PocketVNA API documentation

Brief Info

PocketVNA API allows reading raw (uncalibrated data) from a device. Right now, calibration stuff is not exposed for external use. But there is simple formulas for **simple compensation**. Also, there is an excellent SKRF library for python. It has a lot of RF Math functions along with some calibration algorithms: SOLT/TOSM, LMR16, TRL…

Supported platforms

- Windows 32bit and 64bit. Tested on Window 10, Windows 8.1, Windows 7. In theory, it should work on Windows XP

- Linux distributions: tested on OpenSUSE 42; Knipix 7; CentOS7 both 32/64bit, CentOS6.8 both 32/64; Fedora 23/24 64bit; Ubuntu 12 LTS 32bit, 17, 14, 15; Ubuntu 10 LTS but may work improperly :(Pay attention that libudev is used. On elder systems there was `libudev0`, now there is newer on `libudev1`. So, for linux there are 4 package: for libudev0 x32 and x64 and for libudev1 x32 and x64
- There is a version for raspbian jessie (Raspberry PI)
- MacOS least tested version 10.12 Sierra. Even though MacOS is 64bit system, 32 bit version of API is provided Just in Case

Package Content

Distributed API archive contains following

Examples and bindings

- `pocketvna.py` -- kind of binding for python.
- `make_solt_standard.py` -- example script for collecting measurements for SOLT/TOSM calibration
- `make_lmr16_standard.py` -- example script for taking calibration standards for LMR16
- `make_simple_compensation_standard.py` -- example for collecting data for simple compensation
- `skrf_solt_calibration.py` -- example of performing SOLT (from SKRF)
- `skrf_LMR16_calibration.py` -- example of performing LMR16 (from SKRF)
- `simple_compensation_calibration.py` -- example of performing Simple Compensation (per network parameter separately)
- `enter-dfu-mode.py` -- script to enter device into DeviceFirmwareUpdate mode
- `python_example.py` -- example of using API. Should work on pythons starting from 2.7
- `pyth_unittests.py` -- mostly for internal use. I've added it as additional example of python API use
- `pocketvna.h` -- C-header file for API
- `cpp-example.cpp` -- Example of using libPocketVnaAPI on C++.
- `pocketvna-octave.cc` -- example of octave module;
- `compile-pocketvna-octave.sh` -- script to compile it
- `octave_usage_example.m` -- and example how to call it
- `CSharp_PocketVNA_Example` -- a visual studio (2015) C# project, example of using API
    - `PocketVNA.cs` -- wrapper for DLL
    - `PocketVNADevice.cs` -- wrapper for a `PocketVNA.cs`
    - `CompensationExample.cs` -- example of taking and performing compensation

    - `CompensationCollector.cs` -- example of collecting data for simple compensation
    - `CompensationAlgorithm.cs` -- example of applying compensation
    - `UsingSimpleAPIExample.cs` -- using PocketVNA.cs example

- firmware.hex -- latest firmware version
- device update

Linux Specific

- `98-pocketvna-udev.rules` -- rule file (to gain permissions for reading device) and script to install it
- `installPocketVNArule.sh` -- installer for rules file
- `README_PERMISSION` - brief information about permissions for usb devices
- dfu-program binary -- application to reprogram pocketVNA device (for linux only)

- `libPocketVnaApi[_x64|_x32].(so|dll|dylib)` -- shared object/dynamic library for our API

Using API

Api is written on pure C and distributed as dynamic library (shared object). Thus a wrapper may be written on any language that supports loading dynamic libraries. Number of such wrappers are provided. For example C++, C#, python, octave
Our Gui Application uses this API too

Workflow

- Check if any device is available. In other words **Enumerate Devices**
- If any device is available **Open** one, for example a first one
- Get device information like:
    - **Characteristic Impedance** `(`**`Z0`**`).` Right now device should return it as `50Ω.`
    - **Valid Range** -- if needed. A frequency range device can digest. Api accepts frequency of this range without error, but results may be imperfect. For example, right now it is **`[1Hz; 6GHz]`**
    - **Reasonable Range** -- a frequency range that produces good results. It is narrower than **Valid Range**. For newer firmware it is **`[100KHz; 6GHz],`** for elder **`[500KHz; 4GHz]`**
    - **`Supported Network parameters`** -- elder devices (with elder firmware) were **One Path**: only S11 and S21. Newer devices (with newer firmware) are full network (S11/S21/S12/S22)

