VNA Control 5
Programmer's Guide
C++ Edition

LA Techniques Ltd

Chancerygate Business Centre, Unit 5
Red Lion Road, Surrey KT6 7RA, UK
VAT no GB 689 4720 79
Registered in England No 3356289 Registered Office as above

Tel: 0208 3973150
Fax: 0208 3975372
E-mail: info@latechniques.com
Web site: www.latechniques.com

Contents

Chapter 1

Introduction

1.1 Welcome

The Vector Network Analyzers from LA Techniques are compact, high-performance measuring instruments. A VNA works by driving a swept sine wave test signal into one port of a DUT and measuring the reflected (and optionally the transmitted) signals. The signal may then be injected into the other port of the DUT for a further set of measurements. The VNA then uses these measurements to calculate the DUT's S-parameters.

This manual describes the operation of the VNA Control 5 libraries available to support the PicoVNA 106 and PicoVNA 108 Vector Network Analyzers. The key intention of the libraries is to provide remote automation under standard programming environments such as C++ or Python, or specialist test and measurement, scientific and math environments such as National Instruments LabVIEW and MathWorks MATLAB.

1.2 Requirements

1.2.1 PicoVNA 5 compatible instrument

The VNA Control 5 Software Development Kit (SDK) requires a VNA instrument that is compatible with the VNA Control 5 software. To check if your instrument is compatible, download and run the VNA Control 5 software. You will have the opportunity to upgrade your instrument using the VNA Control 5 software if it is not currently compatible. If your VNA instrument is not compatible with the VNA Control 5 software, you can use the SDK in demonstration mode for evaluation purposes.

1.2.2 Supported operating systems and platforms

- Windows 10 and 11 (x64)
- Linux (x64 and aarch64)
  The following list of distributions and platforms have been tested, although the PicoVNA 5 SDK is designed to work on any Linux distribution released within the last ten years: Debian 8 ("jessie") and later versions (x64), Ubuntu 18.04 (LTS) and later versions (x64), Linux Mint Cinnamon 21.1 (Vera) and later versions (x64), openSUSE Leap 15.0 and later versions (x64), Fedora 28 and later versions (x64), Arch Linux (x64), Raspbian (Raspberry Pi aarch64)
- macOS (Intel x64 and ARM aarch64)

1.2.3 Programming language requirements

- Either the VNA Control 5 SDK (downloadable) or an installed copy of the VNA Control 5 software
- Either an IDE that will open a CMake Project (such as Microsoft Visual Studio 2017 or later), or a C++ compiler supporting C++11 and CMake > 3.20.

1.2.4 Windows-specific requirements

On Windows, if the VNA Control 5 software is not installed, the installer `VNA5_SDK.exe` (obtained as part of the downloadable SDK) must be run. This installer installs the kernel drivers required to communicate with the PicoVNA instruments. This step is not required if the VNA Control 5 software is installed (as the drivers are installed as part of the VNA Control 5 software installation process). Running an installer for the SDK is not required on Linux or macOS.

1.2.5 Linux-specific requirements

On Linux, a `udev` rule must be installed. If the VNA Control 5 software is installed, this will be done automatically. Otherwise, create a file called `96-picovna.rules` in your udev rules directory (typically `/etc/udev/rules.d`) with the following contents:

```
ATTRS{idVendor}=="0ce9", MODE="0666"
ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6015", ATTRS{manufacturer}==
"FTDI", ATTRS{product}=="FT240X USB FIFO", MODE="0777" RUN="/bin/bash -c
 'echo \"${DEVPATH}\" | /bin/sed -E \"s,.*/([0-9.-]+)(/.*)?,\\1:1.0,\" >
 /sys/bus/usb/drivers/ftdi_sio/unbind'"
```

You can also copy this file directly from the downloadable SDK.

If you wish to use the VNA instrument immediately after copying this file, you may need to run the following command to reload the `udev` rules:

```
sudo udevadm control --reload-rules && sudo udevadm trigger
```

1.3 Obtaining the SDK

The SDK is distributed as a zip file, which can be downloaded from `https://www.aairobotics.com/vna5_internal_builds/temp/vna5_sdk.zip`.

The SDK contains all the programming language-specific files that you need to programmatically control your VNA instrument. It does not contain example programs, which can be obtained from GitHub (see Section 1.4).

If you have the VNA Control 5 software installed, it is not necessary to obtain the SDK separately as all required headers, libraries, drivers (Windows) and udev rules (Linux) are already installed as part of the VNA Control 5 installation process.

1.4 Example programs

Example programs using the API can be downloaded from `https://github.com/LA-Techniques/VNAControl5_SDK_Examples`.

1.5 Software license conditions

The material contained in this release is licensed, not sold. LA Techniques Limited grants a license to the person who installs this software, subject to the conditions listed below.

Access. The licensee agrees to allow access to this software only to persons who have been informed of these conditions and agree to abide by them.

Copyright. AAI Robotics Limited and LA Techniques Limited jointly claim the copyright of, and retains the rights to, all SDK materials (software, documents, etc.). You may copy and distribute SDK files without restriction, as long as you do not remove any copyright statements. The example programs may be modified, copied and distributed for the purpose of developing programs to collect data using Pico products.

