Documentation Red Pitaya

Introduction and Goal (Alexander Schmid)

The Red Pitaya STEMlab 125-10 is a “Multifunction Lab Instrument”.

More concretely, it is a single-board computer consisting of a dual-core ARM Cortex-A9 MPCore CPU, a Xilinx Zynq 7010 FPGA, a number of digital IO pins and four coaxial RF I/Os (two inputs and two outputs).

The Red Pitaya runs a custom Ubuntu-based operating system with an nginx-based web server. This allows the user to connect to the Red Pitaya over a local network and, via a web-browser, interact with a number of included applications. Among these applications are an oscilloscope, signal generator, logic analyzer and others.

The included logic analyzer supports reading data from the Red Pitaya’s digital I/O-pins and decoding the I2C, SPI and UART protocols. This limited number of supported protocols has proven to be insufficient.

The goal of this project is to support the analyzing of additional protocols with the Red Pitaya, especially the CAN protocol.

Because the included logic analyzer application is closed-source, this will necessitate building a new logic analyzer application from scratch.

Important/helpful links to get start or to deal with problems

Information about rp.h :
https://github.com/RedPitaya/RedPitaya/blob/master/api/include/redpitaya/rp.h#L145

Examples of RF Input Output:
https://redpitaya.readthedocs.io/en/latest/appsFeatures/examples/genRF-exm1.html#code-c

Information about IOs: https://redpitaya.readthedocs.io/en/latest/developerGuide/125-14/fastIO.html

Specs of the Redpitaya: https://www.redpitaya.com/f146/specifications

Sample rates, etc: https://redpitaya.readthedocs.io/en/latest/appsFeatures/examples/acqRF-samp-and-dec.html#s-rate-and-dec

Ringbuffer: https://github.com/RedPitaya/RedPitaya/issues/100

Sigrok Wiki: https://sigrok.org/wiki/

Sigrokdecode Git: git://sigrok.org/libsigrokdecode

API Definition of V 2.0 Libsigrokdecode: https://sigrok.org/api/libsigrokdecode/0.3.0/index.html

Dokumentation Loguru: https://emilk.github.io/loguru/index.html

Dokumentation Nlohmann Json: https://json.nlohmann.me/

Procedure Initial Start

Initial setup

Insert the provided SD-card into the Red Pitaya. Connect the provided Micro-USB power supply to the Red Pitaya.

If you are working from home: Connect the provided Ethernet cable to the Red Pitaya and to your WiFi router. Go to the URL printed on the sticker on the Red Pitaya's ethernet port, i.e. „rp-fxxxxx.local/". If you get the Red Pitaya's web interface, congratulations, you are connected.

If not, you may need to go into the settings of your WiFi router, check the Red Pitaya's local IP address and type that into your router.

If you are working in the lab: Connect the Ethernet cable to the Red Pitaya and directly into your computer.

Follow the directions here to get a connection to the Red Pitaya: https://redpitaya.readthedocs.io/en/latest/quickStart/connect/connect.html.

Once you are connected, you can plug the WiFi dongle into the Red Pitaya and connect to the TI_Roboter network via the web interface. The password is „ITRobot!". Now, you can access the Red Pitaya via WiFi if your PC or phone is connected to the same WiFi network.

To then do software development via the Red Pitaya, connect via SSH. On linux, simply run „ssh root@<either your Red Pitaya's URL or IP here>". On Windows, use Putty get an SSH connection. The password is also „root".

The Windows Explorer and most Linux file managers support browsing files via SSH or more precisely SFTP or FISH. Simply google the instructions for more convenient file access.

Recurrent work

Building App

The source for the main app can be found in the „RPOSC-LogicAnalyser" folder of the project.

A web app for the Red Pitaya consists of several components. Among these are the backend code written in C++, contained in the src-Folder.

The backend code is compiled into a shared object file loaded by the webserver.

This build is performed via the Makefile contained in the root folder of the app. To perform the build, run „make INSTALL_DIR=/opt/redpitaya".

The current implementation of the RPOSC-LogicAnalyser app has two additional dependencies that must be installed for the backend to successfully compile. These are „libsigrokdecode2" and „libsigrokdecode-dev", both for the armhf architecture.