- Form an array of frequency points that are interesting for us. Take into account that, that it should fit into **Visible Range**, but it would be better to fit it into **Reasonable Range**.
- Call scan for these frequency points
- Close device if it is not required

Each function returns result code. If query is succeeded it should be OK. Otherwise, it returns another code. If device is disconnected, any api call should return InvalidHandle.

Function Descriptions

Pay attention, that each function is prefixed with "**pocketvna_**". As a complex number Sparam struct is used

Driver routines:

- `"driver_version"` -- this function fills a first parameter (uint16) with driver version, and second parameter (double64) with number PI. Also, this function always returns status OK. It is done for testing purposes, thus you can check if binding works properly. *It is not required to call this function*
- `“close”` -- It would be nice to call it when you do not need the library anymore. But it is not strictly necessary. This function clears internal resources. But, **pay attention**, on some OSes if you close driver but then use it for another subroutine the crash may happen

Connection Management:

- `"list_devices"` -- fills the first address with an array of **device descriptors** (DeviceDesc structure). Second parameter is filled with size. It returns status either Ok (if there is any device) or NoDevice (no device available).
- `"free_list"` -- frees memory taken with **device descriptors**. Notice, that you can copy descriptor into process memory, thus you does not need to store device descriptors during a long period of time
- `"get_device_handle_for"` -- In other words, “open device” command. Takes a **device descriptor** as first parameter and fills the second parameter with a **device handle. Device handle** will be used for any further device specific function. Returns statuses (most:
    - BadDescriptor -- invalid descriptor is passed
    - NoAccess -- linux specific, it does mean that device looks available but you have no permission to access it. See README_PERMISSION (permissions/README)
    - FailedToOpen -- is is failed to open for some “unknown” reason
    - Ok -- connected. **Device handle** must become valid and should not be equal to Zero

- `"release_handle"` -- in other words “close device”. In theory, always returns Ok. After call passing **Device handle** must become zero.

Device work:

All functions accept **Device handle** as first parameter. If device is disconnected, or handler is not valid all these functions return InvalidHandle.

- `"is_valid"` -- checks if **Device handle** is valid. Returns InvalidHandle or Ok. Zero handler is InvalidHandle too
- `"version"` -- firmware version
- `"get_characteristic_impedance"` -- fills second parameter (double64) with Characteristic Impedance (Z0). It should be 50Ω
- `"get_valid_frequency_range"` -- frequency range device can accept
- `"get_reasonable_frequency_range"` -- a frequency range that produces good results
- `"is_transmission_supported"` -- check if Network parameter (transmission/mode) is supported. Elder devices (with elder firmware) were **One Path** (S11 and S21 only). Newer firmwares are full network. Second parameter is one of NetworkParam items:
    - Ok -- network parameter is supported
    - UnsupportedTransmission -- network parameter is not supported. For example S12 on elder devices
    - BadTransmission -- unrecognized second parameter
- `"single_query"` -- take measurement for a single **frequency point**. Probably, it is better to use multi_query for it is more optimized.
    - Parameters
        - **frequency** -- uint64. Should be inside **valid range**
        - **average** -- third parameter, how many times to read network parameter (and take average for that)
        - **Network Parameters** -- S11, S21, S12 or S22. Pay attention, it is possible to combine them with bitwise or statement, like: S11|S22 -- take only S11 and S22.
        - 4 network parameters (s11, s21, s12, s22) -- each network parameter will be stored into corresponding address. If Network Parameters do not contain any, address will be unchanged.
- `"multi_query"` -- similar to `"single_query"` but for several **frequency points**. It is preferable to use this one
- `"enter_dfu_mode"` -- puts device into DFU mode (device firmware update). Next you can rewrite a firmware using dfu-programmer or FLIP software (available on windows)

Debug purposes:

These functions are for internal and debug/testing purposes. Probably, you do not it.