Liability. AAI Robotics Limited and LA Techniques Limited and their agents shall not be liable for any loss, damage or injury, howsoever caused, related to the use of Pico Technology equipment or software, unless excluded by statute.

Fitness for purpose. As no two applications are the same, LA Techniques Limited cannot guarantee that its equipment or software is suitable for a given application. It is your responsibility, therefore, to ensure that the product is suitable for your application.

Mission-critical applications. This software is intended for use on a computer that may be running other software products. For this reason, one of the conditions of the license is that it excludes use in mission-critical applications, for example life support systems.

Viruses. This software was continuously monitored for viruses during production, but you are responsible for virus-checking the software once it is installed.

Support. If you are dissatisfied with the performance of this software, please contact our technical support staff, who will try to fix the problem within a reasonable time. If you are still dissatisfied, please return the product and software to your supplier within 14 days of purchase for a full refund.

Upgrades. We provide upgrades from our website at `www.latechniques.net`. We reserve the right to charge for updates or replacements.

1.6 Trademarks

Pico Technology is an internationally registered trademark of Pico Technology Ltd. PicoVNA and PicoSDK are registered trademarks of Pico Technology Ltd.

Windows, Excel and Visual Basic for Applications are registered trademarks or trademarks of Microsoft Corporation in the USA and other countries. LabVIEW is a registered trademark of National Instruments Corporation. MATLAB is a registered trademark of The MathWorks, Inc.

1.7 Contacting us and obtaining support

LA Techniques Limited provides tailored technical and sales support to all customers. Contact us via our website, `www.latechniques.net`, or using the details below.

Technical support forum: `https://picotech.com/support/forum141.html`

United Kingdom
(global headquarters)
Pico Technology
James House
Colmworth Business Park
St Neots
Cambridgeshire
PE19 8YP
United Kingdom

+44 (0) 1480 396 395
`support@picotech.com`
`sales@picotech.com`

UK office hours:
09:00 to 17:00 (UK time)
Monday to Friday

North America regional office
Pico Technology
320 N Glenwood Blvd
Tyler
TX 75702
United States

+1 800 591 2796
`support@picotech.com`
`sales@picotech.com`

North America office hours:
09:00 to 17:00 (Central time)
Monday to Friday

Asia-Pacific regional office
Pico Technology
Room 2252, 22/F, Centro
568 Hengfeng Road
Zhabei District
Shanghai 200070
PR China

+86 21 2226-5152
`pico.asia-pacific@picotech.com`

Chapter 2

Getting started

2.1 Getting started

The best way to get started is by running the example programs. See Section 1.4 for details on how to obtain them.

If you do not have the VNA Control 5 software installed, ensure that you have installed any necessary drivers and/or udev rules (see Sections 1.2.4 and 1.2.5).

In this section, we will build and run the example program `01_simple_frequency_sweep`. The procedure for building and running the other example programs is analogous.

The `CMakeLists.txt` file bundled with the SDK examples assumes that the headers and libraries required are installed to the standard locations by the PicoVNA 5 installer. If you do not have the VNA Control 5 software installed and the required headers and libraries are installed to a different location, you will need to modify the `CMakeLists.txt` files accordingly.

2.1.1 Linux and macOS

To build the example program, execute the following at the command line:

```
cd /path/to/01_simple_frequency_sweep
cmake .
make
```

To run the example program:

```
./01_simple_frequency_sweep
```

Note that the following dynamic libraries are required: `libvna`, `libftd2xx`. If they are not already in a directory on the dynamic library path, run the example program with (on Linux):

```
LD_LIBRARY_PATH=/path/to/libraries ./01_simple_frequency_sweep
```

On macOS, replace `LD_LIBRARY_PATH` with `DYLD_LIBRARY_PATH`.

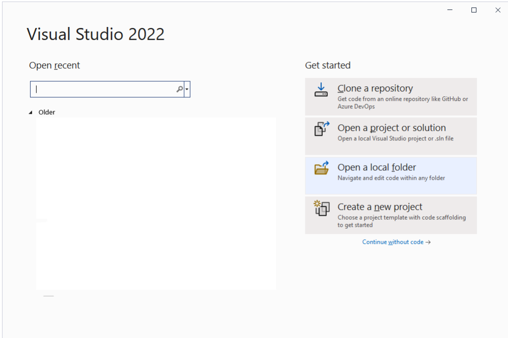


Figure 2.1: *Open a local folder* option in the Microsoft Visual Studio IDE.

2.1.2 Windows

TIP!	On Windows, make sure to build your application with a RELEASE configuration when using the PicoVNA 5 API. Building with DEBUG configurations may result in unexpected or incorrect behaviour.

2.1.2.1 Method 1: via Visual Studio

In Visual Studio (or similar IDEs), use the *Open a local folder* option (shown in Figure 2.1) to open the directory `01_simple_frequency_sweep`.

Visual Studio will automatically configure the CMake project, and the project can then be built and run in the usual way.

2.1.2.2 Method 2: via command line

To build the example program from the command line, you will need to ensure that `CMake` is installed. `CMake` can be obtained from `https://cmake.org/download/`.

