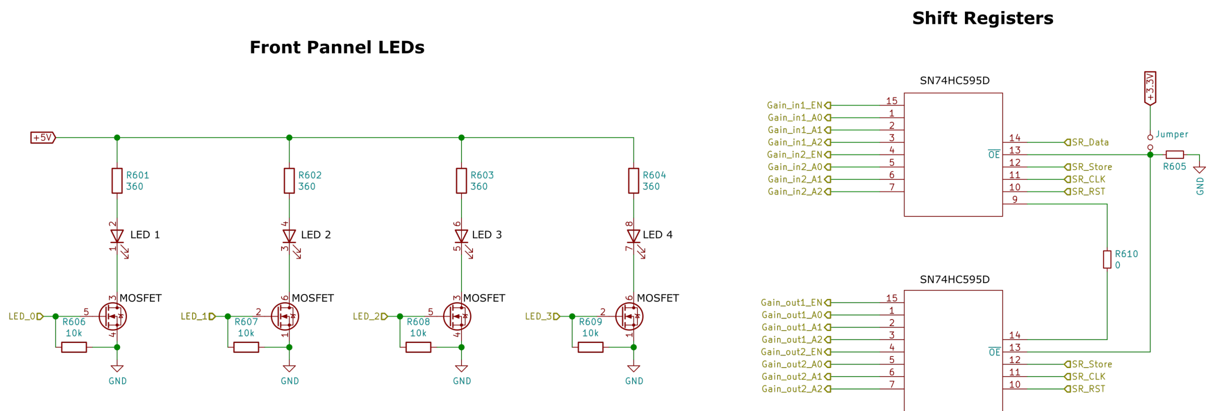



Figure 7: Schematics of the shift registers and LEDs, adapted from the KiCad schematics.

In this section there are 4 LEDs and two 8-bit shift registers which are daisy chained.
The 4 LEDs are status LEDs that can be turned ON and OFF by controlling the Gate of 4 MOSFETs. As written above the signals LED_0-3, that control the Gate, come from the extension connector of the RedPitaya.
The function of the shift registers is to output more signals than they receive as input. This is useful because we have a limited amount of signals coming from the RedPitaya which would not be enough to control all the ICs on the PCB. In our case we control the four multiplexers in the input/output sections, which require 4 signals each, with only 4 signals from the RedPitaya. See the datasheet for more details [10]. The shift registers can be disabled by connecting their output enable pin ($\overline{OE}$ is active low) to 3.3V with a jumper. This is useful for controlling manually the multiplexers.

### 2.1.2 Layout

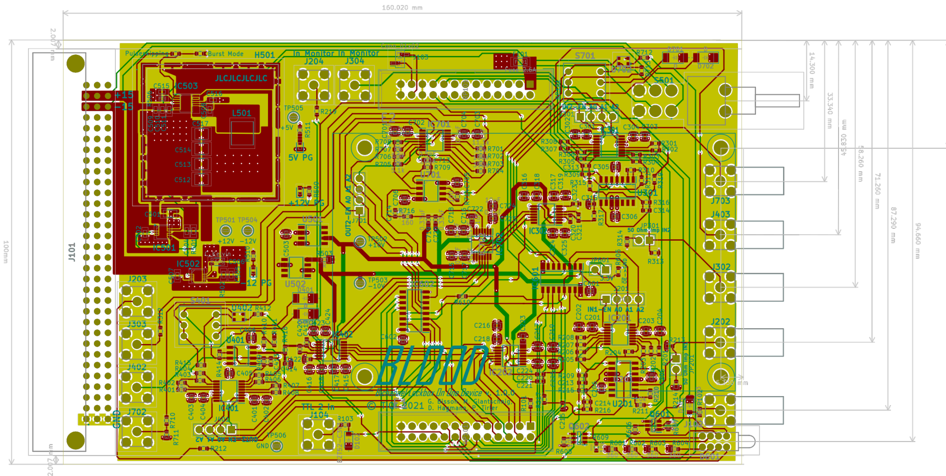


Figure 8: Final layout of the PCB from KiCad.

The carrier board uses the Eurocard format like the old EVIL. The PCB fits in the same casing and has the same connector to the power supplies in the back. The only difference is the front panel which had to be redesigned. As our manufacturer for the PCB we chose JLCPCB and as our stackup we chose the JLC7628. It is a 1.6mm, 4 layers PCB. We organised the layers in the following way:

1. The top layer contains signal traces and the pads for soldering all the components.
2. The second layer is an uninterrupted ground plane.
3. The third layer is divided into all the different power planes that are needed by the components: +5V, +/-12V, 3.3V. These are shaped such that the respective voltages can be reached through vias by the components on the top layer.
4. The bottom layer contains signal traces.

A few other remarks about the layout:

- In order to obtain optimal performance for the power supply section layout we followed the suggestions in the datasheet of the components.
- To improve signal integrity of important signals, e.g. analog input and output signals, we tried to avoid crossing of power planes with different voltages.
- We connected all the decoupling capacitors with vias to the respective power plane
- We decided to mount the RedPitaya on the PCB through 4 standoffs with the heat-sink facing up, this way it is possible to have components on the PCB also under the RedPitaya and the heat dissipated by the SoC will not disturb the components.

### 2.1.3 Soldering

We ordered the PCB from JLCPCB with assembly included for all the resistors and capacitors selected from their components database. We also ordered a stencil which did not arrive in the packaging. In any case it was not possible to use the stencil for soldering the rest of the components because of the capacitors and resistors which arrived already mounted. So we soldered all the components by hand with the soldering iron, a good flux, solder, a heat gun and a lot of patience. We also used a stereo microscope to check if the solder joints looked good and there were no shorts. Figure 9 shows how the PCB looks fully assembled.

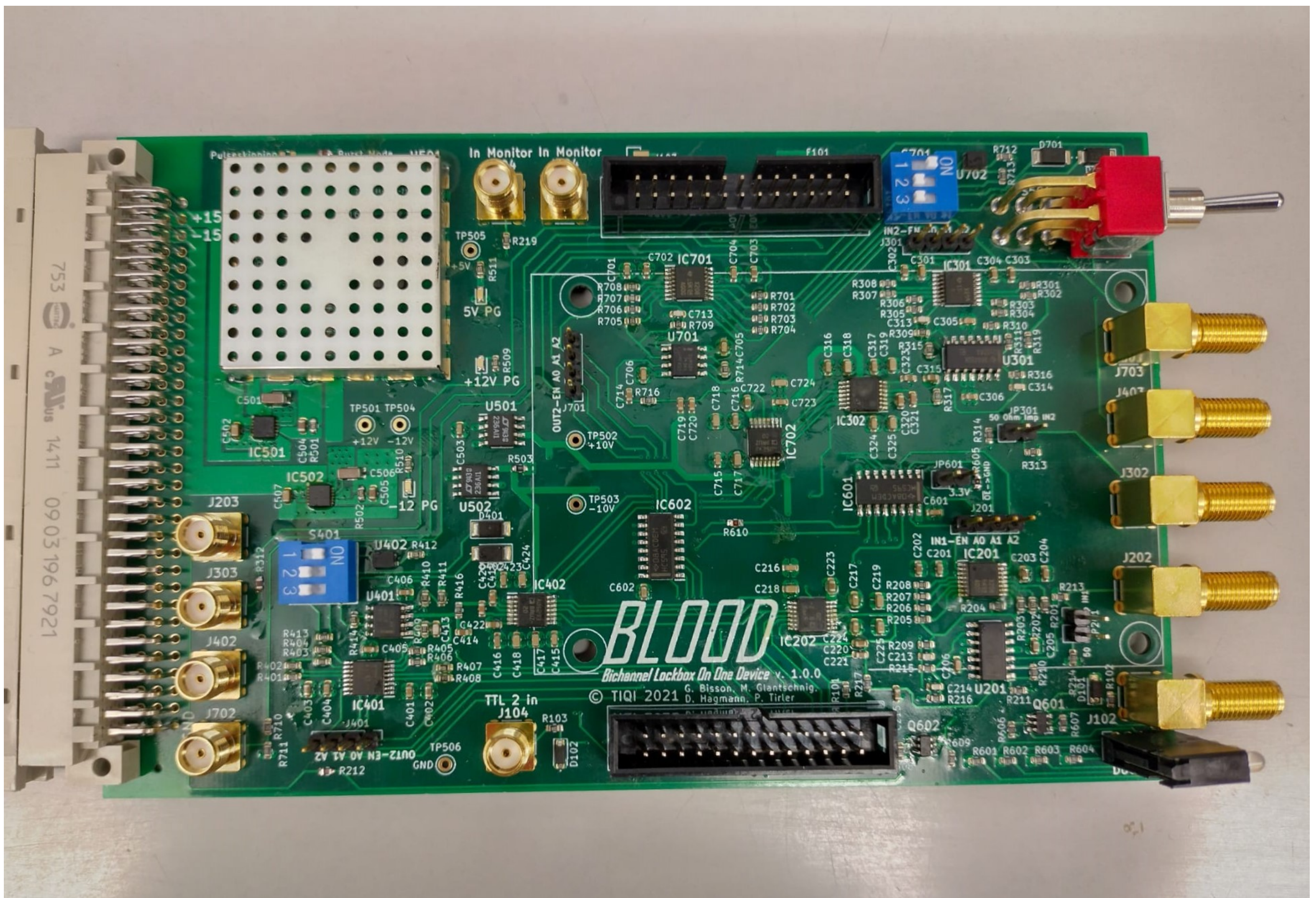



Figure 9: Picture of the PCB after everything is soldered.

### 2.2 Software/Firmware

One of the reasons we decided to build our lockbox around the Red Pitaya STEMlab 125-14 was the integrated CPU on the Xilinx Zynq 7010 SoC. This allows us to run a server application needed to remotely control the lockbox from a lab PC directly on the lockbox itself.

Part of this project was to assess different approaches on how to combine existing software to control the new hardware of the BLOOD. The choice was to use "DEVIL" [13] as the base and combine it with the open-source software package "PyRPL" [19], which was designed by L. Neuhaus and S. Deléglise at the Laboratoire Kastler Brossel in Paris, France, starting in 2014 and was published under the GNU General Public License in 2017.

In the following, we describe the initial testing done on different projects available for the STEM-Lab (2.2.1), describe our final solution (2.2.2) and then go into detail what modifications had to be done to the bitstream (2.2.3), PyRPL (2.2.4), server & client (2.2.5) and the operating system (2.2.6).

#### 2.2.1 Testing of PyRPL

For the Red-Pitaya STEM-Lab there are multiple open-source software projects and part of the selection process was to differentiate between these different solutions and pick the one most suitable for our solution. The different projects evaluated in a first step were:

- Official RedPitaya OS [22]
- PyRPL [19]

The RedPitaya OS provides an FPGA bitstream and a web-application embedded in the Ubuntu based OS which runs on the ARM cores of the Zynq 7010 SoC. Many features are already implemented in

the bitstream like ADC and DAC interface, oscilloscope, Arbitrary Signal Generator (ASG), Network-Analyzer and PID functionality. These can be controlled via the web-application which is accessible from any network capable device connected to the same LAN as the STEM-Lab.

The PyRPL project [19] is another open-source project for the STEM-Lab 14 developed specifically for quantum optics experiments which was originally forked from the original Red Pitaya OS in 2014, but has developed in a different way since then. Like the Red-Pitaya OS it provides an FPGA bitstream which already implements many functional blocks like ADC and DAC interface, Scope, ASG and a custom DSP block. It has no web interface but instead a light-weight server application written in C running on the ARM core. The device can be controlled via the PyRPL client which is written in python but also provides an API such that the device can be controlled by a python script. The custom DSP block provides PID modules, IQ mixers, complex filter functions and advanced trigger options. The strength about the design of the custom DSP block is, that the way the functional blocks are connected can be reconfigured from the client without having to generate a new bitstream.