These should already be installed on the provided SD-Cards.

If you decide to wipe the sd-cards, simply install them via apt if your Red Pitaya has an internet connection or install them via dpkg from the libsigrokdecode_packages folder of the project.

The frontend consists of the index.html file and the js and css folders, containing the Javascript and CSS code respectively.

While a simple frontend for the Red Pitaya that simply consists of hand-written HTML, Javascript and CSS does not need to be built, our UI is built with Vue.js version 3.

A Vue.js project consists of several components in the form of .vue-files. These contain HTML, Javascript and CSS.

To compile the .vue-files into HTML, Javascript and CSS, npm version 6.14.11 is used. The provided virtual machine already contains a working npm installation. Username and password for the VM is both „rposc“.

The UI-code is contained in the logic-analyzer-webui-dev folder. To build it, open a terminal in that folder and run „npm install“ once. Then run „npm run build“. This creates a folder called „dist“, which contains the compiled UI. Copy that folder into the RPOSC-LogicAnalyser folder and you have the complete project which can be copied to the Red Pitaya.

Each Red Pitaya web app also loads a specific FPGA-image. The path to that FPGA-file is specified in the FPGA.conf file. More on that in chapter „Current State and Future Directions“.

To summarize:

Initial setup, done one time:

If the project is not on the Red Pitaya yet, copy the RPOSC-LogicAnalyser folder to the Red Pitaya under the path *„/opt/redpitaya/www/apps“.* If not already installed, install libsigrokdecode2 and libsigrokdecode-dev on the Red Pitaya.

Install npm on your PC or start the provided VM on your PC, go to the logic-analyzer-webui-dev folder and run „npm install“.

To build, done each time changes are made:

If changes were made to the backend, go to the RPOSC-LogicAnalyser folder on the Red Pitaya, run „make clean“ and „make INSTALL_DIR=/opt/redpitaya“.

If changes were made to the frontend, go to the logic-analyzer-webui-dev folder on your pc, run „npm run build“, then copy the newly created dist folder to the Red Pitaya under „/opt/redpitaya/www/apps/RPOSC-LogicAnalyser“, merge and overwrite if prompted.

Tips and tricks for UI development

3rd-party libraries

Bootstrap v5 (CSS-Framework):

- Has many ready-to-use CSS components and a layout-system to develop WebUIs with responsive design more easily.
- https://getbootstrap.com/docs/5.0/getting-started/introduction/

FontAwesome 5 (used for icons in the WebUI):

- https://halilyuce.com/web/how-to-add-font-awesome-to-your-vue-3-project-typescript/
- https://fontawesome.com/

SASS (CSS Pre-Processor):

- Makes it a lot easier to maintain and develop with CSS.
- https://cli.vuejs.org/guide/css.html#referencing-assets

- https://vue-loader.vuejs.org/guide/pre-processors.html#sass

Apexcharts (for generating charts):

- Zooming in apexchart does not support touch for mobile devices. Therefore, a custom zoom range slider was implemented which makes use of apexcharts zoomed event to enable zooming on mobile devices.
- https://apexcharts.com/docs/vue-charts/

vue.config.js

The vue.config.js does two things:

1. It deletes the rules of eslint (line 2-4).
   - ⇨ eslint is a node-module which is used to force clean coding-rules on the developer. It can be very annoying if you are import 3rd -party scripts which do not follow these rules in your own javascript-code.
2. It corrects the generated paths when building the productive version of the WebUI (line 5-8).
   - ⇨ If the paths are not correct, the paths in the files of the generated dist-directory might be wrong and referenced files cannot be found/loaded in the WebUI (e.g., the favicon).

```
1. module.exports = {
2.     chainWebpack: config => {
3.         config.module.rules.delete('eslint');
4.     },
5.     publicPath: '',
6.     configureWebpack: {
7.         devtool: 'source-map'
8.     }
9. };
```

More information on this topic: https://cli.vuejs.org/config/#vue-config-js

Change detection

Our UI uses Vue.js version 3. There is comprehensive documentation out there for Vue.js (https://v3.vuejs.org/guide/introduction.html). Most of the articles apply to Vue.js 3, but some only apply to version 2. In the following section we describe an issue that is so far not officially documented.