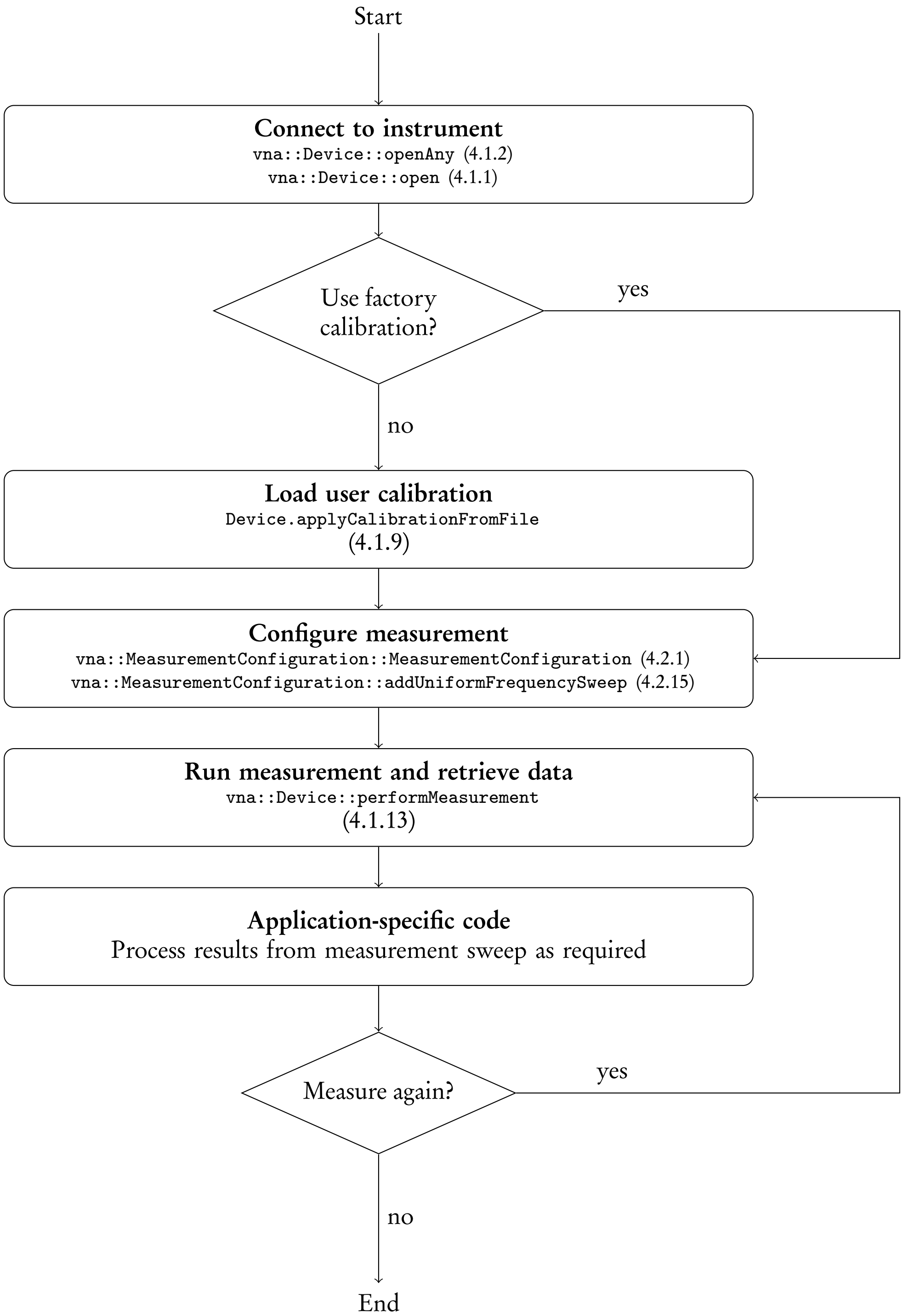
Then, run (from Powershell, ensuring that `CMake` is findable on `$env:PATH`):

```
cd /path/to/01_simple_frequency_sweep

cmake -G "Visual Studio 16 2019" -A "x64" ‘
-DCMAKE_INSTALL_PREFIX="inst" ‘ ‘
-S . -B build

cmake --build build --config Release
cmake --install build --config Release

cd inst/bin
./01_simple_frequency_sweep.exe
```