- `"debug_request"` -- There is some buffer inside a device. It can store some additional information. Most time it is unused. This request fills it with some data using built-in pattern
- `"debug_read_buffer"` -- reading buffer
- `"debug_response"` -- for testing purposes, for example to check binding. Accepts two arrays (last 2 parameters) of a particular size (2nd parameter) and fills it using pattern
  - `Fills p1/p2 using pattern ( i -- zerobased index )`
  - `    p1[i].real = Pi / (i+1)`
  - `    p1[i].imag = 1. / p1[i].real`
  - `    p2[i].real = Pi * i`
  - `    p2[i].imag = Pi ^ (i+1)`

Provided examples and bindings

Please treat examples as Examples, not as high quality production code. Use them carefully. Probably later them will be well tested production code

C

Consists of “`pocketvna.h`” header file. Pay attention that MinGW-GCC is used on windows. This is most tested thing, for we use it on our GUI project. Example of using can be found in “`cpp-example.cpp`” and “`pocketvna_octave.cc`”.

Python

Binding is in “`pocketvna.py`”. It would be nice if you have **numpy** and **skrf** installed. If **numpy** is not installed then API returns python list as results. If **numpy** is installed API returns **numpy’s** arrays and this is much more convenient for you.
Please pay your attention on skrf (http://scikit-rf.readthedocs.io) library. It has a lot of interesting RF related routines along with a variety of calibration algorithms. Check python scripts inside package to learn how to use it. Basic example “python_example.py”

C#

`CSharp_PocketVNA_Example` folder contains Visual Studio (2015) solution for C# project. “PocketVNA.cs” -- simple binding, “PocketVNADevice.cs” -- wrapper for “PocketVNA.cs”. More convenient for use

Octave

There is no possibility in octave to load C dll(so, dylib). It can be done with C++ api.
In linux it is required to install `Octave development package` and `patchelf`.

Example of API binding is inside: “`pocketvna-octave.cc` “

There are Instructions for compilation in "`compile-pocketvna-octave.sh`"

https://www.gnu.org/software/octave/doc/interpreter/index.html#SEC_Contents
https://www.gnu.org/software/octave/doc/interpreter/Oct_002dFiles.html#Oct_002dFiles
Example of using such a binding is "`octave_usage_example.m`"

Simple Compensation

Algorithm is applied to each parameter separately. It means you don't need full network

Reflections

$$Z_{dut} = Z_{std}\frac{(Zo - Zsm)\,(Zxm - Zs)}{(Zsm - Zs)(Zo - Zxm)}$$

- `Zs` -- calibration `Z11` (impedance, complex) short parameter
- `Zo` -- calibration `Z11` (impedance, complex) open parameter for particular frequency point
- `Zsm` -- calibration `Z11` (impedance, complex) load parameter for particular frequency point
- `Zxm` -- calibration `Z11` (impedance, complex) measurement data for particular frequency point
- `Zstd` -- reference resistance. Usually 50

** Pay attention this formula requires Impedance, whereas API returns S: S <=> Z

$$Z = Z_0\frac{1+S}{1-S}\text{ , where } Z_0 = 50$$

$$S = \frac{Z - R_0}{Z + R_0}$$

Transmission/Through mode

$$Sdut = \frac{Sxm - So}{S_{thru} - So}$$

- `Sxm` -- S21 (S12) measurements data
- `So` -- S21 (S12) open data taken from calibration data for particular **frequency point**
- S_{thru} -- S21 (S12) through data taken from calibration data for particular **frequency point**

Implementation

C#

There is `pocketvna.PocketVNADevice.Math` class which have 2 functions: `ReflectionCompensation` and `TransmissionCompensation`. They implement these formulas
"`CompensationCollector.cs`" contains example of collecting data for compensation algorithm.
"`CompensationAlgorithm.cs`" represents example of applying compensation algorithm

Python

"**`make_simple_compensation_standard.py`**" -- example of collecting measurements for **Simple Compensation**
"simple_compensation_calibration.py" -- example of applying these compensation formulas