We did some quick initial test with both software packages to see if we can easily connect and for example generate a signal with the Arbitrary Signal Generator (ASG), feed back the output to the input and look at the sampled waveform from the scope module. With both solutions it was very easy to setup this measurement and both also worked as expected. As the PyRPL project had more interesting features and was actually designed for quantum optics experiments, we decided to preferentially go with this project. In a next step we did some more extensive tests to evaluate the PID module in PyRPL as it is the module used to perform locking and therefore provides the main function of the entire lockbox:

- Measure the transfer function of the PID module
- Measure the achievable data throughput with scope module

**Transfer function**

The transfer function measurements were done using the network-analyzer built into PyRPL. The results showed, that the PID module works as desired, but we also concluded that there is some care to be taken in terms of how this measurement is set up. The measurement should be performed in a closed loop configuration to compensate the numerical drift of the integrator in the PID module due to the high sampling rate of 125 MHz. Otherwise the output of the PID module saturates at the maximum or minimum depending on the first operating point.

**Data Rate**

We measured the achievable data throughput over Ethernet when using the PyRPL client to see if real-time streaming of scope data is possible. We achieved a data rate of 20 Mbit/s which we deemed high enough for real-time plotting of the error signal in the GUI, but not enough for taking real-time traces of 14bit samples at the 125 MHz FPGA clock frequency. As a comparison, the streaming rate achieved with the DEVIL client was 4.8 Mbit/s/channel [3]

The conclusions we found from these tests are:

- PID module works as expected. Some care has to be taken with the integrator.
- The data-rate from scope to client is high enough for real-time plotting of error signal
- The PyRPL PI-control module does not have a working derivative term which is fine at the high sampling rates of 125 Msps. Also the EVIL did not have one either.
- Using PyRPLs python API to control the STEM-Lab is very easy to use and quite powerful. It would be nice to eventually provide such an API to control the BLOOD.
- Measuring the transfer function we found some numerical glitches which were solved later by fixing the timing constraint violation in the PyRPL bitstream (see sec. 2.2.3)

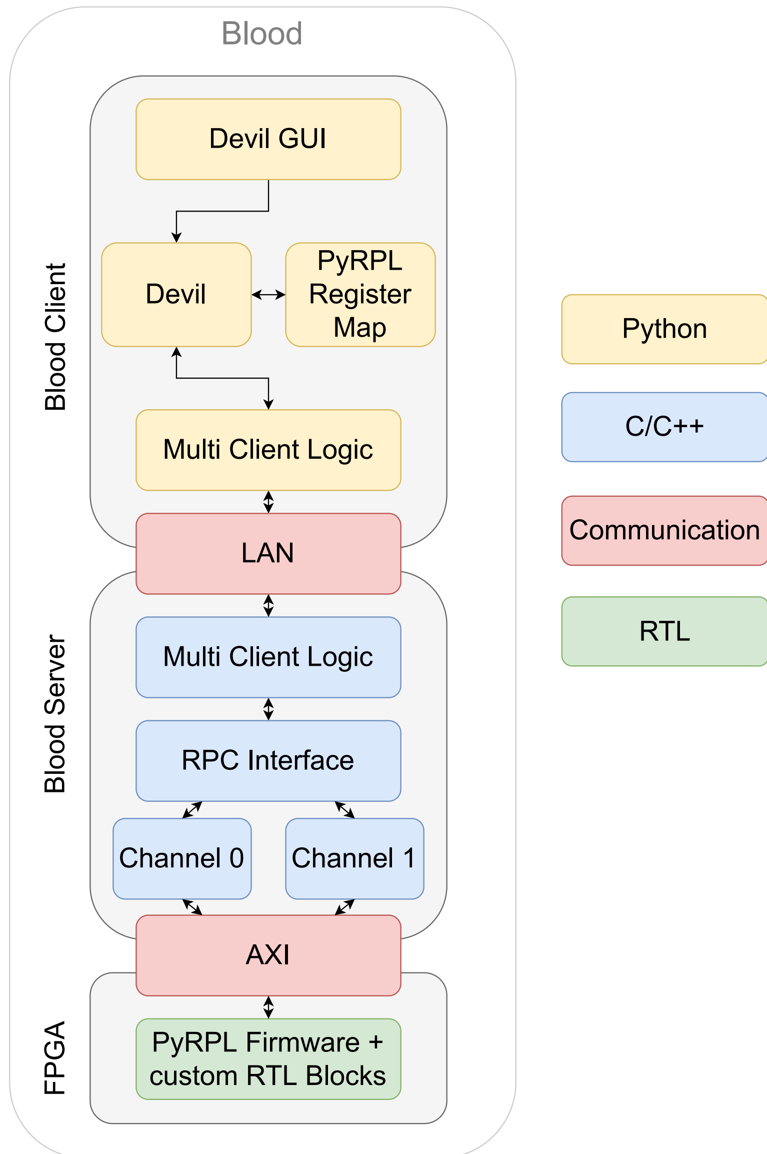



Figure 10: **Software/Firmware Stack** The soft/firmware stack of our final solution. The colors highlight in which language the components are written

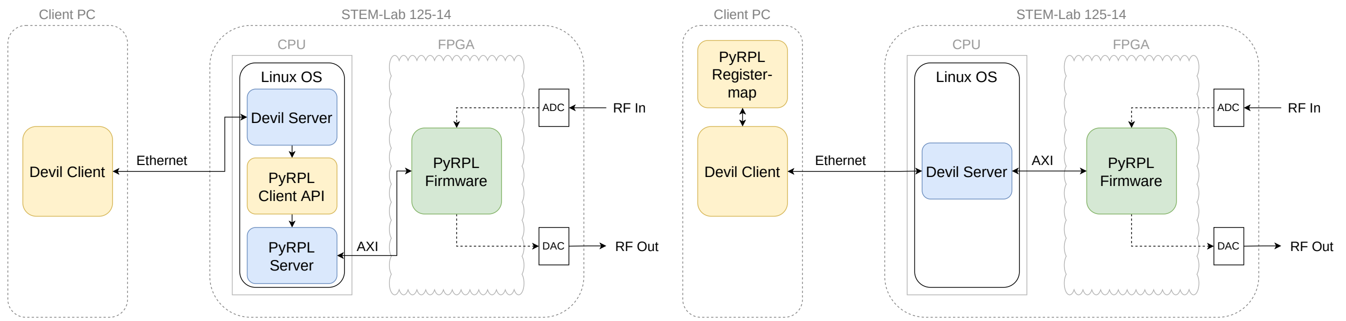



Figure 11: These are the two possible solutions we were considering to interface the bitstream. Blue shows software written in C/C++, yellow shows software written in python. On the left is the approach using a C/python interface, while the solution on the right is the one that we chose ultimately.

### 2.2.2 Software-Firmware Stack

The tests in section 2.2.1 assured us that we want to use the PyRPL bitstream as the base for the BLOOD firmware. In a next step we needed to find a solution how the BLOOD can be controlled from a lab PC. Our main design goal was:

- Keep the user experience similar to the EVIL

The reasoning behind this is that the EVIL has worked quite reliably and is being used quite extensively throughout the TIQI group. By keeping the UI the same we believe that we can maximize the chances of people actually using the BLOOD at some point. From this decision it follows, that we:

- Keep the DEVIL GUI
- Keep multi-client access

Natively PyRPL does not support concurrent multi-client access and we deemed it too difficult for us to integrate such a feature directly into PyRPL. Therefore, we chose the existing DEVIL client and server as our base onto which we add the necessary changes to being able to control and monitor the BLOOD.

The next design decision we had to tackle was how we can interface the DEVIL client and server with the PyRPL bitstream. This was the step where we struggled quite a bit. In the end there were two main approaches that we could follow (see fig 11)

1. DEVIL server has a PyRPL object in its memory and calls the PyRPL API to communicate changes to the FPGA
2. DEVIL client holds a PyRPL object in its memory from where it gets the addresses of registers. It then sends value+address to server. Server then reads and writes to FPGA and makes sure that the values of the registers stay synchronized between the connected clients.

The first approach requires that we are able to call python code from C++ code, which is possible with libraries like `pybind11` or the `python/C API` but ultimately makes the entire system more complex and slower. In addition to that we were not able to run pyhton on our own buildroot OS. Therefore we decided to implement the 2nd approach, where we don't have to run the PyRPL client on the RedPitaya but instead include the PyRPL client into the DEVIL client.

The final software/firmware stack can be seen in Fig. 10.

### 2.2.3 Adaptation of the PyRPL Firmware

To appropriately control the digital ICs on the carrier we added a custom Verilog module to the PyRPL firmware. In this section, we explain how this was interfaced with the rest of the PyRPL RTL code. While making these changes, we realized that the RTL design of PyRPL, as of the latest version from November 2021,[12] did not meet the timing constraints. Towards the end of this section, we will explain which architectural changes we performed to solve this problem.

The top level RTL file of PyRPL connects several modules to the same "system bus", which is essentially a simplified version of an AXI light bus, omitting some of the control signals and making it easier to write a slave. Within the module `red_pitaya_ps`, the system bus signals are converted to AXI and connected to the Zynq7 processing system block. The address range of the system bus is divided into 8 regions of about 1 MB each. Three of these regions were unused in the original PyRPL version. We connected our own module `blood_settings` to one of those regions.

**Control of ICs on BLOOD PCB**

The digital potentiometer (AD5293) for offset control has an SPI interface (CPOL = 0, CPHA = 1) with a word length of 16 bits. We decided to use `spi_master.v` from the TIQI HDL Library [25], because it is well tested and supports several different SPI modes and word lengths. The SPI clock frequency is 1 MHz. During the course of testing, we found that `spi_master.v` needed some slight modifications, see Appendix D.

The multiplexers (TMUX6208) are controlled by four digital inputs (EN, A0, A1, A2), see Sec. 2.1.1, which can be set by writing to two daisy chained shift registers. We wrote our own module `shifter.sv` which is derived from `spi_master.v` and also supports variable clk frequency and word length. Here, the word length was also given by 16 bits (four multiplexers with four bits (EN, A0, A1, A2) each). A Verilog testbench was written and used to compare the timing diagram to the datasheet of the shift registers.

The `blood_settings` module also contains registers to control the status of the front panel LEDs controlled by analog outputs on the extension connector of the RedPitaya as well as the PhotoMOS switches for enabling/disabling the analog signal outputs (see Sec. 2.1.1). In Appendix D, we show a detailed list of all registers with their values and addresses, as well as some details on how the module works.

**Timing Constraints & Utilization**

Running synthesis & implementation in Vivado with the original PyRPL RTL code leads to a severe timing violation. The hold time requirement is violated for about one third of all endpoints and the worst negative slack (WNS) is on the order of 4 ns. Most violations where found for paths belonging to the main clock signal ( `pll_adc_clk` in Vivado, 125 MHz / 8 ns) which also clocks the ADC. Under these circumstances, the correct behavior of the design could not be guaranteed, especially for the IQ modulator blocks, where most timing violations were found. In addition, the utilization of some FPGA resources was beyond 95 %, making it particularly difficult and time intensive for the Synthesis tool to properly account for timing constraints.

The measures taken to mitigate these problems were the following:

1. Remove some of the hardware modules that were not needed for our applications. In the original PyRPL project, there were three PID modules and three IQ modules. Removing one each, we could bring down the utilization of LUTs to about 70 %

2. Add pipelining stages at several places in the datapath. For a detailed description of the modifications, please refer to Appendix E. For the PID module, these modifications result in 16 ns of increased latency, which is acceptable when compared to the 200 ns of latency of the STEMlab with the original PyRPL bitstream.

With these changes the violation of timing constraints could be solved.

### 2.2.4 Modified `pyrpl_tiqi`

In order to control the BLOOD with a version of PyRPL we created a fork of the original GitHub repository on the TIQI GitLab [20] called `pyrpl`. In this version of PyRPL, we added an additional hardware module which wraps the control of the custom module we added to the PyRPL bitstream to control the shift registers, digi-pots, LEDs and output switches. This module is called `BS`, short for ***Blood Settings***. In this repository there is also a folder called *scripts* that contains some example scripts which show how the pyrpl_tiqi client can be used to control the Blood. There are some things to consider when using this client:

- To properly initialize the digipots, the first command that has to be sent over SPI is '0x1802'. This enables updates of the wiper position and only has to be done once after power up.
- The `BS` module has no GUI element, so its settings can only be changed in a python script or the python console (A page in the PyRPL GUI could be added in the future)

With the `pyrpl_tiqi` client the full functionality that PyRPL provides is available, including the python scripting API.