Vue.js uses the Model-View-ViewModel pattern. This works by declaring using data bindings in the UI. For example, a component might declare a drop-down menu in the following way:

```
1. <select class="form-select">
2. <option v-for="possibleSampleRate in possibleSampleRates">
3. {{ possibleSampleRate }}
4. </option>
5. </select>
```

In this case, the „script“ part of your component must have an iterable property named „possibleSampleRates“. This property can be mutated anywhere in your code and the changes will reflect in the UI. This process is known as reactivity.

Reactivity in Vue.js has some caveats that the programmer needs to be aware of. If an object is mutated in a way that Vue.js cannot detect, the changes will not be visible in the UI.

For stack allocated objects, such as numbers and strings, change detection always works. (Note: strings are most likely not actually stored on the stack, but they behave as if they were. Using Java/C# terminology: strings exhibit primitive/value type semantics, while arrays and objects exhibit reference type semantics).

Javascript also has two types of heap allocated objects, arrays and objects (which are simply hash maps in JS).

For these two types of objects, reactivity is lost when assigning a new value.

Example:

```
1. data () {
2.   return {
3.     obj: {x: "Value of property x"},
4.     arr: ["Value of index 0"]
5.   }
6. },
7. …
8. obj = {new_property: "value"}; //reactivity is lost
9. arr = ["new array"];           //reactivity is also lost
```

For arrays, simply either mutate individual indices in place, or empty the array via „splice(0)“ and add new values.

arr[0] = "new value"; //This did not work in Vue 2, but works in Vue 3
arr.splice(0);
arr.push("new value"); //reactivity is kept

For objects, you can overwrite and add properties by indexing and assigning and delete properties via the delete operator.

```
1. obj["x"] = "New value of property x." //Reactivity is kept.
2. obj["y"] = "Value of new property y." //This didn't work in Vue 2, works
    in Vue 3
3. delete obj["x"]; //Didn't work in Vue 2, works in Vue 3.
4. //Clearing an object:
5. //watch out: In other programming languages, a for loop on a hash map
6. //would give key-value pairs. In JS it just gives the keys.
7. //Same goes for for-
   loops on arrays, your iteration variable always contains
8. //the current index.
9. for (key in object)
10.{
11.  delete x[key];
12.}
```

For all the mutations that didn't work in Vue 2, the method Vue.set was provided. This method is no longer available in Vue 3.

Developing the UI without having to start the Red Pitaya

If there are fewer boards in your group than members, it is useful to be able to develop the UI without it needing to connect to the Red Pitaya.

To do this, we have developed a stub for the Red Pitaya in Node.js.

The stub accepts a web socket connection, then sends and receives the same signals and parameters as the backend application for the Red Pitaya. The parameters and signals sent by the stub contain constant example data.

The stub can be found in the „RedpitayaStub" folder. To launch it, simply run „node index" in that folder.

You can configure the UI to either connect to the stub via the mounted() method in App.vue. Instantiate an object of class RedPitaya to connect to the real device and instantiate and object of class RedPitayaStub to connect to the stub.

The stub logs all sent and received parameters and signals on stdout. This can be used as an interface specification for the data that is sent between the web UI and the Red Pitaya.

Tipps for backend development

How the backend of a Redpitaya WebApp works

Minimal webapp template can be found in the official redpitaya git:
https://github.com/RedPitaya/RedPitaya/tree/master/Examples/web-tutorial/1.template

In case of the backend the app consists of two Makefiles aswell as the main.h and .cpp.

The outer Makefile in the supports the commands „make" (which has to be called as „make INSTALL_DIR=/opt/redpitaya") and „make clean". As you can tell, the files in INSTALL_DIR provide the redpitya headers and libraries like „rp.h", „rp_app.h" and the „CustomParameters.h". The inner Makefile (located in „src") is for building the application.

„CCustomParameters.h", „BaseParamter.h" and „Parameter.h" define the classes for the CCustomParameters like CBoolParameter. They can be found in the git under „Bazaar/nginx/ngx_ext_modules/ws_server/rp_sdk/" are very important for development, because of the missing documentation.

The „main.h" file describes the api for the shared object which gets built by the Makefiles and includes the Redpitaya libraries. This api is called by the „ngnix" server which manages the webserver functionality of the Redpitaya.