2.2 Typical program structure

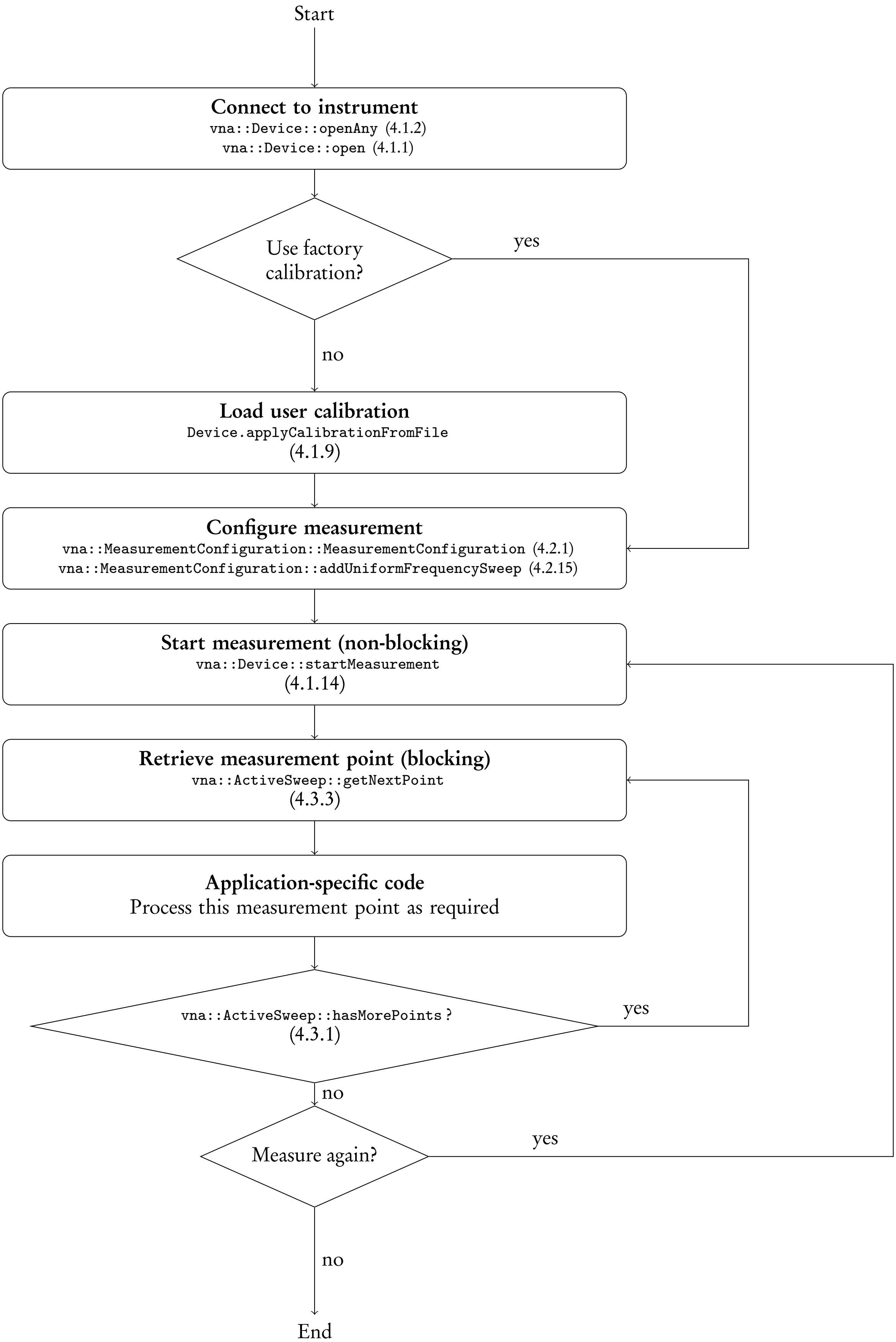
2.2.1 Synchronous mode

The diagram below shows the steps to carry out a measurement using synchronous mode (i.e. waiting for the entire measurement sweep to complete before processing measurements). It is simplified and incorporates the minimum number of steps to perform a measurement.



2.2.2 Asynchronous mode

The diagram below shows the steps to carry out a measurement using asynchronous mode (i.e. processing measurements point-by-point, rather than waiting for the whole measurement sweep to complete). It is simplified and incorporates the minimum number of steps to perform a measurement.


Start
Connect to instrument
vna::Device::openAny (4.1.2)
vna::Device::open (4.1.1)
Use factory calibration?
yes
no
Load user calibration
Device.applyCalibrationFromFile
(4.1.9)
Configure measurement
vna::MeasurementConfiguration::MeasurementConfiguration (4.2.1)
vna::MeasurementConfiguration::addUniformFrequencySweep (4.2.15)
Start measurement (non-blocking)
vna::Device::startMeasurement
(4.1.14)
Retrieve measurement point (blocking)
vna::ActiveSweep::getNextPoint
(4.3.3)
Application-specific code
Process this measurement point as required
vna::ActiveSweep::hasMorePoints ?
(4.3.1)
yes
no
Measure again?
yes
no
End

Chapter 3

API overview

3.1 Connecting to the instrument (or multiple instruments)

3.1.1 Connecting to any available instrument

If a only a single PicoVNA instrument is available, the simplest way to connect to it is to use the static `vna::Device::openAny` function (4.1.2). The function returns a `vna::Device` object that can then be used for instrument configuration and performing measurements.

This function may also be used if many instruments are available and required, since the device metadata can be queried after connecting, and the instruments can be distinguished by their serial numbers.

3.1.2 Connecting to a specific instrument

To connect to an instrument with a specific serial number, use the static `vna::Device::open` function (4.1.1). The serial number of the instrument to open may be specified as a parameter.

3.1.3 Using a simulated demonstration instrument (for evaluation or debugging)

To open a simulated demonstration instrument for evaluation or debugging, use the static `vna::Device::openDemo` function (4.1.3). The simulated DUT to use can be specified as a parameter; available options are a bandpass filter, a lowpass filter, an attenuator, or an antenna (S11 only).

Please note the calibration functions will not be available when using the simulated demonstration device.

3.1.4 Querying instrument metadata (e.g. serial number)

The intrinsic properties of a connected instrument can be inspected using the `Device.getInfo` function (4.1.5). This returns a `vna::DeviceInfo` object that includes metadata including the device serial, model and information about its capabilities.