#### 2.2.5 Adaptation of the DEVIL server & client

The situation the DEVIL client and server have to face in this new setup is fundamentally different to the one of the DEVIL. The server is now running on a CPU having high speed interfaces to the FPGA, which completely removes the USB connection and the interface the server used to communicate with the FPGA on the EVIL. Also, the firmware on the FPGA, now provided by PyRPL, is structured differently from the old EVIL firmware, which means client and server have to address and match this change.

**Software functionality**
The main purpose of the BLOOD lockbox is to perform real-time PID control, especially when thinking about a PDH-locking scenario. The primary goal of its software is therefore to allow the user to set control parameters conveniently from a lab PC and to stream the error signal to this lab pc, such that the user has detailed information about the state of the lock. Particular in the case of a PDH lock, the user also has to be able to generate a sawtooth ramp that allows them to identify the right initial output voltage for a successful attempt to lock onto a certain resonance.

The already existing DEVIL software implements exactly that, and additionally has a very nice multi-client logic, and a simple GUI people working in the lab had already gotten used to (see Figure 13). All of this was intended to remain the same, so the old client and server were used as a basis for the BLOOD software, and modifications were made such that they could be adapted in the new scenario. Figure 12 shows a schematic overview of the client-server-firmware relationship and sketches the structure of the data flow. A brief description of the most important changes can be found below. For more detail, please refer to the code on the TIQI GitLab [20].

**Communication with Firmware:**
The new server now communicates with the firmware by directly writing into the memory of the FPGA, which is mapped into the memory of the operating system. This means it can be completely agnostic of both what values it writes to the FPGA as well as what the addresses of the different blocks are, both is simply passed down by the client.
In order to match the expectations of the firmware, the client therefore incorporates now a big part of the PyRPL client, which is also written in python.

**Streaming:**
The original DEVIL server supported live streaming of the error signal and the PID controller output from the FPGA to the client, which was needed for a good calibration of the PDH locking parameters. As PyRPL does not directly support this the way the old firmware did, streaming now works slightly differently. The server continuously reads out the value of a special register from an address hard-coded in the server, using the PyRPL `sampler` module, that reflects the current value of the FPGA's out- or input. Although these values are still sent to the client as a packet, they are now acquired one by one asynchronously, making the time gaps between two acquired points of the same trace more evenly spaced. A technically even better solution is using the PyRPL `scope` module, but as PyRPL comes with only a two-channel oscilloscope, this would have caused trouble when extending the functionality to a second channel.

**Initialisation:**
As the PyRPL firmware is designed to be more general than the BLOOD, many of the registers need to be initialised to given values as soon as the device is started. While the very essential parts will be handled

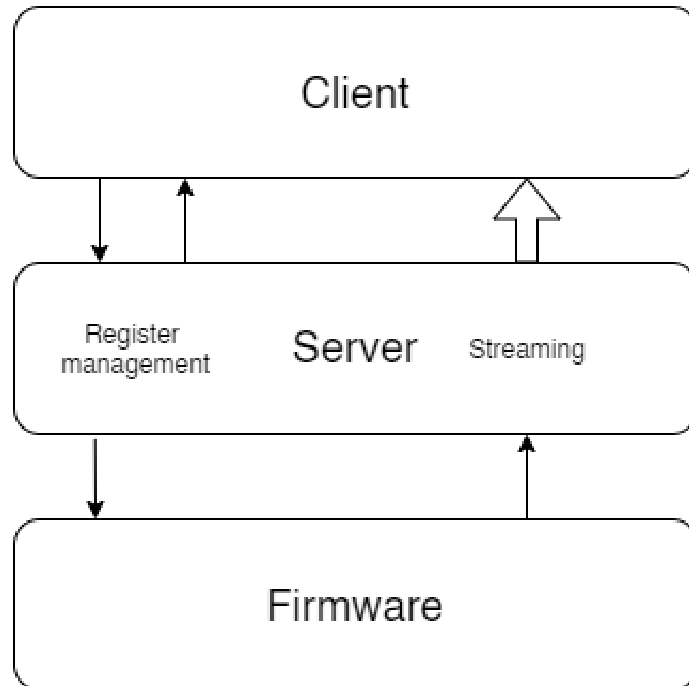


Figure 12: Overview of the workflow of the new BLOOD software. The client passes all settings to the firmware via the server. The server forwards everything to the firmware and saves the settings, such that when the client requests them again it replies from its own cache, never having to read settings from the FPGA.
Meanwhile, the server reads the streaming data as individual equally spaced samples from the FPGA, and groups them together into packages of modifiable size before sending them to the client.

by the server, most registers will be set by the first client that connects to a running server.

**Compatibility:**
It is very likely that in a period of transition older EVILs and some BLOODs will be used side by side simultaneously. To make this transition easier, the new client was intended to still be able to control old EVILs, but there are a few ad-hoc code changes which break this compatibility. These are listed in the Appendix I but should be easy to fix.

**Sliders**
The deadline at the end of the project meant that there was no time left to optimize the usability of the GUI, and so some of the changes we implemented result in buggy sliders. While not a fundamental limit, this has to be fixed before the device can be effectively put to use, and should be addressed first. Other improvements that could not be implemented due to time constraints can be found in appendix H.

#### 2.2.6 Operating System on STEMlab

RedPitaya already provides its STEMlab boards with their own operating system, firmware and software that comes preinstalled on the SD-cards, and maintains documentation which covers many applications.. While being crucial for the straightforward use of the software controlling the STEMlab, this operating system is based on Ubuntu 16.04, which does not support the newest versions of certain C++ libraries, such as *boost* or *azmq*. The Devil server that has been used to control the old EVIL devices however has been kept up-to-date, and requires newer versions of these libraries that are not compatible with the old OS. In order to ensure forward compatibility and avoid a rapid deprecation of the new server software, a custom operating system would be used on the STEMlab, allowing us to use the newest DEVIL-version as a basis for the new BLOOD-server.
This operating system was created using *buildroot*, a tool specifically designed to generate Linux environment for embedded systems. Luckily, a template for the STEMlab was found on Github [9], that creates

Figure 13: Picture of the GUI displaying a PDH signal. The structure has been kept the same, the only thing that changed visually is the left side with the sliders.

an operating system with a modern interface to the FPGA using the Xilinx FPGA manager. It cross-compiles on any machine (e.g. the group high-performance server `tiqiblitz`), while the OS itself targets ARM-architecture. Many software components can be freely chosen like the init system, where we chose systemd.

As part of its philosophy, buildroot does not support the use of C-compilers on the target system, so not only all dependencies, but also the devil and later the Blood-server itself were integrated as packages into the OS. While this was cumbersome at first, it means that once a complete image has been created, it can be flashed onto all devices that use the server, without the need for any further compilation and installation. Concrete advice on how updating and distributing the BLOOD-Server can be done is located in appendix F, and the necessary buildroot files are located on the J-drive under `J:00020210files` .

# 3 Results

## 3.1 Assembly

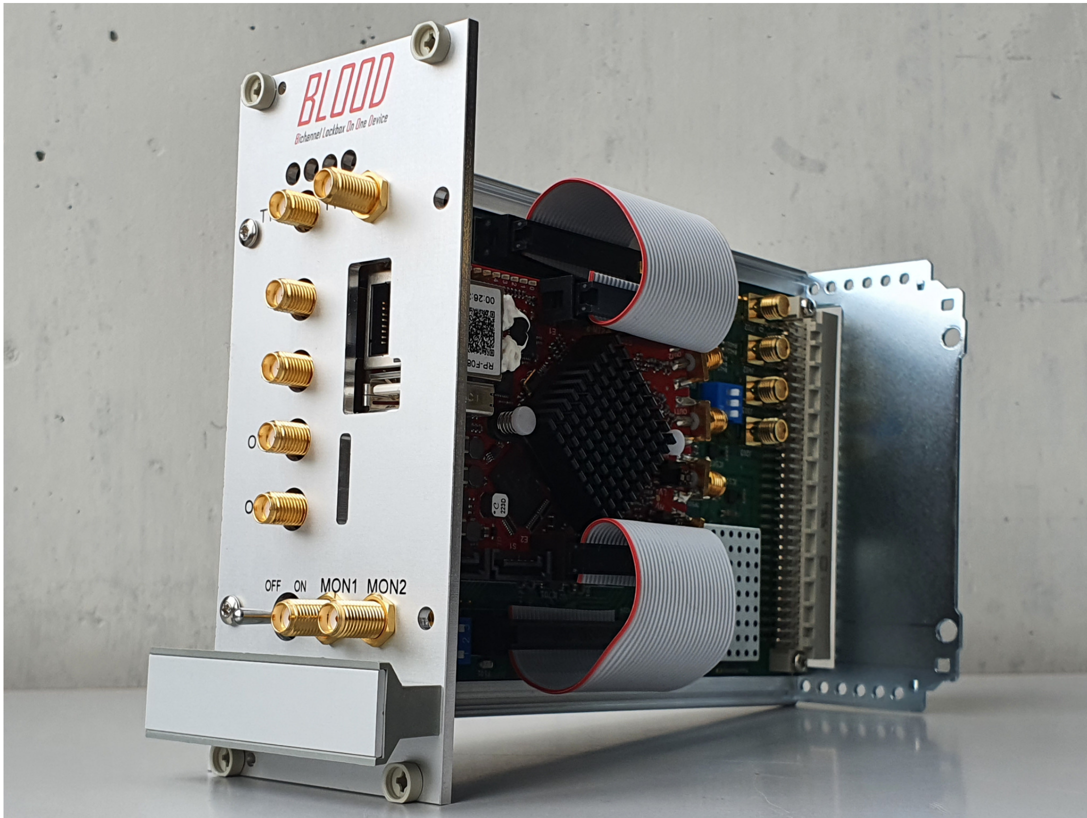



Figure 14: Picture of the final version of the BLOOD almost fully assembled. The internal SMA cables are missing here.

In order to put all the components together we fixed the RedPitaya on the PCB through M2 16mm long standoffs, washers and screws that we purchased in the D-PHYS shop. We connected the RedPitaya to the PCB with two custom made ribbon cables and we used short semi-rigid SMA cables to connect the analog signal inputs and output of the carrier and the RedPitaya. We then assembled our hardware in the updated version of the casing which was used for the EVIL and that fits in the 19” racks used in the laboratory (the casing is sold by "nVent SCHROFF"). Because our PCB had different positions for the inputs and outputs than the EVIL we designed our own front panel which was then manufactured by the company "Beta LAYOUT" (it is possible to find the files of the front panel on the QuanTech folder on the group drive under `J:\Projects\QuanTec\HS2021` ). We connected the upper rows of SMA connectors on the front panel to the PCB with semi-rigid SMA cables. In fig. 14 you can see how the BLOOD looks like once put all together.

## 3.2 PCB Tests

In order to test the functionality of the carrier PCB we used several different pieces of measurement equipment: A FieldFox RF and Microwave analyzer, a 5 GS/s digital oscilloscope and an Analog Discovery 2 (AS2) from Digilent. The latter has two output channels that can be used to generate signals and two input channels. The `WaveForms` software provides a user-friendly graphical interface. This device also has 16 digital I/O that we used to set the gain of the various input/output sections by connecting them to the address pins of the multiplexers and to change the offset of the input/output sections by communicating through SPI with the digital potentiometers. Afterwards, we connected the RedPitaya to the PCB and we checked that the communication between the two worked as expected.
For every input and output section we performed the following tests:

- Gain test (with the AS2): A sinusoidal function was sent as the input of an input or output section and the gain of that section was changed by sending the address to the multiplexer through the header pins on the PCB (in this case the shift registers must be disabled). We checked that the output signal changed accordingly to the gain that was selected.
- Offset test (with the AS2): we verified that the digital potentiometers were able to change the offset correctly by sending commands through the SPI interface. To do this we set the input signal to 0V DC and we set the gain to 1/10 (lowest setting). This way $V_{out} = (\frac{1}{10} + 1)V_{offset}$ (see eq. 2.1).
- Bandwidth test (with the AS2): We measured the bandwidth of each input/output section as the frequency at which the amplitude of the signal decreases by 3dB from its maximum using the AS2.
- Noise test (with the FieldFox): We measured the noise spectrum of each input/output section.