The functionality is then implemented in the „main.cpp" file. Of course other .cpp files can be added, but keep in mind to add them to the inner Makefile. A wildcard like „CXXSOURCES=$(wildcard *.cpp)" can also be used to add all the .cpp files in a folder.

The main file must contain following functions:

- rp_app_desc: returns a const char pointer to a string describing the application
- rp_app_init: Gets called by the „nginx" server on loading the application. Is used for initializing the app and creating objects , etc.

Must call „rp_Init()“ to initialize the app. Do **not** use „rpApp_Init()“ like in many of the examples in the git. This has the effect, that the read pointer of the redpitaya fpga does not stop reading...

- rp_app_exit: Is like the destructor of the application and gets called, when the user leaves the website.
- rp_set_params, rp_get_params, rp_get_signals: Are internal functions, which do not have to be implemented, but have to exits!
- UpdateSignals, UpdateParams are functions which get called in the Signal/ParamInterval which can be set by calling „CdataManager::GetInstance()->SetSignalInterval(t in ms)“ repectivly „CdataManager::GetInstance()->SetParamInterval()“.

  They can be used to set the Parameters and Signals but you can do this at any other point of the application too. Setting new values is done by „CTemplateParameter::Set()“ This also set a internal flag to tell the nginx server to send the new data to the frontend.
- OnNewParams, OnNewSignals: Are the really important functions. They are called, when the redpitaya websocket received data from the frontend. To work with any data the function „Update()“ **has to be called** on every CTemplateParameter! Otherwise you simply dont get the new data into your application. „CTemplateParameter::IsValueChanged()“ can then be used to determine, which datapoint has been changed.
- PostUpdateSignals: Was not tested by us. But should should be called after a signal update.

The above mentionend CTemplateParameter define multiple Parameters and Signals from multiple cpp data types like bool, char and string (only for paramters, but can be created for signals too). The main differnece between signals and parameters is, that a parameter expects one datapoint and a signals expects multiple datapoints in form of a „std::vector<datatype>“.

Short description of current class diagram

The current class diagram which is mostly implemented in the application can be found on this CD. (Quellcode/Klassendiagramm.pdf) Classes which are marked by the stereotype „«example»“ are only examples to show how the JSON Object, which this class parses or sends are composed. They can, but don’t need to be implemented as classes.

Mainly the diagramm consits of 5 parts:

- The „ServerMain“ class which is actually the main.cpp file of the application and which provides the redpitaya apis functions like OnNewParams() or UpdateSignals(). It is also currently the place, where all the instances of other classes are created and managed. The two lists „pContainerList“ and „sContainerList“ contain all P- and SContainers to be called in a loop at the OnNewParams() and OnNewSignal() functions.
- The blue marked part of the diagram is a construct which defines the containers for all CCustomParameters. It declares the functions Update(), OnNew() and OnNewInternal(). Update() only a function which should be used to update the parameter. However OnNew() and OnNewInternal() compose the receiving and processing of data from the frontend. OnNew() is implemented in PContainer (resp. SContainer) and simply calls the Update() function on a parameter to get new data. It then calls the Objects OnNewInternal() function,

if new data is available. OnNewInternal() is then implemented in all the receiving child classes.

- The yellow part of the diagram are all the classes which are used to manage the decoders.
- The red part represents the data aquisition and the management of an acquirers options.
- The rest of the classes binds it all together. „LogicSession“ is where the magic happens: It starts the acquisition, reads back the data, puts the data to send, mixes and maps the data and starts the decoding process. The „Error“ class is for sending errors from the backend to the frontend.

Libraries used in the backend

On top of the standard redpitaya libraries we used 3 additional.

LibsigrokDecode

LibsigrokDecode forms the main part of the decodation. It is a library is open source and shared in the GNU GPL 3 license. It is written C and has also a C api, but its decoders are written in Python. The main use of the library is the free logic analyzer PulseView (https://sigrok.org/wiki/PulseView).