3.2 Loading a user calibration

3.2.1 Exporting a user calibration from the PicoVNA 5 software

It is not currently possible to perform a calibration via the API. Instead, a calibration must be:

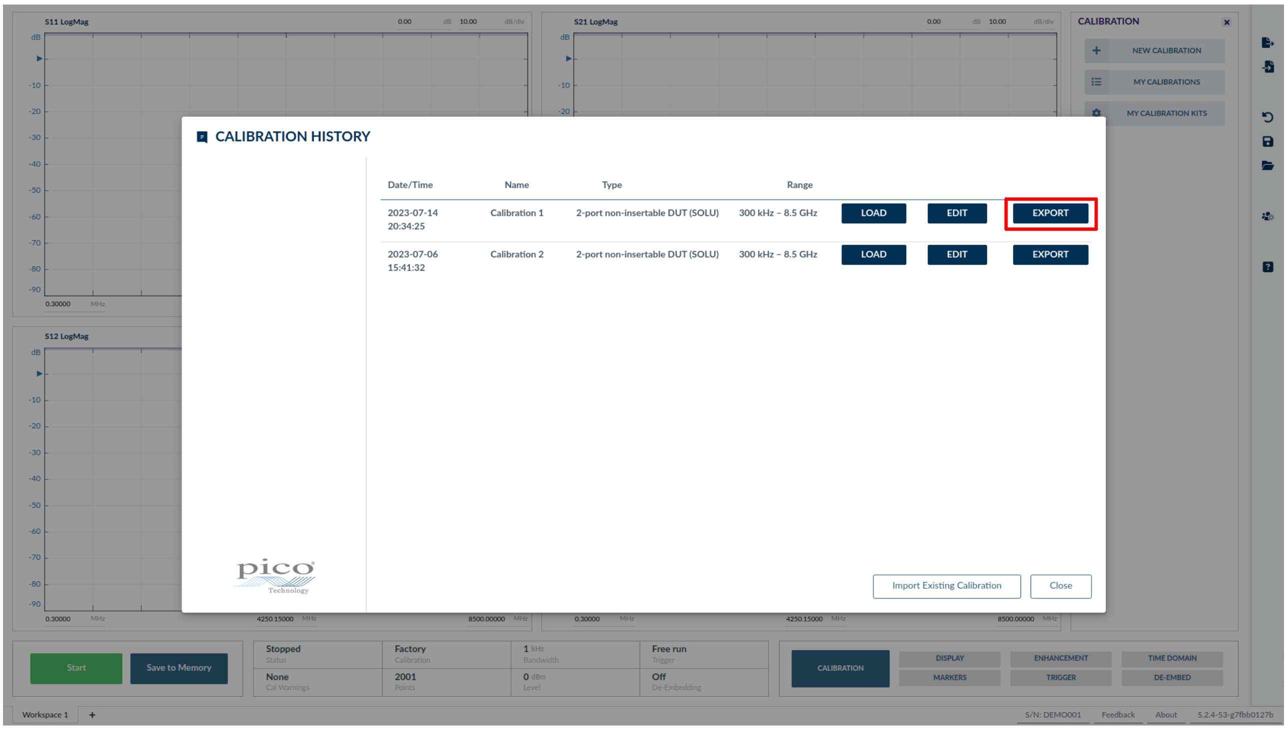


Figure 3.1: Exporting a calibration from the PicoVNA 5 software.

- (*in advance*) Performed within the PicoVNA 5 software
- (*in advance*) Exported to a file from the PicoVNA 5 software
- (*at user program runtime, every run*) Applied via the API

If no user calibration is loaded at runtime, the factory calibration will be used.

To export a user calibration from the PicoVNA 5 software, go to `Calibrations` → `My Calibrations` and press the "Export" button associated with the relevant calibration. You will then be given the opportunity to save your calibration to a file, which can be applied via the API. Figure 3.1 illustrates how the user calibration named `Calibration 1` would be exported.

3.2.2 Applying a user calibration at runtime via the API

Use the `Device.applyCalibrationFromFile` function (4.1.9) to load the user calibration file that was exported following the instructions in Section 3.2.1. This function can be passed an absolute or relative path to the calibration to be applied.

The user calibration to be loaded must have been performed from the PicoVNA 5 software (i.e. it must be a `.calx` file). It is not possible to load calibrations that were performed by the PicoVNA 3 software (from `.cal` files) via the API.

3.3 Setting up and performing a measurement

The parameters for a measurement must be specified before the measurement is carried out. Even when a user calibration has been applied, the parameters for the measurement must be set explicitly. If the measurement parameters set do not match those of the applied calibration, the calibration will be automatically interpolated without warning.

A measurement is configured using the `MeasurementConfiguration` object (4.2). To initiate the measurement once it has been configured, pass the `MeasurementConfiguration` object to the `Device::performMeasureme` function (4.1.13).

3.3.1 Setting up a measurement with uniform frequency step and constant power level

A convenient helper function `MeasurementConfiguration::addUniformFrequencySweep` (4.2.15) is available to set up a measurement with uniform frequency step and constant power level.