### 3.2.1 Input & Output Sections

Here we present some of the results of the tests. The result of the measurements are very similar for the various input and output sections so we show only the one for input section 1.

Gain and Offset tests: for every section it was possible to change the gain and the offset correctly.

Bandwidth test: the measured bandwidth "B" was the following:

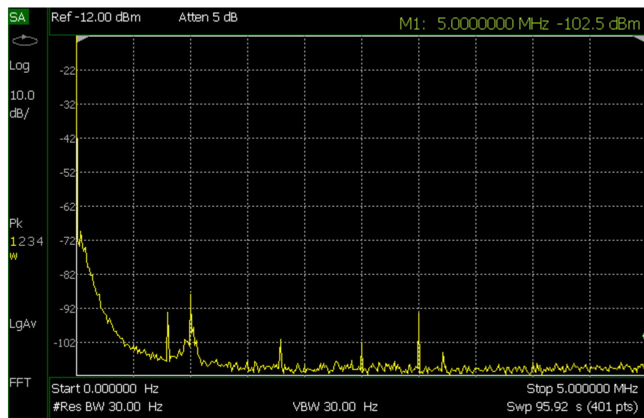
- For a gain G = 0.1 and a probing signal of 3V: B > 5 MHz. In this case the measurement is limited by the bandwidth the Analog Discovery 2.
- For a gain G = 1 and a probing signal of 1V: B = 1.93 ± 0.1 MHz. See fig. 15.
- For a gain G = 7.96 and a probing signal of 0.2V: B = 137 ± 10 kHz.

The uncertainties in the bandwidth are given by how well we could read off the numerical value in the GUI of the AS2.

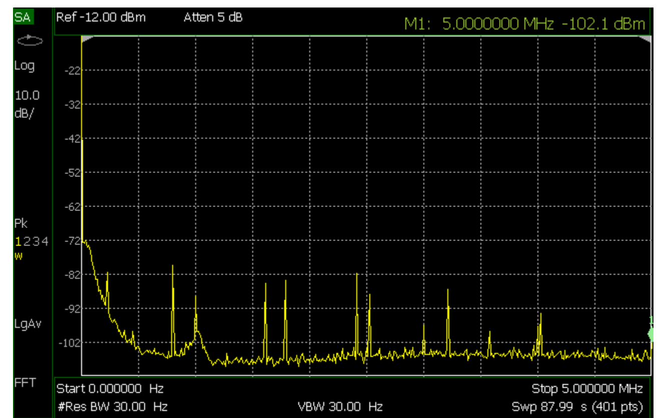
Figure 15: Measurement of the bandwidth of the input section 1 with $G = 1$, measured for a signal of amplitude of 1V, taken with the Analog Discovery 2. The bandwidth was measured as the frequency at which the amplitude of the signal decreases of $\Delta = 3$ dB from its maximum. Here it is equal to $1.93 \pm 0.1$ MHz.

We tried to remove the capacitor "C213" on the feedback path of the OPAMP "OPA1604 (B)" (refer to fig. 5) to see if we could increase the bandwidth this way, however the improvement was negligible. We think this could be due to the parasitic capacitance on the PCB that cannot be removed.

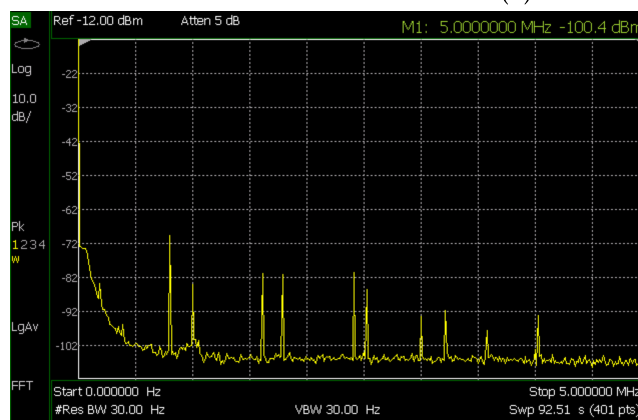
Noise test: for this measurement we used the N9912A FieldFox RF Analyzer which has a higher resolution and lower noise floor than the AD2. In fig. 16 you can see the spectrum. The most prominent peak can be seen at 800 kHz and can be identified with the noise of the 5V switching regulator. The spurs at 1.6 and 2.4 MHz are likely to be higher harmonics thereof. All of these spurs are more prominent for higher amplifications which is expected from noise that couples into an amplifier circuit. At G = 0.1 the most prominent peaks are at 1 and 3 MHz and they do not scale with the gain of the circuit. This suggests that these noise sources are inherent to the FieldFox. There are also some spurs whose origin could not be determined such as the one at 1.75 or 2.5 MHz. We conclude that the worst case spurious free dynamic range (SFDR) is 72 dB with respect to a 0 dBm or 0.6 V rms signal.




(a) Noise measurement for G=0.1.




(b) Noise measurement for G=1.




(c) Noise measurement for G=10.

Figure 16: Noise measurement taken with the N9912A FieldFox RF Analyzer for input section 1 for three different gain settings. The units of the measurement are dBm and the resolution bandwidth is 30Hz.

### 3.2.2 Communication between RedPitaya and ICs on the PCB

After making sure that it was possible to control all the ICs on the PCB through the Analog Discovery 2, we repeated the same tests and measurements with the RedPitaya controlling the PCB. We verified that also in this case it was possible to change gain and offset for all the sections, that we could address the four controllable LEDs on the PCB and that it was possible to enable and disable the outputs by acting on the PhotoMos switches. Everything worked as expected and the results of the measurements were the same as the ones shown above.

## 3.3 Open-loop transfer function

As a next step, we measured the delay of a signal propagating through the whole system, i.e. through our custom PCB *and* the Red Pitaya. We did this by injecting a step signal using the Analog Discovery Kit

and measuring the response with a fast oscilloscope. We defined the delay as the time between the points where the step signal and the response have risen to 50 % of its peak value (see 2).

| Configuration | Latency [ns] |
|---|---|
| Red Pitaya | $218 \pm 5$ |
| Input/Output Section | $74 \pm 5$ |
| Total | $402 \pm 5$ |

Table 2: Results of the latency measurement. We injected a step signal and measured the response with an oscilloscope. We defined the latency to be the time between the points where the signals have risen to 50 % of their peak amplitude. The uncertainty is limited by how precisely we could read off the time difference from the oscilloscope display.

Finally, we measured the open-loop transfer function of the PID module (see Fig. 17). In order to prevent integrator saturation, the accumulator was reset periodically during the measurement. The measurement points are overlayed with the theoretical transfer function, which consists of the ideal PI transfer function $G_{\text{ideal}}(s) = k_P + k_I/s$ and the delay of the BLOOD $G_{\text{delay}}(s) = e^{-sT}$, where $T$ is the delay of the RedPitaya alone (i.e. ADC/DAC and digital delay) from Tab. 2.

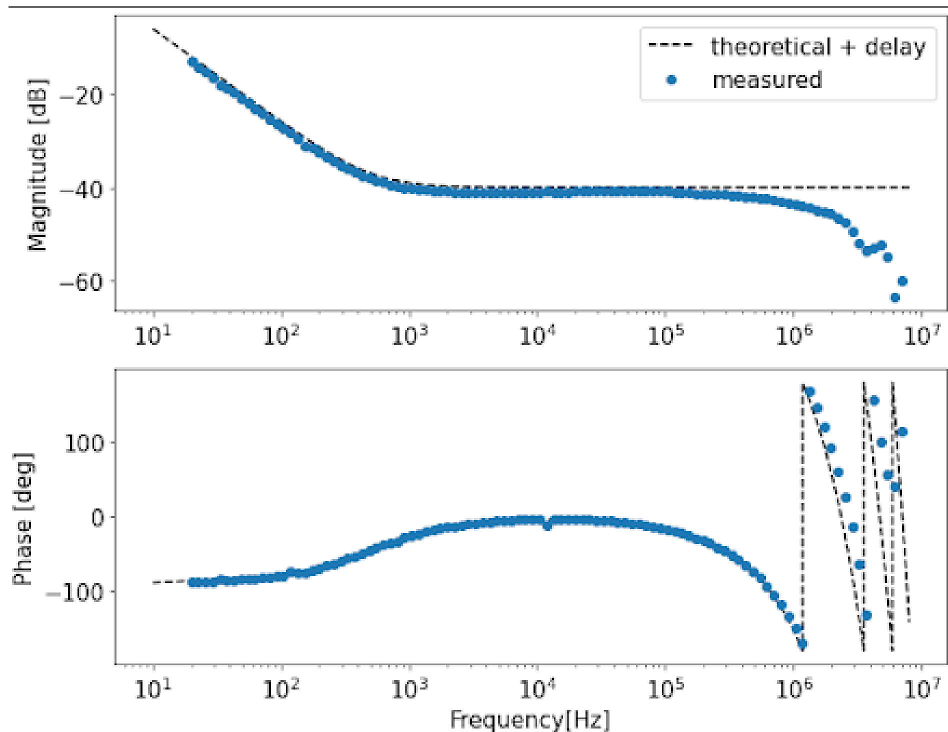



Figure 17: Open loop transfer function of the BLOOD. The PID module was set to $k_P = 0.01$ and $k_I = 5$ (D is not usable at the moment). For the theoretical curve, the delay of 200 ns through the Red Pitaya has been taken into account. While measuring this curve with the Analog Discovery Kit, the accumulator had to be reset periodically to prevent saturation.

### 3.4 PDH lock of laser to a cavity

Although the BLOOD can be used for multiple purposes in the field of feedback control, it is specifically designed to work in the context of frequency stabilisation using a PDH lock, and so performing such a lock was one of the major project goals. This was ultimately performed in the lab of the TIQI group, working on the setup of the molecules experiment, using an already set up PDH lock, that so far had been using the old EVILs. Here, a cavity with a non-linear crystal is used to frequency double a red 626 nm laser into a ultraviolet 313 nm beam. In this case, instead of locking the frequency of a laser to a cavity, the

opposite was done, and the cavity resonance frequency was locked to the laser. This was implemented using a piezo driver, that was controlled via the output voltage of the BLOOD. Thus, the piezo moved according to the BLOOD output voltage, adjusting the cavity resonance frequency. The PDH-principle applies just the same, as the error signal just depends on detuning and direction of the laser and resonance frequency. A schematic overview can be seen in Figure 18.

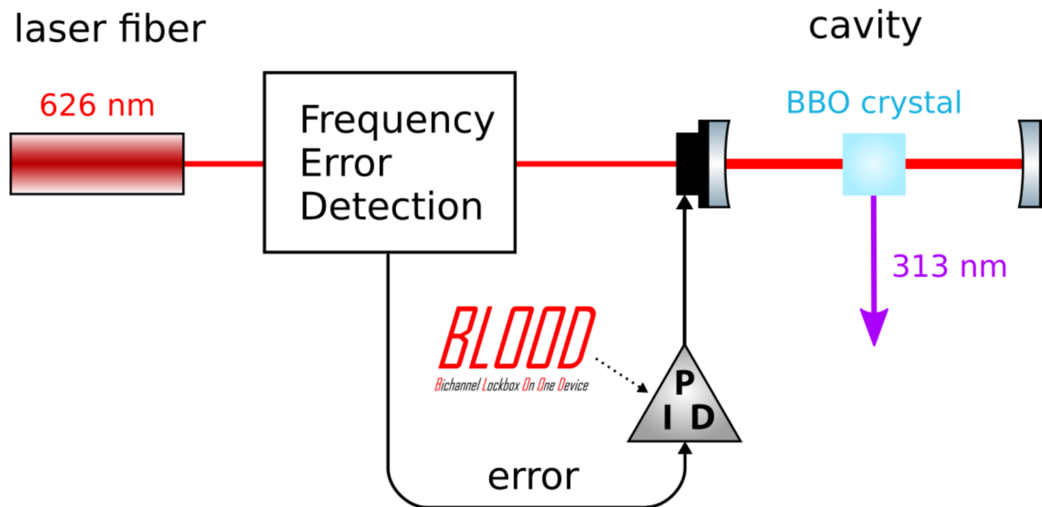



Figure 18: Setup of the PDH lock performed using the BLOOD.