The main entry point for the git, documentation, building information, etc. Can be found here: https://sigrok.org/wiki/Libsigrokdecode

Because the Redpitaya apt does not supply the newest version (4) of LibsigrokDecode, version 2 is used on the device. This comes with a few api changes, so the correct api description is found here: https://sigrok.org/api/libsigrokdecode/0.3.0/index.html

When developping with LibsigrokDecode there are many contact points with gLib (Docmentaion can be found here: https://developer.gnome.org/glib/unstable/) and on reading current decoder options you get in contact with the python c-api (https://docs.python.org/3/c-api/index.html).

The Project which can be found at the CD in „Quellcode/LibTests/test“ is the development project for LibsigrokDecode and includes the normal process for decrypting signals. The following will show the main process and a few helper functions included in the project (I will not show exact function calls. Please refer to the api description for them):

- Main process:
  - call to „srd_init()“ for intialization of the sigrok library
  - creation of a session via „srd_session_new()“
  - loading of a available decoder via „srd_decoder_load()“. Available decoders can be found by running „srd_decoder_load_all()“ and „srd_decoder_list()“ or can be found in the LibsigrokDecode git under „decoders“.
  - Instatiating the loaded decoder with „srd_inst_new()“
  - Afterwards the available decoder options, channels and optional channels can be viewed.
  - Before running any decoder, a channel map has to be set via „srd_inst_channel_set_all()“. This map maps channel ids to numbers. They are relevant when passing data to the decoder.

For filling the arguments of the map, please take a look at the project. The line „GVariant *gkey = (GVariant *)(„rx“)“ is a typical example of how libsigrok sometimes uses glib in a very interesting way. Normally a GVariant is created by calling a function which also accepts a type string like „s“ for string. But here a „const char *“ just gets converted.

- After this the options have to be set. Even if you are using the default values, „srd_inst_options_set()“ has to be **at least called once.** This first run replaces the object at „srd_decoder_inst->decoder->options“ with a new python object which can be read by the decoder.
- To start a session, the samplerate has to be set too. This is done session wide and not for every decoder by calling „srd_session_metadata_set(..., SRD_CONF_SAMPLERATE, ...)“
- After this the input data has to be sanitized. A decoder accepts data in two unitsizes: 1 is for 8bit data and 2 is for 16bit. So the data has to be mapped to a integer value in the range of the unitsize. In the example this is done by finding the minimum and maximum and setting them to 0 and 255 and then converting the double values to a integer between.
- After this the data has to be arranged in the right order. Because the decoders can be used with a continous stream of data, multiple data channels can be interleaved. This is done by the order set with the channel map. (Not included in example, because only one channel was used.)
- At any point in the program there has to be set a callback for process the output data. This can be done via „srd_pd_output_callback_add()“. You can set multiple versions of callbacks. A SRD_OUTPUT_ANN is a version, where you get a annotation class and a annotation text. So it is for humand readable displaying. If you want the data in binary format to process further you could use SRD_OUTPUT_BINARY. Attention: When getting the annotation class from the decoder it is a double pointer „char **“. Also there is a difference in the api version! In the api which is used on the Redpitaya „ann_class“ is called „ann_format“.
- At last it is only a call to „srd_session_start()“ to start the session and the „srd_session_send()“ to send a package of data to the decoders.

- Convertion of available options to strings:

  The function „printOptions()“ now only prints the options, but can be simply converted to output strings or a JSON. The main points done in the function is to loop over the „GSList *“ at „srd_decoder_inst->decoder->options“ and convert the „GVariants“ in there to a printable string. Libsigrok only uses GVARIANT_TYPE_INT64, GVARIANT_TYPE_STRING and GVARIANT_TYPE_DOUBLE. So we only have to check which one it is and then call the approprate „g_variant_get_xxx()“ function.

- Setting of options from a map<string, string>:

  „setOptions()“ is the other direction of the above function. It looks in the options, which GVariant type the default value has and then converts the supplied string into this type. It then composes a key value table „g_hash_table“ to call „srd_inst_option_set()“.

- Printing of current options:

Printing the current options is done with „printCurrentOptions()“ and leaves the Glib library and enters the python c-api. It is mostly the same as „printOptions()“ but has to convert python objects to strings.