For example, to set up a 1001-point measurement, from 1 GHz to 2 GHz, with a power level of 0.0 dBm and 1000 Hz bandwidth:

```
vna::MeasurementConfiguration cfg;
cfg.addUniformFrequencySweep(
    1001,
    1e9,
    2e9,
    0.0,
    1000
);
```

3.3.2 Setting up a complex measurement with non-uniform frequency step or swept power level

A measurement can be set up point-by-point, so that each measurement point can use a different frequency and/or power level. All points in the sweep must be measured using the same bandwidth.

For example, to set up a 3-point measurement, measuring a frequency of 1 GHz at three different power levels (the syntax used in this example assumes C++ 20 is available; if not, the syntax will need to be adapted):

```
vna::MeasurementConfiguration cfg;
cfg.addPoint({.powerLeveldBm = -20.0, .frequencyHz = 1E9, .bandwidthHz = 1000});
cfg.addPoint({.powerLeveldBm = 0.0, .frequencyHz = 1E9, .bandwidthHz = 1000});
cfg.addPoint({.powerLeveldBm = 6.0, .frequencyHz = 1E9, .bandwidthHz = 1000});
```

3.4 Retrieving the data resulting from a measurement

The `vna::Device::performMeasurement` function returns a `ActiveMeasurement` object (4.3) that can be queried to:

- Check the status of the measurement. Has it completed yet?
- Retrieve the resulting data from the measurement.

Two modes are available for retrieving data from the `ActiveMeasurement`: synchronous mode and asynchronous mode. In synchronous mode, the function used to retrieve data from the device will block until all the data from all measurement points are available. In asynchronous mode, measurements can be retrieved point-by-point, and processed in parallel with the device collecting more measurement points. In asynchronous mode, the function used to retrieve data from the device will only block until the next measurement point is available.

3.4.1 Collecting measurements synchronously

Use the `vna::ActiveMeasurement::getAllPoints` function (4.3.5) to retrieve all the measurements in one `vector`. The function will block until all the measurements are available.

3.4.2 Collecting measurements asynchronously

To retrieve the data from measurement points one-by-one, so that data can be processed as soon as it is available without waiting for the sweep to complete, use the `vna::ActiveMeasurement::getNextPoint` function. This function will block until the data from the next point in the measurement sweep is available.

A non-blocking function, `vna::ActiveMeasurement::tryGetNextPoint` (4.3.4), is also available. This will return data if and only if it is not available without blocking the thread.

3.4.3 Aborting an in-progress measurement

Use the `vna::ActiveMeasurement::abort` function (4.3.6) to cancel a measurement that is currently in progress.

Chapter 4

API reference

4.1 Class: vna::Device

File:
include/vna/cxx/Device.h

Description:
The Device represents a connection to a VNA attached to this computer.

4.1.1 open

Declaration:
```
static vna::open(const std::string& serial) VNA_THROWS(
        vna::CommsError,
        vna::DeviceOpenFailedException,
        vna::DeviceNotRecognisedException,
        vna::DeviceNotFoundException,
        vna::UpgradeRequiredException
      );
```
Description:
Open the instrument with the given serial number.

Opening the same device twice is not allowed. An instrument may be closed by destructing the object that was created when it was opened (there is no explicit function for closing an instrument).

For testing and product evaluation, serial numbers of the form DEMOXXXX (where XXXX is some integer) may be used to request a simulated demonstration device. This provides a convenient way to test your code where no real device is available.

Parameters:

serial	The serial of the instrument to open.

Returns:
An object that can be used to control the instrument that has been opened.

4.1.2 openAny

Declaration:
```
static vna::Device openAny() VNA_THROWS(
      vna::CommsError,
      vna::DeviceOpenFailedException,
      vna::DeviceNotRecognisedException,
      vna::DeviceNotFoundException,
      vna::UpgradeRequiredException
);
```

Description:
Open any real instrument, but not the demo device.

This is a convenient function to open the instrument where only one instrument is expected to be connected to the computer, and your program only requires one instrument.

Since opening the same device twice is not allowed, it is not possible to use this function repeatedly to open multiple devices. To achieve that, you must get a list of available devices, and then open them by serial number.

Parameters:
This function takes no parameters.

Returns:
An object that can be used to control the instrument that has been opened.

4.1.3 openDemo

Declaration:
```
static vna::Device openDemo(DemoDeviceDut dut = DemoDeviceDut::BAND_PASS_FILTER);
```
Description:
Open the simulated demo device.

This is equivalent to calling `open("DEMO000")`, except this function also allows the simulated DUT to be selected from a set of options.

Parameters:
`dut` The device under test to demo.

Returns:
An object that can be used to control the simulated instrument that has been opened.

4.1.4 Destructor

Declaration:
```
~Device();
```
Description:
Destructor for the `Device`. This closes the connection to the instrument and cleans up associated resources.

Parameters:
This function takes no parameters.

4.1.5 getInfo

Declaration:
```
vna::DeviceInfo getInfo() const;
```
Description:
Query properties of the connected instrument.

The returned struct describes immutable, intrinsic properties of the attached instrument, such as model number and various hardware limits.

Parameters:
This function takes no parameters.

4.1.6 isIdle

Declaration:

```
bool isIdle() const VNA_THROWS(vna::CommsError);
```

Description:
Returns true iff the device is not currently collecting measurements.

Parameters:
This function takes no parameters.

4.1.7 getTemperature (PicoVNA 108 only)

Declaration:
```
double getTemperature() const VNA_THROWS(vna::OperationNotSupportedException);
```

Description:
Returns the instrument's internal temperature in degrees centigrade.

