PocketVNA API documentation

Brief Info

PocketVNA API allows reading raw (uncalibrated data) from a device. Right now, some calibration stuff is not exposed for external use. But there are simple formulas for **simple compensation** (they are exposed Just in Case in API). 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`...
PocketVNA API is provided as dynamic link library (shared object) with `*.dll` extension (on windows; on linux it is `*.so` and on macos -- `*.dylib`)

Supported platforms

- **Windows** 32bit and 64bit. Tested on Windows 10, 8.1 and 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 older systems there was `libudev0`, now there is newer on `libudev1`. So, for linux there are 4 versions: for libudev0 x32 and x64 and for libudev1 x32 and x64
    - So **Modern** distributions contain libudev1.so
    - **Old** distributions contain libudev0.so
    - Some distribution has actual and alive repos that provide support for elder libraries
- Some PocketVNA devices have a firmware with VCI-usb interface support. So API's for linux and windows may require a **`libusb`** library. In this case a bundle for Windows will provide it. Linux may require a **`libusb`** to be installed
- There is a version for **raspbian** jessie/stretch (Raspberry PI). For now, is provided by demand
- **Mac-OS** minimum tested version is 10.12 Sierra. Apple cancels support 32bit version, so 32 bit version will be available for older versions only on demand (and not guaranteed)

Package Content

Distributed API archive contains following

Examples and bindings

- **`pocketvna.py`** -- binding for python.
- Python examples:
    - SOTL/TOSM calibration

        - **make_solt_standard.py** -- example script for collecting measurements for SOLT/TOSM calibration. This script collects data for SOLT calibration along with Raw Measurements and stores them into touchstones files.
        - **skrf_solt_calibration.py** -- example of performing SOLT calibration (implementation from SKRF library). It takes data from touchstones files from **make_solt_standard.py** script
    - LMR16 calibration
        - **make_lmr16_standard.py** -- example script for taking calibration standards for LMR16 along with raw measurements. Taken data is stored into touchstone files
        - skrf_LMR16_calibration.py -- example of performing LMR16 (from SKRF). It loads data from **make_lmr16_standard.py** script
    - Simple compensation
        - **make_simple_compensation_standard.py** -- example for collecting data for simple compensation
        - **simple_compensation_calibration.py** -- example of performing Simple Compensation (per network parameter separately)
    - **enter-dfu-mode.py** -- script to enter device into Device-Firmware-Update mode. Then flip software or dfu_programmer may be used.
    - **force-unlock-devices.py** -- script to force unlock devices. Unfortunately, it is not a good idea to connect several concurrent connections to the same "PocketVNA" device simultaneously. On some Operating Systems (MacOS) it is performed automatically and a program/connection has exclusive access rights. But on others OS-es (Linux) do not restrict access so we need to imitate it manually. But if a program is crashed unexpectedly or killed a lock may survive. To remove such a lock this program may be used. It calls **force_unlock_devices** function
    - **python_example.py** -- example of using API. Should work on pythons starting from 2.7
    - **python_simplified_example.py** -- example of using simplified connection "**get_first_device_handle**".
    - **pyth_unittests.py** -- mostly for internal use. I've added it as an additional example of python API use
- **pocketvna.h** -- C-header file for API
- **api_cpp_example** -- Example of using libPocketVnaAPI on C++:
    - Project file is qmake's "cpp-simple-example.pro"
- Octave binding Example (Just an example)
    - **pocketvna-octave.cc** -- example of octave module. For now it is just an example. Nothing more
    - **compile-pocketvna-octave.sh** -- script to compile octave module
    - **octave_usage_example.m** -- example of using octave pocketvna module. Just an example

- **`CSharp_PocketVNA_Example`** -- a visual studio (2015) solution, example of using API. More details in the C# section below.
- **`firmware_hex`** -- firmware versions
- device update

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

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). Probably, it will not work for customer device. Not all devices support the feature

Using API

Api is written using pure C and distributed as a dynamic library (shared object). Thus a wrapper may be written in 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. Looks kind of test

Workflow

Take measurements

Connect against the device. First:

- Check if any device is available. In other words **Enumerate Devices**
- If any device is available **Open** one, for example a first one. Unfortunately, on some systems it is hard to distinguish pocketVNA devices. So for now, use a first one

Or, second way, Instead of the previous 2 steps it is possible to use a newer helper function **`get_first_device_handle`**. Which returns Ok status with a valid device handler if connection open successfully