Nlohmann/Json

Nlohmann is a pretty easy to use JSON parser and serializer. In the header only version it only consists of a singel header file. Becuase this library uses a macro called JSON_ASSERT, which is also defined by the Redpitaya library, but with two parameters instead of one and also because there is a problem with cassert on the Redpitaya, simply including it does not work. This is solved by using our „jsonWrapper.hpp“ which defines NDEBUG, which removes the asserts from the nlohmann library.

The library is pretty easy to use. With nlohmann::json a new json object can be created. The elements of it can then be accesses with the [] operator containing a string (e.g. json[„id“]). Using this acessor also creates elements, if they do not exitst and sets them new if they already do.

For creating creating a string from a JSON object, the method „nlohmann::json::dump()“ is used. The other way around it is possible to parse strings into JSON Objects by calling „nlohmann::json::parse()“.

With this function we currently experience our main problem. If it is called in an OnNewParams function the apllication crashes with a „std::bad_alloc“ exception. This results also in a crash of the nginx server, which can only be solved by removing the applications shared object file completely. However calling the function in the „init“ function works without flaws.

Loguru

Loguru is simple logging manager which provides a lot of functionality. The base functionality consits of logging functions which use several levels like INFO and ERROR. It has also a configuration file which is used to set the format of the output and also add things like timestamps to the log. On top of that it supports multiple output targets like normal console, file and syslog.

Rp.h

Gives us the possibility to access the high frequency input from a higher level (C, C++). Thanks to this it is not necessary to touch the FPGA and the image of it.

The Library holds methods to set different parameters, start acquisition and get them from the ring buffer of the FPGA, it build the interface to the IOs.

Access Logs of Web Application

Redpitaya creates logfiles under „/var/log/redpitaya_nginx“

- debug.log: printfs and fprintfs from the c/c++ application
- error.log: Log from the nginx server
- rp_sdk.log: parameter and signal logs
- ws_server.log: Wbsocket logs

Known issues and workaround

The included applications on the Red Pitaya do not start, the progress ring simply keeps spinning

- If the browser's javascript console shows „Uncaught ReferenceError: AnalyticsCore is not defined", you most likely have an ad-blocker enabled in your browser. The included applications will not work with an ad-blocker. Simply disable it for your Red Pitaya's URL and reload the page.

Filesystem is read-only

Call `rw` inside of the RP console to make it writeable.

SSH connection breaks

Simple restart should work. We don't know why it happens, but maybe it's a temperature problem.

Acquiring data never stops

If you call the "rp_App_Init()" to initialize the Webapp, replace it with "rp_Init()". Until now we didn't notice any different behavior, only the acquiring process stops correctly.

To stop the acquiring process, its needed to push the FPGA image again via "cat /opt/redpitaya/fpga/fpga_0.94.bit > /dev/xdevcfg".

ws.app is null when starting up the Webapp

The name of the application (in the app.js or App.vue) must be the same as the name of the folder.

Cannot read fpga file

Maybe you modified the fpga.conf of your project in windows than every time you open the file in nano, the file gets parsed from "DOT format" and somehow the path gets corrupted. Just copy a file from a working project or build up the file from scratch but do it in Linux.

vue-cli-service: not found when building UI

Delete the node_modules folder and the package-lock.json file. Then run „npm install" and „npm run build" again

The main web UI of the Red Pitaya is stuck at „Getting List of Applications"

The backend part of the application most likely ran into an std::bad_alloc exception. This also results in nginx „worker process xxx exited on signal 6" in debug.log and error.log. To get the frontend running again run „make clean" in your application folder. We do not currently know what exactly causes the bad_alloc exceptions or how to debug them.

„ENOSPC: System limit for number of file watchers reached" when building UI

Run „`echo fs.inotify.max_user_watches=524288 | sudo tee -a /etc/sysctl.conf && sudo sysctl -p`"

Our VPN solution for „home office"

1. Build and configure a wireguard server
2. Build certificates for every client.
3. Install wireguard client software on your pc
4. Wireguard doesn't work on the redpitaya directly therefore we did a little workaround
    a. Install wireguard on a raspberry or similar (following link should contain tutorial)
        i. https://www.sigmdel.ca/michel/ha/wireguard/wireguard_02_en.html

b. Configure and connect the raspi
c. Activate vnc server on the Raspberry
d. Connect the redpitaya via lan to the raspberry
e. You should now be able to control the raspi via vncViewer
    i. It is very important to use (real vncViewer) other clients didn't work in our tests (https://www.realvnc.com/de/connect/download/viewer/)
f. And on the raspi the Redpitaya should be available
g. Finally build autostart (build in wireguard service)
    i. https://www.sigmdel.ca/michel/ha/wireguard/wireguard_02_en.html