This function is only available on the PicoVNA 108 instrument; on other instruments a `vna::OperationNotSupported` will be thrown. If the instrument is still initialising, NaN will be returned. In this case, call this function again later to obtain a temperature reading.

Parameters:
This function takes no parameters.

4.1.8 loadFactoryCalibration

Declaration:
```
void loadFactoryCalibration() VNA_THROWS(vna::CommsError);
```

Description:
Apply the factory calibration.

Parameters:
This function takes no parameters.

4.1.9 applyCalibrationFromFile

Declaration:
```
void applyCalibrationFromFile(const std::string& path) VNA_THROWS(
        vna::CommsError,
        vna::ParseError
);
```

Description:
Apply the calibration stored in the given file path.

The user calibration to be loaded must have been performed from the PicoVNA 5 software (i.e. it must be a `.calx` file). It is not possible to load calibrations that were performed by the PicoVNA 3 software (from `.cal` files) via the API.

To export a calibration from the PicoVNA 5 software to a file, so this command can be used to load it, use `Export` feature in the Calibration History modal, accessed via the `Calibrations` -> `My Calibrations` menu.

Parameters:

`path`	Absolute or relative path to the .calx file storing the calibration to apply.

4.1.10 getMetadataForCurrentCalibration

Declaration:
```
const vna::CalibrationMetadata& getMetadataForCurrentCalibration();
```


Description:
Returns metadata for the calibration that is currently applied.

Parameters:
This function takes no parameters.

4.1.11 pulseTriggerOutput

Declaration:
```
void pulseTriggerOutput() VNA_THROWS(
        vna::CommsError,
        vna::OperationNotSupportedException
);
```
Description:
Send a pulse through the VNA's trigger output port.

Parameters:
This function takes no parameters.

4.1.12 trigger

Declaration:
```
void trigger() VNA_THROWS(
    vna::CommsError,
    vna::OperationNotSupportedException
);
```
Description:
If any sweep plans have been started with `startTriggeredSweep` in `MANUAL` mode, this function will activate them. Otherwise, this function has no effect.

Parameters:
This function takes no parameters.

4.1.13 performMeasurement

Declaration:
```
std::vector<vna::SParameterMeasurementPoint> performMeasurement(const vna::MeasurementConfiguration& sweep) VNA_THR
        vna::CommsError
);
```
Description:
Collect all the measurement points described by the given `vna::MeasurementConfiguration`, and return them in an `std::vector`.

This function blocks until the device has collected all of the measurements, which could take any time from several microseconds to several minutes depending on the `vna::MeasurementConfiguration`.

If you wish to consume the data points incrementally as they become available (rather than blocking your application), use `startMeasurement()` to perform an asynchronous sweep.

4.1.14 startMeasurement

Declaration:
```
vna::ActiveMeasurement startMeasurement(const vna::MeasurementConfiguration& sweep) VNA_THROWS(
        vna::CommsError
);
```

Description:
Asynchronously start the sweep described by the given sweep plan. The sweep is handed to the sweep scheduler, and its data will become available soon. Use the functions of the `ActiveMeasurement` class to consume the data points incrementally as they become available.

Returns:
An ActiveMeasurement object representing the started sweep. The data for this sweep may be extracted from this object.

4.2 Class: vna::MeasurementConfiguration

File:
`include/vna/cxx/MeasurementConfiguration.h`

Description:
Describes a sweep to be performed by the device.

4.2.1 MeasurementConfiguration (constructor)

Declaration:
```
MeasurementConfiguration();
```
Description:
Create an empty MeasurementConfiguration.

This is useful if you plan to use `addPoint()` to add a custom set of measurement points to the sweep (as opposed to using `MeasurementConfiguration::frequency()` and friends to generate a uniform sweep).

4.2.2 getTriggerMode

Declaration:
```
vna::TriggerMode getTriggerMode() const;
```
Description:
When configuring a triggered sweep, what trigger event will cause the sweep to start?

Parameters:
This function takes no parameters.

4.2.3 setTriggerMode

Declaration:
```
void setTriggerMode(vna::TriggerMode triggerMode);
```
Description:
For a triggered sweep, configure the trigger event will cause the sweep to start.

Parameters:

`triggerMode`	The trigger event that will cause the sweep to start (if a triggered sweep is enabled).

4.2.4 getTriggerAction

Declaration:
```
vna::TriggerAction getTriggerAction() const;
```


Description:
In a triggered sweep, what action will be taken when a trigger event occurs?

Parameters:
This function takes no parameters.

4.2.5 setTriggerAction (PicoVNA 108 only)

CAUTION!	This command is experimental. Using it may yield unexpected or incorrect results. The command may be changed or removed in future versions of the API.

Declaration:
```
void setTriggerAction(vna::TriggerAction triggerAction);
```
Description:
In a triggered sweep, configure the action will be taken when a trigger event occurs.

Parameters:

`triggerAction`	The action that will be taken when a trigger even occurs (if a triggered sweep is enabled).

4.2.6 clear

Declaration:
```
void clear();
```
Description:
Delete all measurement points.

Parameters:
This function takes no parameters.

4.2.7 addPoint

Declaration:
```
void addPoint(const vna::MeasurementPoint& p);
```
Description:
Add a point to the sweep. A sweep may be composed of an arbitrary set of points.