Next

- If it is required, get device information like:
  - **Characteristic Impedance** (`Z0`). Right now the 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 the 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 the **Valid Range**. For some devices now it is `[500KHz; 4GHz]`
    - **Supported Network parameters** -- older devices (with older firmware) were **One Path** (forward only; S11 and S21). Whereas, newer devices (with a newer firmware) have full network support (S11/S21/S12/S22)
- Form an array of frequency points that are interesting. Take into account, it should fit into the **Visible Range**, but it would be better to fit it into **Reasonable Range**.
- Call scan for these frequency points
- API returns raw non-calibrated data. So it must be calibrated or compensated. API provides Simple Compensation . Pay attention to what is the calibration for "pocketVNA" device.
- Close device if it is not required anymore.

Each function returns result code. If the query is successful it should be OK. Otherwise, it returns another code. Each code can be converted into a string representation. If the device is disconnected, any api call should return InvalidHandle.

Compensation/Calibration Data

The workflow for taking the compensation data is similar to Taking measurements above. You may organize it in a separate application and store the data into a file. Then, when you need to calibrate the raw measurements you just have to load the compensation data and pass it into Formulae (reflection or transmission) or into functions provided

Function Descriptions

Pay attention, that each function is prefixed by "**pocketvna_**". As a complex number, "SParam" struct is used. In theory, it is compatible with `double complex` of C language (C99 type)

Driver routines:

- `"driver_version"` -- this function fills the first parameter (uint16) with driver version, and the 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
- `"result_string"` -- get string representation for ResultEnum code. Almost each function returns ResultEnum. This function returns text description for it

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 the first parameter and fills the second parameter with a **device handle.** If a device descriptor contains a path equal to "simulation" then a **simulation handle** will be returned. **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 -- it is failed to open for some “unknown” reason
    - Ok -- connected. **Device handle** must become valid and should not be equal to Zero (null, NULL, None so on)
- `"release_handle"` -- in other words “close device”. In theory, it always returns Ok. After the call passing **Device handle** must become zero.
- “`get_first_device_handle`” -- just a helper For simplicity. The intended workflow is: call `list_devices`. If there is any device then call `get_device_handle_for` and freeing list’s memory by `free_list`. But sometimes it is not convenient, especially for some languages because it requires definition of complex structures wrappers. This function opens any device if available. It accepts just an address of the handler. Function returns the same result as `get_device_handle_for`.

A simulation handle -- imitates a device which returns random data.

Device work:

All functions accept **Device handle** as the first parameter. If the device is disconnected, or the 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 the 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**. 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 the 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
- `"multi_query_with_cproc"` -- like `"multi_query"`. But in contrast to `"multi_query"` it accepts a simple C function as progress callback. Previous function accepts struct with function pointer, to imitate function-objects
- `"enter_dfu_mode"` -- puts the device into DFU mode (device firmware update). Next you can rewrite a firmware using dfu-programmer or FLIP software (available on windows)
- `"range_query"` -- in contrast to `"multi_query"` it takes start and end frequencies. 2 distributions are available: Linear (logspace) and Linear (linspace)

Calibration purposes:

These functions expose a simple compensation (calibration) algorithm (see below). This compensation is applying for each network parameter separately. Pay attention, that frequency points for Raw Measurements and Calibration data should be the same

- `"rfmath_calibrate_reflection"` -- implementation of compensation for reflection parameter (S11 or S22).
- `"rfmath_calibrate_transmission"` -- implementation of compensation for transmission (S21 or S12)

Debug purposes:

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

- `"debug_request"` -- There is some buffer inside a device. It can store some additional information. Most of the 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 -- zero based 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)`
- `force_unlock_devices` -- unlocks locked devices. To limit the number of connections to a single device a semaphore is used. But if an application crashes or it is killed it may cause semaphore leak. Mostly for linux. This function resets the semaphore.

Provided examples and bindings

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

C

Consists of "**`pocketvna.h`**" header file. Pay attention that on Windows MinGW-GCC is used. This is the most tested binding, for we use it on our GUI project.
Examples of using can be found in "`api_cpp_example`" and "`pocketvna_octave.cc`".
API has a Simple Compensation algorithm exposed. **Pay attention**: Now, these 2 examples should be treated as examples. They are not up-to-date.