4.1 Install (Ubuntu/Mint/Arch/Manjaro):

- `sudo apt install wireguard` (Ubuntu/Mint)
- `sudo pacman -s wireguard-tools` (Arch/Manjaro)
- Save .conf file to /etc/wireguard
- To start VPN: `wg-quick up <nameofconf>`
- To stop VPN: `wg-quick down <nameofconf>`
- If error `resolvconf: command not found` run `sudo ln -s /usr/bin/resolvectl /usr/local/bin/resolvconf`
- `sudo wg show all` shows the current connection
-

4.2 Install (Windows)

- Install: https://download.wireguard.com/windows-client/wireguard-installer.exe
- Run and import config file

Current State and Future Directions

Current State

Frontend

The following UI features are implemented:

- The whole WebUI is responsive and can be used on desktop and mobile devices.
- Apexchart is implemented and data can be visualized as stepline charts. However, the data are not sourced from the Redpitaya, yet.
- Channel names can be edited by double clicking on them.
- Zooming in and out in the charts is possible on desktop and mobile devices.

The following features are currently functional on the UI-side:

- Choosing the decoder
- Choosing decoder parameters
- Choosing decoder channels

Backend

- Code for data acquisition works
  - ➔ Class for the ACQChosenParameters should be more user friendly.
- Code for packing data into JSON format and sending it to the frontend works
- Code for parsing JSON data from the frontend does not work (std::bad_alloc)
  - ➔ This is the main problem. Would probably take very long to debug.
  - ➔ A hacky workaround could be to not use JSON for receiving data from the frontend, instead send every needed parameter step by step.
- Code for setting/loading the decoder, decoder options, decoder channels is transferred from the LibTests/test project, but not tested in redpitaya environment.
- Code in „LogicSession“ is incomplete and untested. Mainly the part for mapping the data from double to uint8_t and calling of the decoder session is not implementd.
- Usage of Logging via Loguru into the debug.log works, Logging to the frontend via the Error class is not tested.
- Compilation via Makefiles works but takes about 7 Minutes. Partial builds do not work.
  - ➔ Probably a toolchain depending on cmake would do a better job.
  - ➔ Also prebuilding loguru and nlohmann/json would be a good step to reduce compile time

Future Directions

Frontend

Zooming for Apexcharts should be made more user friendly.

A documentation page should be embedded in the WebUI.

The configurations of the Logic-Analyzer should be stored in session variables or cookies, so that they don’t get lost when refreshing the WebUI.

Choosing acquirer parameters is currently partly implemented.

The following features are currently only implemented in the form of views, no logic:

- Starting the acquisition
- Showing acquired data
- Showing decoded data

Backend

- Mainly a better compiler toolchain must be implemented. Compile times of about 7 minutes are **not acceptable** for debugging and testing.

- ➔ Usage of better toolcahin.
- ➔ Integrating of pre built libraries.
- ➔ Best option would be the integration of a cross compiler to avoid building on target.

- JSON parsing must be debugged
- Completing LogicSession class
- Restructuring SPContainer aswell as S and Pcontainer.
  - ➔ Integrating s and pDecoderList as static into the classes
  - ➔ Usage of getInstance() function to get Insatance by Ccustomparamter::name. This would simplyfy the constructor situation drastically.
- Restructuring of whole class diagram to support multiple decoders
  - ➔ Maybe integrating a srd_decoder_instance manager to store inforamtion about loaded decoders and supply functions to load/unload decoders and access the decoder instances.
- Extend the ACQChoosenOptions class to provide more easily setting of all the options.
  - ➔ Currently every single parameter has to be translated from string and set to the variable.
- Change the Acquirer to an Interface
  - ➔ Allows more easy generation of different Acquirer.
- Allow the Use to acquirer also data with the digital inputs.
  - ➔ Makes it possible to use more channels even it is slower than the high frequencies.