Using the BLOOD as a controller implementing PI-control (the D-gain was actually never used), it was possible to perform the lock and stabilize the cavity frequency to the laser. Evidence of this can be seen in the comparison in Figure 19. However, this only worked on our first attempt. When some time later we wanted to lock the system, we were unable to do so, and due to this being at the very end of our project time frame, we were not able to repeat the locking and collect quantitative data to analyse the lock quality.

We conjecture, that this lack of repeatability is caused by a still incomplete connection of the GUI settings to the FPGA. Locking needs a decent starting point, that is usually achieved by outputting a ramp and then observing how the streamed PDH signal changes under changes to the output ramp. It also needs PID parameters that are set to reasonable values to allow locking. At this point, the GUI however had buggy sliders, that made it very difficult to make sure all parameters in the FPGA corresponded to what was shown in the GUI. Additionally, some of the GUI settings, such as the "reset PID" button did not work at all, meaning every time we wanted to reset the integrator we had to re-load the bitstream into the FPGA. Also, being convinced we would be able to lock again at a later point in time, we did not note down the exact parameters of the first locking, and might just have been unable to get the parameters just right the second time.

A straightforward approach to testing whether the failure to lock a second time was indeed caused by the remaining software issues of the BLOOD would be to try to fix them, and then attempt to lock again.

Figure 19: Screenshots of a video showing the locking process. While on the left the cavity is not on resonance with the laser, on the right the cavity freq. has been locked to the laser. The laser is now resonant in the cavity, which can be seen from the red radiation being scattered by the air, and the violet spot on the paper indicating that UV light is being emitted.

## 3.5 Outlook

### 3.5.1 Open issues

It has been already mentioned, that due to the time constraints of the project, especially on the software side, there have been some issues that could not be resolved and some features that could not be implemented. In a follow-up project these can be improved upon, such that the BLOOD can fully operate the way it was intended to. To this end, a list of changes to the current code we deem necessary can be found in Appendix H. Still, we want to briefly mention the two most crucial aspects that limit the usability of the BLOOD at the moment:

While the BLOOD server was developed so far that it could be used for experiments and was successfully tested, the changes made to the GUI are incomplete to the point where it works and can be used, but can also be a frustrating experience, as many sliders are glitching, and some buttons do not have any effects.

The current software only supports one of the two channels that are physically present on the BLOOD. If the second channel has to be used, (and the cost advantage over the older EVILs is leveraged), the software must be modified to also address the second channel just like the first one. As mentioned, for all further proposed changes and improvements to the software, please refer to Appendix H

### 3.5.2 Conclusion

We managed to design and build a controller for laser stabilisation specifically dedicated to the needs of the TIQI research group, that has been demonstrated to work to the degree where it can perform a PDH lock. It is still incomplete, but with some improvements the BLOOD can live up to the tasks posed by the lab routine in research with trapped ions, and will take its place as a functional and affordable successor of the EVIL.

## References


[1] Daniel Y. Abramovitch. "Built-in stepped-sine measurements for digital control systems". In: *2015 IEEE Conference on Control Applications (CCA)*. 2015, pp. 145–150. DOI: 10.1109/CCA.2015.7320624.

[2] *boost.asio cross-platform C++ library for network and low-level I/O programming*. URL: https://www.boost.org/doc/libs/1_76_0/doc/html/boost_asio.html (visited on 02/20/2022).

[3] Ludwig de Clercq. "Transport Quantum Logic Gates for Trapped Ions". PhD thesis. Zurich: ETH Zurich, 2015. URL: https://ethz.ch/content/dam/ethz/special-interest/phys/quantum-electronics/tiqi-dam/documents/phd_theses/Thesis_L%20de%20Clercq_final.pdf (visited on 02/24/2021).

[4] ANALOG DEVICES. *AD5293 Digital Potentiometer Data sheet.*

[5] ANALOG DEVICES. *LT1236 Datasheet*. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/lt1236.pdf.

[6] ANALOG DEVICES. *LT3045 Datasheet*. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/lt3045.pdf.

[7] ANALOG DEVICES. *LT8610 Datasheet*. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/lt8610a-8610ab.pdf.

[8] R. W. P. Drever et al. "Laser phase and frequency stabilization using an optical resonator". In: *Applied Physics B: Lasers and Optics* 31.2 (June 1983), pp. 97–105. DOI: 10.1007/BF00702605.

[9] Gwenhael Goavec-Merou. *redpitaya*. https://github.com/trabucayre/redpitaya. 2021.

[10] TEXAS INSTRUMENTS. *SN74HC595 Shift Register Datasheet*. URL: https://www.ti.com/lit/gpn/sn74hc595.

[11] TEXAS INSTRUMENTS. *TMUX620x Multiplexer Datasheet*. URL: https://www.ti.com/lit/ds/symlink/tmux6208.pdf.

[12] Samuel Deleglise Leo Neuhaus. *PyRPL github log file.*

[13] David Nadlinger. "Experimental Control and Benchmarking for Single-Qubit Trapped-Ion Transport Gates". Semester thesis. Zurich: ETH Zurich, Mar. 2016. URL: https://ethz.ch/content/dam/ethz/special-interest/phys/quantum-electronics/tiqi-dam/documents/semester_theses/semesterthesis-David_Nadlinger_2016 (visited on 02/06/2022).

[14] Vlad Negnevitsky. "Feedback-stabilised quantum states in a mixed-species ion system". PhD thesis. Zurich: ETH Zurich, 2018. URL: https://ethz.ch/content/dam/ethz/special-interest/phys/quantum-electronics/tiqi-dam/documents/phd_theses/Thesis_VNegnevitsky.pdf (visited on 02/24/2021).

[15] NEXPERIA. *74HC595 Datasheet*. URL: https://assets.nexperia.com/documents/data-sheet/74HC_HCT595.pdf.

[16] Panasonic. *AQY221N3MY PhotoMOS Datasheet*. URL: https://www3.panasonic.biz/ac/e_download/control/relay/photomos/catalog/semi_eng_rf1a_aqy22_m_cr10.pdf.

[17] T. Preuschoff, M. Schlosser, and G. Birkl. "Digital laser frequency and intensity stabilization based on the STEMlab platform (originally Red Pitaya)". In: *Review of Scientific Instruments* 91.8 (Aug. 1, 2020). Publisher: American Institute of Physics, p. 083001. ISSN: 0034-6748. DOI: 10.1063/5.0009524. URL: https://aip.scitation.org/doi/10.1063/5.0009524 (visited on 03/15/2021).

[18] *PyQt binding of Qt framework*. URL: https://wiki.python.org/moin/PyQt (visited on 02/20/2022).

[19] *PyRPL*. URL: https://pyrpl.readthedocs.io/en/latest/ (visited on 02/18/2022).

[20] *PyRPL fork on TIQI GitLab*. URL: https://gitlab.phys.ethz.ch/tiqi-projects/pyrpl (visited on 02/18/2022).

[21] *Qt framework*. URL: https://www.qt.io/ (visited on 02/20/2022).

[22] *Red Pitaya OS*. URL: https://github.com/RedPitaya/RedPitaya (visited on 02/22/2022).

[23] *RedPitaya STEMlab 125-14*. URL: https://redpitaya.com/stemlab-125-14/ (visited on 02/20/2022).

[24] *RedPitaya-IntStab github project*. URL: https://github.com/TU-Darmstadt-APQ/RedPitaya-IntStab.

[25] *TIQI HDL Library - GitLab*. URL: https://gitlab.phys.ethz.ch/tiqi-projects/fpga/tiqi_hdl_lib (visited on 02/06/2022).

[26] *ZeroMQ, An open-source universal messaging library*. URL: https://zeromq.org/ (visited on 02/20/2022).

# A Specification Table

The following table compares the key specifications of the EVIL (current solution at TIQI) with our newly designed BLOOD device.

Table 3: Comparison between EVIL and BLOOD

| | EVIL | BLOOD |
|---|---|---|
| Processor | – | Dual-Core ARM Cortex-A9 |
| FPGA | Xilinx Spartan 3 XC3S500E | Xilinx Zynq 7010 |
| Connectivity | USB 2.0 | Ethernet |
| RF Inputs | | |
| Channels | 2 | 2 |
| Sample rate | 96 MSPS | 125 MSPS |
| ADC resolution | 10 Bit | 14 Bit |
| Full scale voltage range | ±10V | ±10V |
| Input impedance | 50 Ω | 1 MΩ or 50 Ω (user selectable with jumper) |
| Adjustable gain | Yes (potentiometer on PCB) | Yes (digitally) |
| Adjustable offset | Yes (potentiometer on PCB) | Yes (digitally) |
| RF Outputs | | |
| Channels | 1 fast, 1 slow | 2 |
| Sample rate | 96 MSPS, 10 MSPS | 125 MSPS |
| DAC resolution | 14 Bit, 12 Bit | 14 Bit |
| Full scale voltage range | ±10V | ±10V |
| Load Impedance | 50 Ω | 50 Ω |
| Adjustable gain | Yes (potentiometer on PCB) | Yes (digitally) |
| Adjustable offset | No | Yes (digitally) |
| Control loop characteristics | | |
| Latency | 250 ns (fast path) | 400 ns |
| Bandwidth | 500 kHz (fast), 80 kHz (slow) from [3] | To be measured* |

*The loop bandwidth is not limited by the latency of the system. Assuming a latency of 400 ns and a desired phase margin of 45°, the maximum achievable loop bandwidth is about 1.5 MHz. The other limiting factor is the bandwidth of the analog input and output sections, which depends heavily on the gain setting used, as explained in Sec. 3.2.1.

# B Cost-Calculation

In this section, we show how the price for one BLOOD unit is composed. In general, we bought components for the full assembly of five boards (Except for the RedPitaya where we only bought 4). For the non-trivial components we bough six, in order to have a spare one if something goes wrong during soldering. The number of PCBs and front panels ordered was also five.

Table 4: Cost-Calculation for the Project

| Position | Price per BLOOD unit |
|---|---|
| Red-Pitaya STEMlab 125-14 | CHF 310.13 |
| Schroff Aluminium Casette for 19' racks | CHF 45.08 |
| Non-trivial components for BLOOD PCB (OpAmps, LDOs, Multiplexers, DigiPots, etc.) | CHF 108.80 |
| Remaining components for BLOOD PCB | CHF 126.85 |
| PCB Manufacturing and SMD Assembly | CHF 23.17 |
| Front panel | CHF 18.48 |
| Total | CHF 632.51 |

Per channel, the total cost for the BLOOD is CHF 316.25. Comparing this to the cost for the EVIL per channel ( $\approx$ CHF 500), we have lowered the costs by about 35 %.

All the invoices/confirmation files for the orders made during this project can be found in the QuanTech folder on the TIQI J: drive.

# C Repositories

Here you find a list of all the repositories and the respective branch we worked on:

### pyrpl (pyrpl_tiqi)

- branch: `tiqi/blood/master`
- URL: https://gitlab.phys.ethz.ch/tiqi-projects/pyrpl

### devil

- branch: `blood_devil , blood_devil_dev`
- URL: gitlab.phys.ethz.ch/tiqi-projects/devil/-/tree/blood_devil_dev

### devil_server

- branch: `Blood—devil—server`
- URL: gitlab.phys.ethz.ch/tiqi-projects/devil_server/-/tree/Blood-devil-server

### BloodPCB

- branch: `main`
- URL: gitlab.phys.ethz.ch/tiqi-projects/quantech/blood-pcb

# D Verilog module for gain & offset control

This section contains details about the implementation of `blood_settings`. The base address for all user accessible registers is 0x4050_0000. Table 5 shows a list of registers including their address offset, bit offset and number of bits.

| Register | Address offset | Bit offset | # of bits |
|---|---|---|---|
| gain_in1 | 0x00 | 0 | 4 |
| gain_in2 | 0x04 | 0 | 4 |
| gain_out1 | 0x08 | 0 | 4 |
| gain_out2 | 0x0C | 0 | 4 |
| offset_in1 | 0x10 | 0 | 16 |
| offset_in2 | 0x14 | 0 | 16 |
| offset_out1 | 0x18 | 0 | 16 |
| offset_out2 | 0x1C | 0 | 16 |
| output_switch[0] | 0x20 | 0 | 1 |
| output_switch[1] | 0x20 | 1 | 1 |
| deactivate_module | 0x24 | 0 | 1 |
| spi_received (read only) | 0x28 | 0 | 16 |
| leds_o[0] | 0x2C | 0 | 1 |
| leds_o[1] | 0x2C | 1 | 1 |
| leds_o[2] | 0x2C | 2 | 1 |
| leds_o[3] | 0x2C | 3 | 1 |

Table 5: List of user accessible registers in `blood_settings`. All register except `spi_received` are read-/write.