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 a python list as results. If **numpy** is installed API returns **numpy's** arrays and this is much more convenient.
Please pay attention to 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 the package to learn how to use it. Basic example "**`python_example.py`**" and "**`pyth_unittest.py`**". Examples for calibration algorithm are provided too
In a few words:

- Make sure "**`pocketvna.py`**" and PocketVnaApi (.dll, .so, .dylib) are in the same directory
- Import "**`pocketvna.py`**"
- Connect device
- Read raw data

C# example

`CSharp_PocketVNA_Example` folder contains Visual Studio (2015) solution for C# project. "`PocketVNA.cs`" -- simple binding, And "`PocketVNADevice.cs`" -- wrapper for "`PocketVNA.cs`" that is more convenient for use.
To use PocketVNA API

- just import "`PocketVNA.cs`" and "`PocketVNADevice.cs`"
- Add reference to "`System.Numerics`" which provides Complex numbers support
- Maker sure `PocketVnaApi.dll` is available. (PocketVnaApi_x32.dll or PocketVnaApi_x64.dll depending on selected bitness). It is 13-19 lines in "`PocketVNA.cs`". It can be done either by hardcoding path to library or by modifying variable: Solution Explorer > Properties > Debug > Working Directory

The solution contains following projects

1. `PocketVNA_Example` -- it contains binding files and some examples of using it
    a. `PocketVNA.cs` -- binding to PocketVNA.dll
    b. `PocketVNADevice.cs` -- wrapper for `PocketVNA.cs`.
    c. `UsingSimpleAPIExample.cs` -- example of using `PocketVNA` class
    d. `UsingBuiltInCalibrationExample.cs` -- example of using compensation algorithm provided by PocketVNA.dll
    e. CompensationCollector.cs -- example of collecting calibration data for compensation algorithm
    f. `CompensationAlgorithm.cs` -- using C# implementation of compensation algorithm
2. `Example_OpenDeviceLocally` -- demonstrates that device can be opened only when needed and closed immediately
3. `Example_CollectFullCalibrationData` -- Example how to take calibration data and dump it into a **simplest binary file**
4. `Example_TakeMeasurementsAndCalibrateOverDumpedData` -- Demonstrate how to take measurements and calibrate them over calibration data loaded from the **simplest binary file**.
5. `Example_TakeMeasurementsAndPlot` -- GUI example. **Please note** that it uses **Microsoft Chart Controls** library. The example demonstrates a simple window with a chart. It is possible to initiate scan (if device present); It allows switching port to be shown on a chart; It also allows switching between formats. If the **simplest binary file** made by `Example_CollectFullCalibrationData` is found then it is possible to switch between calibrated and uncalibrated data.

Octave

There is no possibility in octave to load C PocketVnaApi.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 in "`octave_usage_example.m`"
These files are just examples and may be very imperfect. Now it is rather a proof of concept. May be enhanced on demand

LabView

LabView examples are available separately. NI LabView 2018 x32 bit for Windows is used.
There are two examples of using API:

1. `_Example_GetForSinglePoint.vi` -- taking measurements for a single frequency step. Pay attention on Parameters enum: S11, S21, S12, S22, Full can be selected
2. `_Example_GetProperties.vi` -- getting some properties from a device such as frequency range, Z0, version, supported network parameters
3. `_Example_ListScan` -- collect data for frequencies list and plot S11 on a chart. It uses *Ramp Pattern* for generating `linspace` frequency. This feature may be unavailable on Base version
4. `_Example_DebugTestBindingWorks.vi` -- for debug purposes. I may help to test compatibility on linux or 64bit
5. **Disclaimers**
    a. The example is made on the 32bit version. In case of using the 64bit version make sure that all handlers are "Unsigned Int 64". Because by default LABVIEW uses a 32bit integer that may lead to bugs and crashes on 64bit version. I've changed the pointer type to "Pointer sized Unsigned Int". Hope it works but is it not tested
    b. Connection to `pocketVNA.dll` is rigid. There are two folders: `dll_x32` and `dll_x64`. Each one contains its own `PocketVNA.dll`. Opening on the 64bit version requires changing path to `pocketVNA.dll`: choose `dll_x64/PocketVNA.dll`.
    c. The example is made using NI LabView 2018. For the older version I can use the "Saving for a Previous Version" feature on demand.
    d. Make sure that you do not have several LabView examples unpacked. It may happen that the current example references the older version of `PocketVNA.dll`.
    e. **I'm labview-ignorant**. So treat these examples as "Just Examples". They are ugly, have a bad look

The simplest workflow to scan anything

- Call **`pocketvna get first device handle.vi`**
- If getting device handle succeeded (error code is `0` and handle is not `0`)
    - Call `pocketvna single query.vi` for each frequency point that is interesting for you

    - If device is not required anymore call **pocketvna release handle.vi**

The simplest workflow to get some device properties

- Call **pocketvna get first device handle.vi**
- If getting device handle succeeded
    - Call `get characteristic impedance.vi` for Z0/Characteristic Impedance
    - Call `pocketvna get reasonable frequency range.vi` to get a reasonable frequency range. It is a range which device produces
    - `pocketvna get valid frequency range.vi` -- get valid frequency range for a device
    - `pocketvna is transmission supported.vi` -- check if transmission supported
    - If device is not required anymore call **pocketvna release handle.vi**

It is very important to “release handle”. Especially on linux systems. Especially if you don’t want to re-open LABVIEW each time.

Simple Compensation

Algorithms are applied to each parameter separately. It means you don’t need a full network. You can collect data that is interesting for you and omit others.

What is a calibration/compensation for pocketVNA device

One Important Note:

1. For our device a calibration does not belong to the device itself. Rather it belongs to measurements. So probably, it would be better to use “Compensation” instead of “Calibration” term.
2. Calibration data taken for one device may be incompatible with another device.
3. When you take measurements from the device you receive raw, calibrated and "meaningless" data. These raw measurements must be calibrated (or compensated).
4. When you take calibration data nothing will be changed inside the device. Nothing will be changed inside the API.
5. Neither the device nor the API does not know whether calibration data are taken or not.
6. To calibrate these raw measurements you have to apply a calibration/compensation algorithm. You have to pass the calibration data and the raw measurements into formula
7. By taking calibration data you do not affect the next scan.
8. After taking calibration data you'll receive very similar raw measurements as were before
9. That's why you don't have to make a "calibration" each time you run an application or connect a device

Reflection mode

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

`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. 50 by default

* Taken from “The Impedance Measurement Handbook” Agilent Technologies
** Pay attention this formula requires Impedance API returns Gamma (S): S <=> Z

$$Z = Z_0 \frac{1 + Gamma}{1 - Gamma}$$, where $Z_0 = R_0 = Z_{std}$ `(Usually 50Ω)`

$$Gamma = \frac{Z - Z_0}{Z + Z_0}$$

Transmission/Through mode

$$S_{dut} = \frac{S_{xm} - S_o}{S_{through} - S_o}$$

- S_{xm} -- S21 (S12) measurements data
- S_O -- 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

Pay attention: python has an excellent skrf library which provides a variety of routines for operations over Rf-Networks along with calibrations algorithms: TRL, LMR16, TOSM so on

These compensation functions are exposed in api. Pay attention, that functions in API accept S parameters. No conversion is required. In other words:

- collect calibration data and store them **as is**
- Read raw measurements and store them **as is**
- Pass needed parameters **as is** into these functions.
- Get calibrated data

C/C++

Functions are exposed in api. See “`api_cpp_example/calibration-example.cpp`”

C#

It contains 2 examples:

1. Using built-in api that can be found in "`UsingBuiltInCalibrationExample.cs`". It is designed as a test. So it is more preferable to rely on this example.
2. Another, is an example of implementation of the simple compensation formulas on C#. There is `pocketvna.PocketVNADevice.Math` class which has 2 functions: `ReflectionCompensation` and `TransmissionCompensation`. They implement those formulas above
    - "`CompensationCollector.cs`" contains example of collecting data for compensation algorithm
    - "`CompensationAlgorithm.cs`" represents example of applying the algorithm

Python

Using built-in api functions are exposed in "**`pyth_unittest.py`**". See class **`TestRfMathCalibration`**
Also there are some additional examples of collecting data and calibrating/compensating it
Pay attention numpy and skrf are required

Simple Compensation

- "**`make_simple_compensation_standard.py`**" -- collecting measurements
- **"`simple_compensation_calibration.py`" --** example of applying those compensation formulas

FAQ

- **Is it possible to connect two devices for the same application?**
- **Is it possible to select between two devices that are connected simultaneously**
    - Yes it is. But there are some notes:
        - DeviceDesc should be a valid path. Otherwise vip/pid will be used which is the same for all devices.
        - Some patches are required for C# and python bindings.
- **Is lower level protocols available**

Rather No. Device's firmware is pretty simple: it accepts frequency points and returns S-parameter codes for each one. Decoding, enhancements and fixes (also depending on firmware/device version) are hidden inside the API library. Exposing them is not a good idea.

Moreover, Firmware is harder to update, so in case of some issue it is easier to update the API than the firmware. Providing support of such protocols would be a headache.