**Gain settings:** For the four gain registers the bit assignment is the following: [0] EN, [1] A0, [2] A1, [3] A2, where EN, A0, A1, A2 are the control pins on one TMUX6208 multiplexer. As explained in Sec. 2.1.1, two daisy-chained 8-bit shift registers are used to supply the multiplexers with the gain settings. Within `blood_settings`, the module `shifter` implements the communication with the shift registers. The module is parametrized and the user can choose the transfer length and clock frequency. A transfer is triggered when `start_gain` is asserted, which is the case when any of the four gain registers changes its value. The message is then given by the concatenation of the current values of all gain registers: `{gain_out2, gain_out1, gain_in2, gain_in1}`. Note that also the values of those registers which did not change are transferred again. This is necessary because the two shift registers are connected in series. Figure 20 shows a waveform of the RTL simulation. Important to note here is the pulse on `sr_store` which comes after all bits has been shifted in. This is necessary because the SN74HC595D has two layers of registers, see Figure 21. `sr_store` is connected to latches between the actual shift registers (flip-flops) and the outputs. The single pulse on `sr_store` makes sure that the outputs only change once all of the data has been shifted in.

**Offset:** The offset registers contain the whole SPI command that will be sent to the digital potentiometer. The bit assignment is the following: [9:0] value of RDAC register (proportional to the voltage ouput of the digipot), [13:10] control bits, [15:14] unused (set to zero). The two most important SPI commands shall be explained in the following, for further information please refer to the data sheet of the AD5293. [4] After power up, one needs to enable writing to the RDAC register by sending the command 0x1802. This is currently automatically done by the BLOOD software once it boots up. Then, one can change the RDAC value by putting the bits [13:10] to 0b0001 and [9:0] to the desired value. The RDAC value is represented

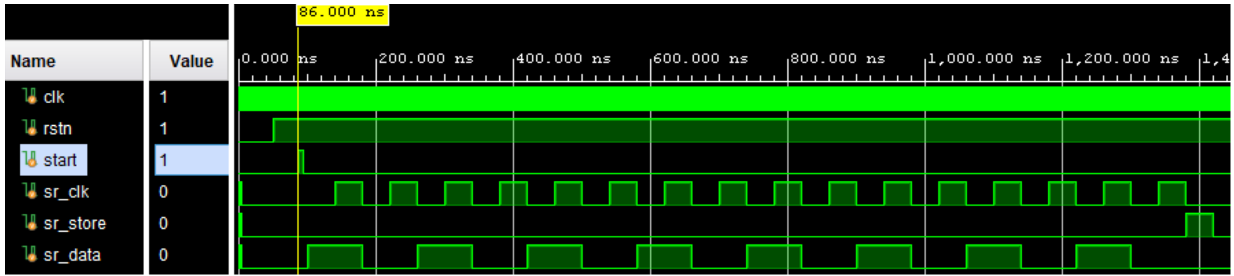



Figure 20: Behavioral simulation of `shifter.sv` in Vivado. The signal `clk` simulates the main clock signal of 125 MHz. The frequency of `sr_clk` is adjustable by the user (in our design 1 MHz). The data sent in this example is `0b1010_1010_1010_1010`.

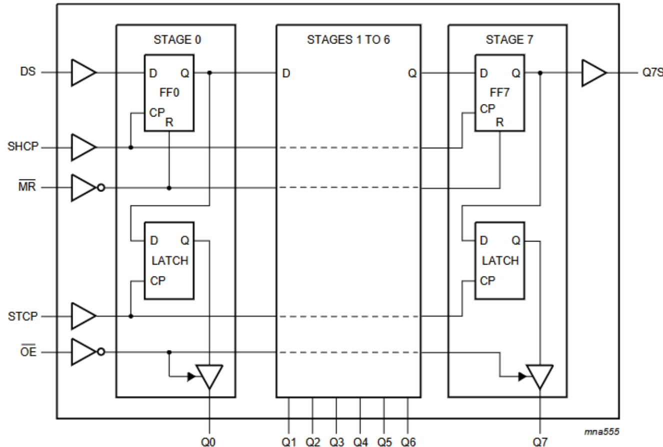



Figure 21: Schematics from the data sheet of the 74HC595 shift register from Nexperia [15], which is identical to the SN74HC595D from TI we used. Here, STCP corresponds to `sr_store` as mentioned in the main text. SHCP is the main clock that controls the transfer (1 MHz in our design).

as an unsigned number, so to set the wiper to the middle position (corresponding to an output voltage of 0 V), one has to use the value 0b1000_0000 for [9:0], which corresponds to half the full range.

For implementing the SPI communication we used `spi_master.v` from the TIQI HDL library. The following modifications had to made to `spi_master.v` to make it work with the AD5293:

1. There is an increased time of `SCLK_BEGIN_END_CYCLES` half cycles between the negative edge of CS and the first sclk pulse. The same extra time was introduced between the last sclk pulse and the positive edge of CS. While not strictly required by the SPI protocol, we found that the AD5293 would not communicate via `spi_master.v` unless these changes were implemented. We became aware of this because the Analog Discovery Kit's SPI module, for which CS is high long before the first and last sclk edge, worked right away with the AD5293.

2. sclk does not go high anymore after the last negative transition.

Fig. 22 shows these changes by comparing the simulated wave forms of `spi_master.v` for one SPI transfer.

**Other functionality** The PhotoMOS switch for enabling/disabling the analog ouput channels (see Sec. 3) of the BLOOD are directly controlled by the bit written to `output_switch[0]` or `output_switch[1]`. The default value after power up is 0 (i.e. output disabled). Same goes for controlling the LEDs on the front panel through `led_o[i]`, i = 0,1,2,3.

The 1-bit signal `deactivate_module` is required because `blood_settings` needs to share access to the signals on the expansion connector of the STEMlab with another module from the original PyRPL project. By writing 1 to `deactivate_module`, `blood_settings` is disconnected from the expansion connector and the original functionality of PyRPL is regained.

Figure 22: Comparison of the changes made to `spi_master.v`. (a) The behavior of the original version of the module. (b) Adapted version needed to make it work with the AD5293. In both cases, the data sent is `0b1010_1010_1010_1010`.

# E Pipelining of PyRPL RTL code

In the following section, we want to describe the changes made to the PyRPL RTL code in an attempt to mitigate the timing errors. We will describe for each modified Verilog module what were the exact changes and what parts of the design are affected by them.

1. In `red_pitaya_product_sat`, one pipelining stage was added (between the sum and the product). Instances of this module were present in several places in the design: in the `red_pitaya_iq_modulator_block` and in the `red_pitaya_iir_block`. The first one was straightforward to deal with because `red_pitaya_product_sat` was never used within the feedback path of a filter. The `red_pitaya_iir_block`, however, did contain loops and required experience with digital filter design to solve it. Martin Stadler helped us out here. The overall latency increase here was **two** clock cycles.

2. There was another pipelining stage added to `red_pitaya_iq_modulator_block` for the signals `secondproduct1` and `secondproduct2`. In total, the latency of `red_pitaya_iq_modulator_block` was increased by **two**.

3. The `red_pitaya_lpf_block` module is used in almost all of PyRPL. It can dynamically be configured to one out of three modes: "low-pass", "high-pass" or "filter disabled" (i.e. signal passes right through). In the latter case, there was no register in the signal path. Often, several instances of these filter blocks were used in series. Together with the fact that all DSP modules are interconnected via a multiplexer (i.e. zero latency interconnection), this lead to long combinatorial paths, which did not fulfill timing. This was resolved by introducing one pipeline register on the "filter disabled" path. As a consequence, the PID module, amongst others, now has an increased latency of **two** clock cycles when the filter stages are turned off.

On the J: drive in the QuanTech project folder, you can find a PDF which graphically shows where the pipelining stages were added.

# F Handling the BLOOD Server

**Start the Server** Before the BLOOD-server can run succesfully, its bitstream must be loaded into the FPGA. At the moment, this still has to be done manually. The first time a device is used, or whenever the bitstream has been updated, it has to be transferred manually to the BLOOD device in question, e.g. via scp or sftp. It then has to be loaded into the FPGA using the Xilinx FPGA manager, by executing four commands in the terminal:

1. `echo 0x10 > /sys/class/fpga_manager/fpga0/flags`
   (`0x10` is the compressed bitstream flag)

2. `mkdir -p /lib/firmware`
3. `cp ./foo_bitsteam.bin /lib/firmware`
4. `echo foo_bitstream.bin > /sys/class/fpga_manager/fpga0/firmware`

This has to be performed after every powercycle, and it is therefore recommended to use simple scripts to simplify or automate this. Note here, that the first command sets the flag for a compressed bitstream, as this is what PyRPL will give us.

Once the bitstream is loaded, the devil server can be started by simply calling `devild` in the terminal.

**Update the Server:** The building of the OS image will already have preinstalled the BLOOD server on the image. This has the advantage that it does not have to be built or installed for every single device, but already preinstalled it when the OS is flashed onto the SD-card. However, it makes implementing changes more difficult. If a new version of the server is to be installed onto a device, a new buildroot OS must be created, which includes the BLOOD server as a package. The easiest way to do so, is to use a buildroot copy already set up to create Linux environments for the BLOOD, e.g. the one in the project's folder on the J-drive. In such a case, just copy the new source code as a `tar.gz` file into ./dl/devil-server. Note that it must have the same name as the file already existing there, which in this case would be `devil_server-boost_1_71_0.tar.gz`. Going back to the main folder the new OS can now be built using the `make` command. As only the Blood-server part is actually rebuilt, this does not take so much time.

Once this is done, the image in `./output/images/sdcard.img` can be flashed onto a sdcard and used. In most cases it would also suffice to just copy the shared object file from `./output/target/usr/lib/libdevil.so` to `/usr/lib` of the OS running on the STEMlab, which in many cases is more practical.

If a new Buildroot folder is set up to create operating systems for the BLOOD, not based on an already existing one, some care should be taken so that is has similar features and settings as the ones used so far. There exists a template for an image for the RedPitaya on gitlab [9], to which modifications have to be made such that it can support the devil server. The most important (and only essential) ones are adding

- `systemd`
- `boost`
- `azmq`

libraries and making sure

- `dhcpd`
- `ssh`

are activated to be able to connect to the device. Note that when using `systemd` in buildroot, one has to first change the standard C library to `glibc`. To speed up this process, a new gitlab repository for the buildroot OS could be made, which already includes both the Blood server and the right RedPitaya template as a submodule. This would speed up the deployment process of new BLOOD server versions.

# G Documentation for BLOOD-Devil client

## G.1 Integration of PyRPL client into Devil client

Both PyRPL and Devil client have python register objects which represent a register on the PyRPL FPGA. But the objects themselves are quite different. While PyRPL uses a large inheritance tree to differentiate many different types of registers (see PyRPL documentation) ultimately based on the `BaseRegister` class, the Devil only has one register type, the `Register` class. We chose to write a `BloodRegister` class that wraps a PyRPL register and makes it available to the Devil client. As there are many different PyRPL registers, there are also several BloodRegisters. The Devil Client has to be able to deal with the new registers and also adapt to the changes in the devil server, therefore a `BloodChannel` class, that inherits from the

`Channel` was written, that does most of the work. The original Devil client was written in such a way, that it can deal with general *resources* which get announced by the devil server over the custom protocol called *fliquer* (using port 8474). The blood server was changed to announce a `tiqi.blood—devil.channel`. One of the main design principles we followed was to keep the blood client compatible with old devil servers announcing `tiqi.devil.channel`. This principle was followed throughout the client code except for some rushed changes towards the end of the project which are listed in the appendix I.

## G.2 The `BloodRegister` class

The BloodRegister is the main PyRPL register wrapper class. At initialisation it expects a PyRPL hardware module `pyrpl_module`, the name of the PyRPL register `reg_name` and a dictionary `reg_cache` where the current value of the register will be stored. The BloodRegister uses the reference to a PyRPL register to get the following information:

- register address
- bitmask
- default value
- maximum value
- minimum value
- conversion functions `register.to_python()`, `register.from_python()`
- value validation function `register.validate_and_normalize()`

In the original `Register` class the `uval` and `sval` (unsigned and signed value) of the register value are differentiated. In the `BloodRegister` these are re purposed to:

| | |
|---|---|
| `register.sval` | register bits converted to the python value |
| `register.uval` | bits of the actual FPGA register |

To understand this consider the following: In the FPGA, float values are represented in fixed-point notation. To work with the actual value these can be converted to a float by using the `register.sval` attribute. We do this conversion because PyRPL also does it. To avoid doing the same work twice, we reuse the conversion functions already provided by the PyRPL register class.

Compared to the `Register` class the `BloodRegister` class has a `setup_widget()` method which gets called by the `BloodChannel` to setup the GUI widget and connect it to this register. Register classes inheriting from `BloodRegister` can redefine the `setup_widget()` method if they need to set up a different type of GUI widget.

The reason that the `BloodRegister` requires a `reg_cache` (which is a dictionary provided by the `BloodChannel`) is because PyRPL defines different python registers which point to the same physical 32 bit register in the FPGA. Two examples where this is used:

- control bits of a module: For efficient use of address space several different binary control settings are taken together in one register, where each bit represents a different true or false setting. In PyRPL each of these bits is then represented as its own independent `BoolRegister`
- In the ASG module, the upper 16 bits of the register with offset `0x4` represent the output waveform offset and the lower 16 bits represent the amplitude of the output waveform which are also separated as different registers in PyRPL.

This mechanism complicates how a `BloodChannel` propagates a changed value received from the notification socket to the actual `BloodRegister` object (see fig. 23) The `register.bitmask` defines which bits of `reg_cache[register.address]` belong to the `register`.

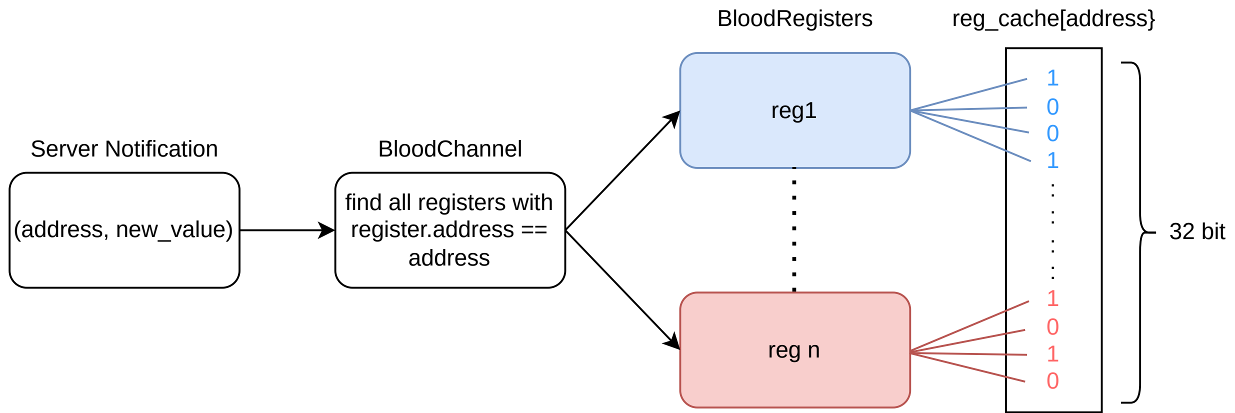



Figure 23: The Blood Client receives a notification from the Blood server. It then propagates the `new_value` to all the `BloodRegisters` that have the address specified in the notification. Each `BloodRegister` then updates its bits in the `reg_cache[address]`

## G.3 The `BloodChannel` class

The `BloodChannel` class inherits from the `Channel` class and abstracts one physical channel of the BLOOD. Currently it is written in a way that only one channel of the BLOOD is functional, but it should be relatively easy to adapt the `Channel` class such that there are 2 `BloodChannel` objects for each `BloodServer` that connects. The main differences of the `BloodChannel` class compared to the `Channel` class are:

- It can deal with `BloodRegisters` and interface with PyRPL to get information like register address, conversion functions, etc.
- It implements a mechanism to initialize the FPGA registers after a power-up of the Blood. This mechanism also checks if another client is in the process of initializing the FPGA. For details see G.4.
- It provides a `modifyRegisters` RPC command to speed up the writing of the ASG waveform
- Adds `_init_regs` array which are the registers which should be initialized by the client after a power-up of the BLOOD.
  **Note:** Some of the registers are also initialized by the Blood server.
- Deals with the fact that multiple `BloodRegister` can reference to different bits of the same physical FPGA register. See also the section G.2 on the `BloodRegister` class.
- Has additional references to its parent server and the `redpitaya` instance (Module from PyRPL representing the physical state of the RedPitaya device. See the PyRPL docs for details [19])
- The client differentiates `BloodRegisters` by using an ID. The server on the other hand only uses the register address to distinguish different registers.

The BloodChannel class spins up several ZeroMQ sockets which connect to the respective sockets on the server side. You can see the details of this in section G.10.

## G.4 Initialization of Server after power-up

To properly initialize the Blood after power-up, the first client that connects to it will initialize the registers of the FPGA which need to be initialized. To prevent a second client to start the initialization process, while the first one is still initializing, the special register `SERVER_BOOTUP_REG` with address 0x00 is used as a semaphore (see Table 6). The second client then waits until the first client has finished the initialization

## G.5 The `BloodServer` class

This is a new class which is essentially a copy of the `Channel` class with less features. Its purpose is to create and manage a `pyrpl.redpitaya` instance which is then passed to the `BloodChannel` instances of this server. It also provides an RPC interface which can connect to the respective socket in the `Server::managementInterface_`. Right now the server does not implement any of the RPC calls which are already present in the client version. This could be used at a later point to, for example, control registers which are common to both hardware channels of the Blood. In this project we only implemented the control of one hardware channel due to lack of time towards the end of the project.

## G.6 Channel Detection process

The Channel detection process uses the custom *fliquer* protocol which was developed for the original Devil. It is a simple network protocol where different nodes of the network have local resources that they can announce to the other nodes in the network. Once the server node has broadcasted its resources over fliquer to the client node, the client then creates `BloodChannels` for each `tiqi.blood—devil.channel` resource that has been returned by fliquer (can also be multiple if there are multiple servers in the same network). Then the initialisation process of the channel starts, which can be seen in fig. 24 .

## G.7 Changes in the GUI

The only part of the GUI that we modified for the blood is the *register area* with the sliders and buttons on the left in fig. 13. To still be compatible with the old Evils, we created a new `bloodregisterarea.ui` file which holds the sliders and buttons to control the Blood. To address the fact, that many of the registers in PyRPL are actually represented as a double, we chose different QtWidgets and also created new ones. For values where the type is double, we use a QDoubleSpinBox, for a discrete set of selectable values like the analog output and input gain we use a QComboBox. We wanted to keep the sliders, but Qt doesn't provide sliders which return a double value, so we created a wrapper class of the QSlider class which we called the QDoubleSlider and returns double values. For the analog input and output section we also added a modified QDoubleSpinBox which converts the offset settings of the digipots to the actual offset voltage. The new widgets can be found in the file `customwidgets.py`

## G.8 transferfunctionplot.py

We also implemented a function which lets the user plot the open loop transfer function of the PID module with the current $k_P, k_I, k_D$ values. This uses the built-in function in PyRPL to compute the transfer-function.

## G.9 Server only registers used between client and server

These are registers which are not mapped to the FPGA but are used to synchronize the state between server and client.

**Server bootup register: 0x0**

This is the register used to implement the semaphore which is used in the initialization process of the BLOOD after power-up. Each connecting client checks the state of this register before any other registers are read or changed. See also sec. G.4 and table 6

| `SERVER_BOOTUP_REG` value | **State description** |
|---|---|
| 0 | Server is uninitialized |
| 1 | Server setup is in progress |
| 2 | Server setup completed |

Table 6: States of the `SERVER_BOOTUP_REG`

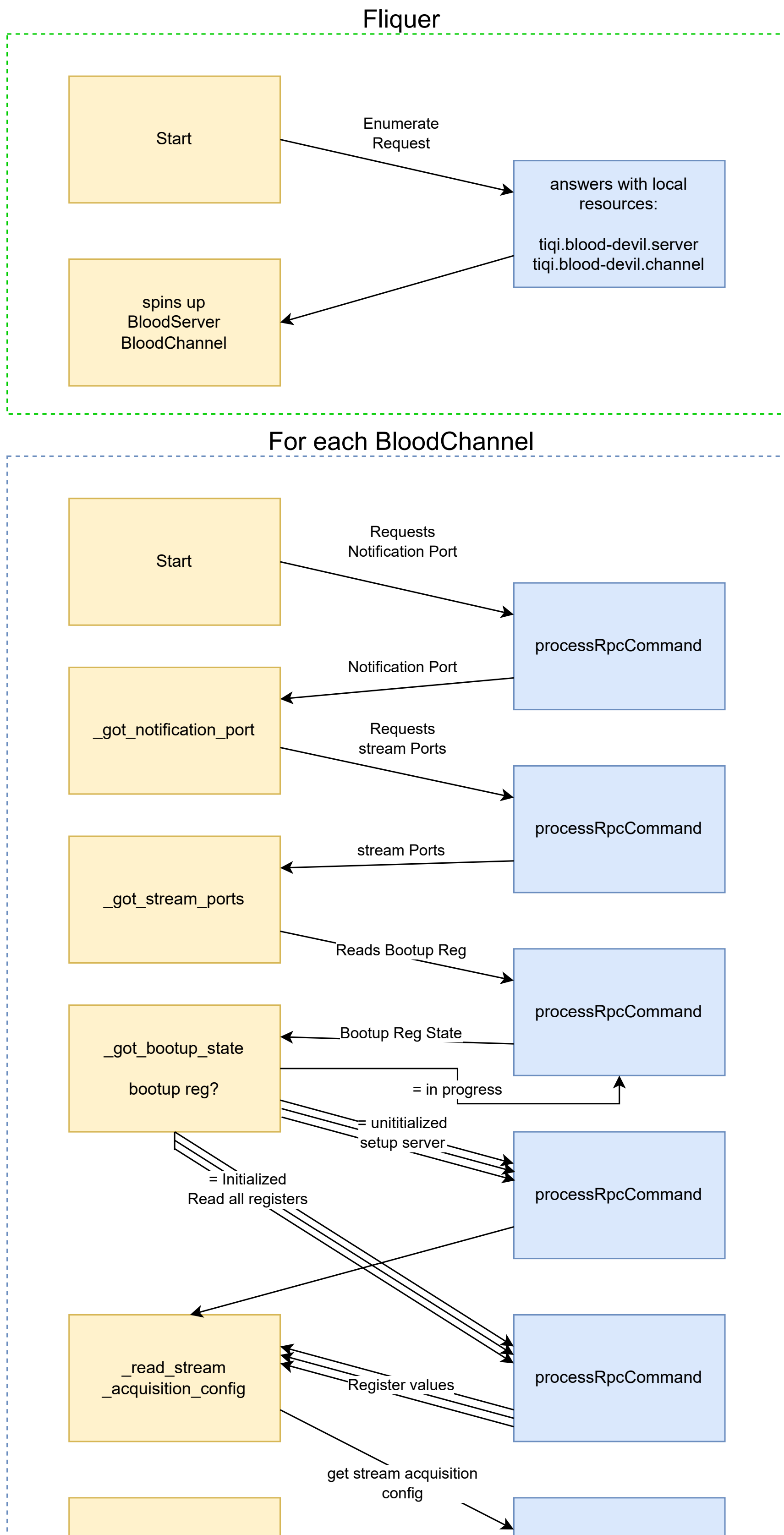



Figure 24: This graph shows in the first part (green frame) how the discovery of resources over fliquer happens. In the second part it shows the RpcCalls from the client that initialize the `BloodChannel` (blue frame)

**Control register: 0x4**

This is the register used to control the state of the channel. For example one of the bits is used to tell the server weather to lock or to scan the cavity.

| SERVER_CONTROL_REG bit | name | setting |
|---|---|---|
| 0 | RAMP_EN | Enables the ramping from ASG |
| 1 | PID_EN | PID output routed to Output |
| 2 | PID_RST | Signals to other clients that PID is being reset |
| 3 | OUT_EN | Enables the output of the channel |

Table 7: States of the SERVER_CONTROL_REG

This implementation is chosen to keep the signaling of PID reset between clients independent of the actual register writes that have to be done to reset the PID module. Currently this is done by setting the ival register of the PID module to 0.

**Status register: 0x8**

This register is used to send status bits back from the server to the client. Currently it is not used, but we left it as a placeholder. Conditions that would be worth to write to this register:

- A bit that signals when the integrator of the PID is saturated at maximum or minimum value.
- A bit that signals when the ADC input is saturated at maximum or minimum value.
- A bit that signals when the DAC output is saturated at maximum or minimum value.

The client could then read these bits and provide visual feedback through the GUI. Exactly such a mechanism is implemented for the Evil.

## G.10 Communication between server `DummyChannel` and client `BloodChannel`

The communication between server and client uses the ZeroMQ library [26] to spin up multiple sockets. These are:

- RPC socket
- notification socket
- streaming sockets

Each channel object spins up 1 RPC socket, 1 notification socket and 3 streaming sockets. See also fig. 25

**RPC socket**

The RPC socket is the socket through which the client sends commands to the server. The server then sends a response message. Available commands are:

| | |
|---|---|
| `ping` | Used to implement a Heartbeat |
| `readRegister` | Request to read a register |
| `modifyRegister` | Request to modify a register |
| `modifyRegisters` | Request to modify an array of registers |
| `notificationPort` | Requests the port of the notification socket |
| `streamPorts` | Requests the ports of available stream sockets |
| `streamAcquisitionConfig` | Requests the current stream acquisition config |
| `setStreamAcquisitionConfig` | Request to modify stream acquisition config |

| server | client |
|---|---|
| ZeroMQ `REP` socket | ZeroMQ `REQ` socket |

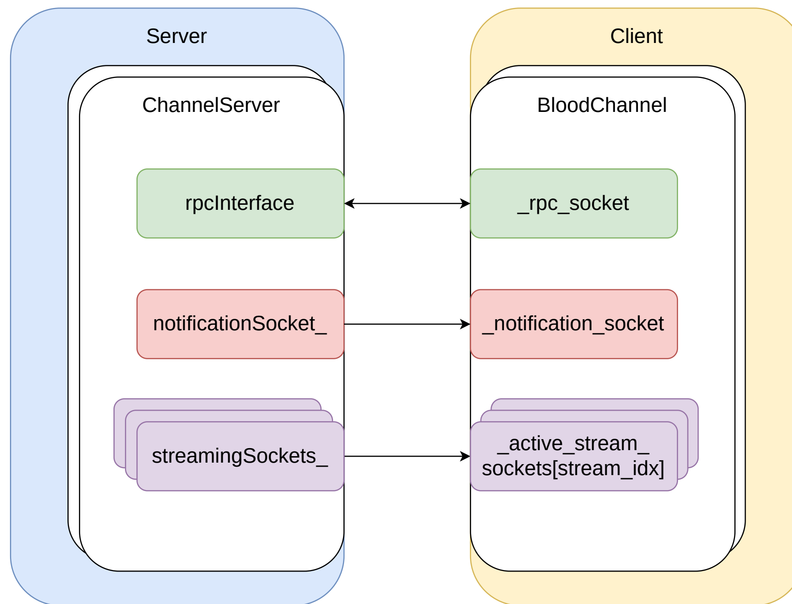



Figure 25: Shows how the different sockets between the client and the server are connected. The RPC interface uses the request and reply scheme `REQ, REP` from ZeroMQ. The notification and streaming sockets use the publish and subscribe scheme `PUB, SUB`

**notification socket**

The server uses this socket to communicate changes in registers and stream acquisition configuration to all the connected clients. It is therefore responsible for keeping all the clients synchronized.

| **server** | **client** |
|---|---|
| ZeroMQ `PUB` socket | ZeroMQ `SUB` socket |

**streaming socket**

For each available stream the server spins up a streaming socket to which the clients can connect to receive real-time data from the ADC output or the PID output. This data is used to produce the streaming plots in the GUI. Available streams:

1. ADC (error signal)
2. PID/ramp output
3. ASG output

| **server** | **client** |
|---|---|
| ZeroMQ `PUB` socket | ZeroMQ `SUB` socket |

# H Detailed list of proposed code changes and missing features

## H.1 General Features

### H.1.1 Entire System

- Use the LEDs on the front pannel of the Blood to display useful system information like if the devil server has successfully booted up or the integrator is saturated or the Blood is out of lock.
- Implement the second hardware channel in the server and client.
- Reactivate the DeviceObserver in the BloodServer such that an EVIL could be connected to the available USB port of the RedPitaya.
- Implement a stepped sine in-loop system characterization which can help the user to optimize the controller performance in terms of stability and noise. This is a rather difficult task and will require changes in both the server and the GUI. The PyRPL bitstream already provides the basic elements which are required to perform this. These are the `scope` and `IQ` module. This measurement can be performed with PyRPL when using the software module `network analyzer`. Some good resources are:
  - Abramovitch [1], Built-in stepped-sine measurements for digital control systems
  - User Guide from HP on how to correctly perform Closed Loop Sensitivity and Responsivity measurements
  - Blog on how to implement the fine details of a PID controller
- Add the capability that the name of the channel can be changed from GUI and doesn't have to be changed by connection to the RedPitaya using an SSH shell and modifying the `devild.json` file.
- Implement automatic relocking. Requires more than a simple PID controller for PDH locks. An implementation using a low-pass filter to detect when a temporary, large disturbance comes in has already been tried with the Evil but didn't work sufficiently well.
- Somehow make a similar API like the PyRPL API available so that easily python scripts can be written to control the Blood.

### H.1.2 Server

- Add ability to store and recover the current state of the Blood using the SD card. This way, at bootup the Blood doesn't have to wait for a Client to connect to initialize itself but can rather use the data from the SD card.
- Implement a better sampling by using the scope instead of the sampler module in the FPGA. Then trigger the scope on the trigger signal from the ASG.

### H.1.3 Client

- The PID module in the PyRPL bitstream has input filter coefficients. These are not yet available to the user in the Blood-Devil GUI. Make these available for further design options of the control loop. These could be added to the transfer function plot window.

## H.2 Fixes for the Client

The development of the blood-devil client happened under time pressure. Some changes should be refactored and there are some features which should be implemented. Here we try to give an extensive list of code improvements:

### H.2.1 `main.py`

- If the server announces the `tiqi.blood—devil.channel` before the `tiqi.blood—devil.server` is announced, the `find_server()` method fails to find the parent server and the client might crash.
- Improve the mapping of channels to the parent server: Right now the parent server is determined based on the host-address. A channel is associated with the server with the same host address. This is probably not safe if network address translation (NAT) happens because then multiple servers could be hidden behind the same IP and thus the channel could be assigned to the wrong server. Two possible solutions are:
    1. Adapt the Fliquer protocol that each fliquer node receives a unique ID, then compare the fliquer ID's to retrieve the parent server
    2. Make it such that each connected blood-server automatically creates two channel instances and change server code such that it doesn't announce channels.
    3. Make sure that both the parent server and the channel resource get announced in the same fliqer message by the server. Then give the received message an id and pass it down to the `_new_resource()` function as a parameter. Store this message id in the created server and client resource. So then in the `_new_resource` function when creating a channel resource, find the server resource that has the matching message ID.

### H.2.2 `bloodserver.py`

- Refactor the `BloodServer` class together with the `Channel` and `BloodChannel` classes depending on whether the given object has an RpcInterface or also a notification and streaming sockets.
- Think about the necessary RPC commands that should be implemented in the `BloodServer` between the server and client RPC sockets.

### H.2.3 `bloodchannel.py`

- Put all the register classes in a separate file `BloodRegisters.py` to make two smaller files out of the huge `bloodchannel.py` file. You could also think about including the `Register` class from the `channel.py` file and then call the new file `Registers.py`
- Remove the empty `BloodBuffer` class.
- Remove all the print statements once the GUI works a little more reliable to make the code more readable
- Look at the `evil2channel.py` code and understand how the `_sweep_timings` should be adjusted to work for the Blood. If this is done correctly, then the extraplot item in `streamingview.py` `_add_extra_items_from_dict()` at line `period = items.get('period')` should work again. This then shows the output ramp plotted on top of the received ADC signal.
- Reconsider the Gain and CornerFrequency sliders. The issue here is that the PID module from PyRPL allows for negative P and I gains, so this separation into Gain and CornerFrequency is no longer unique. Either take into account the polarity of the P and I sliders or remove them.
- Think of which registers really need to be initialized by the client and double check with with the current initialization process in the server. Currently switching between ramping/sweeping, resetting the PID, enabling the output, streaming synchronization is handled by the server. Remove the ones which are not really needed (also the comments)

### H.2.4 `customwidgets.py`

- For the `QDoubleSlider` class add different scaling types as the sliders are not usefull right now. For example the Frequency slider has linear scaling. This is a problem, as we would like to have a lot of selection points for low frequencies and only a few ones for high frequencies. Here an exponential/logarithmic scaling would be perfect. For the P and I gain sliders, there's a problem as well. The max and min value of the sliders is extracted from PyRPL. But also here the linear scaling only gives us a very small amount of points around 0, while it gives us a lot of selection points for $I, P > 1$ which is useless. Here a different scaling based for example on a $P = sinh(x_{slider})$ function would make sense. Giving a lot of selection points around 0 and much fewer towards the boundaries going to $P_{max}, P_{min}$ If the scaling is fixed, the sliders will start to become much more useful.
- Implement some way that the slider does not send too many updates of the value to the client when it is dragged with the mouse. Currently dragging the sliders too fast can lead to a crash of the client.
- Debug why the `valueFromText` function of the `AnalogOffsetSpinBox` never gets called. This should make it possible to change the analog offset voltage by setting a value in the Offset spinbox.
- Check if there should be different conversion functions for the input and output offset voltage. The relevant conversions here are:
  - input: digipot value to input offset voltage at device input as a function of input gain
  - output: digipot value to output offset voltage at device output as a function of output gain.

  Right now the conversion function for analog input offset and analog output offset are the same.

## H.3 Server

### H.3.1 `DummyChannel.cpp`

- Rename all classes and files in a reasonable way, changing the `devil`-based nomenclature to `blood`.
- Change the `const auto clockInterval = 1s / 96e6` to `const auto clockInteral = 1s / 125e6` as the RedPitaya has a FPGA clock frequency of 125 MHz. The old value is still from the Evil.

## H.4 Bitstream

- Connect the TTL lines to the pause signal of the integrator in the PID module. Then add a setting to the `BS.py` module of `pyrpl_tiqi` which allows one to enable and disable the pausing of the integrator via a TTL signal. Also make this setting available in the Devil GUI. This could be used to stabilize systems without a continuous error signal.

## H.5 `pyrpl_tiqi`

- Add a poetry configuration to install `pyrpl_tiqi`
- Add a statement to `pyrpl_tiqi` that automatically writes the first command needed to initialize the digipots.
- Create a GUI window for the `BS.py` module in the PyRPL GUI such that the full PyRPL GUI can be used to control a Blood.
- In transfer function computation of (PID, IIR, IQ) take into account the extra delay introduced to fix the timing constraints of the FPGA design. In particular when the filter value is set to 0 then instead of having 0 delay the filter block has a delay of 1 clock cycle. Also add the exra delay introduced by the Blood analog input and output stages.
- Pyrpl fails to load proper default values for Registers.

# I Modifications that make the BloodClient incompatible with the EVIL

These modifications should be straightforward to implement such that backward compatibility with EVILs is restored.

- Modification of `StreamPacket` class in `channel.py` which changed how the stream packet values are converted. Revert the code changes back to the original Devil and instead create a `BloodStreamPacket` class in `bloodchannel.py`.
- Put back the -1 sign in 'offset' extra plot item in `streamingiew.py`. Instead multiply the 'offset' value by -1 in the `BloodChannel` class.