It is not necessary for the frequency of each measurement point to exactly match the frequency of error terms in the loaded calibration. Adding points that do not exactly match the frequency of error terms in the loaded calibration will cause a new set of (interpolated) error terms to be generated so that a new calibration need not be carried out. Note that in order to obtain the best instrument capability a new calibration should be performed whenever the sweep parameters change.

Point types (i.e: MeasurementPoint or CWMeasurementPoint) may not be mixed.
At present, all point point types must have the same bandwidth.

Parameters:

`p`	The measurement point to add to the sweep.

4.2.8 setAveraging

Declaration:
```
void setAveraging(unsigned int n) VNA_THROWS(vna::InvalidParameterException);
```
Description:
If n>1, the device will be instructed to perform several sweeps and each measurement point returned by `performMeasurement` or by querying the `ActiveMeasurement` will be the average of the several measurements performed by the device. This averaging process will yield identical results to requesting

several sweeps from the device (with averaging switched off) and computing the mean value for each measurement point in your code.

Set `n=1` to turn off averaging.

Parameters:

`n`	The number of samples to average over. Must not be zero.

4.2.9 setPortOffset

Declaration:

```
void setPortOffset(double port1Length, double port2Length,
                   double dielectricConstant = 1.0) VNA_THROWS(vna::InvalidParameterException);
```

Description:

Apply reference plane offsetting in place.

This is useful to correct for the connection of a device under test via a pair of cables of a known length. This happens before other de-embedding.

Parameters:

`port1Length`	The length, in m, of the cable between port 1 and the device under test. Must be at least 0.
`port2Length`	The length, in m, of the cable between port 2 and the device under test. Must be at least 0.
`dielectricConstant`	The dielectric constant of the medium. This must be at least 1.0.

4.2.10 setDeEmbedPortNetworks

4.2.10.1 setDeEmbedPortNetworks overload 1

Declaration:

```
void setDeEmbedPortNetworks(int port,
                            const std::vector<std::string>& networks) VNA_THROWS(vna::InvalidParameterException);
```

Description:

Set networks to de-embed from the measurement.

Parameters:

`port`	Port to put the network on.
`networks`	S2P files of networks on the port, starting with the network closest to the port and ending with the network closest to the DUT.

4.2.10.2 setDeEmbedPortNetworks overload 2 (C++ 20 or greater)

Declaration:

```
void setDeEmbedPortNetworks(int port,
                            std::span<const std::string> networks) VNA_THROWS(vna::InvalidParameterException);
```

Description:

Set networks to de-embed from the measurement.

Parameters:

`port`	Port to put the network on.
`networks`	S2P files of networks on the port, starting with the network closest to the port and ending with the network closest to the DUT.

4.2.11 getPoint

4.2.11.1 getPoint overload 1

Declaration:
`vna::MeasurementPoint& getPoint(int i);`
Description:
Get the `i`th point from the sweep.

Parameters:
`i` Index of the point to get from the sweep (starting from `0`).

4.2.11.2 getPoint overload 2

Declaration:
`const vna::MeasurementPoint& getPoint(int i) const;`
Description:
Get the `i`th point from the sweep.

Parameters:
`i` Index of the point to get from the sweep (starting from `0`).

4.2.12 getPoints

4.2.12.1 getPoints overload 1

Declaration:
`std::vector<vna::MeasurementPoint>& getPoints();`
Description:
Get the list of all points.

Parameters:
This function takes no parameters.

4.2.12.2 getPoints overload 2

Declaration:
`const std::vector<vna::MeasurementPoint>& getPoints() const;`
Description:
Get the list of all points.

Parameters:
This function takes no parameters.

4.2.13 getMeasurementFrequencies

Declaration:
`std::vector<double> getMeasurementFrequencies() const;`
Description:
Extract the measurement frequencies from all points and return as a list of doubles. This is a handy shortcut for the case where you want to key something by frequency.

Parameters:
This function takes no parameters.

4.2.14 numPoints

Declaration:

```
int numPoints() const;
```

Description:
Returns the number of points in the configuration.

Parameters:
This function takes no parameters.

4.2.15 addUniformFrequencySweep

Declaration:

```
void addUniformFrequencySweep(
        int numPoints,
        double startFreqHz,
        double stopFreqHz,
        double powerLeveldBm,
        double bandwidthHz
    );
```

Description:
Convenience function for generating sweeps in which all but the frequency parameter is fixed, and the frequency parameter is moved through a range with equal intervals.

Parameters:

`numPoints`	Number of points in the sweep.
`startFreqHz`	The first frequency to measure.
`stopFreqHz`	The last frequency to measure.
`powerLeveldBm`	The power level to use.
`bandwidthHz`	The bandwidth to use.

4.2.16 addUniformPowerSweep

Declaration:

```
void addUniformPowerSweep(
        int numPoints,
        double startPowerLeveldBm,
        double stopPowerLeveldBm,
        double frequencyHz,
        double bandwidthHz
    );
```

Description:
Convenience function for generating sweeps in which all but the power parameter is fixed, and the power parameter is moved through a range with equal intervals.

Parameters:

`numPoints`	Number of points in the sweep.
`startPowerLeveldBm`	The first power to measure.
`stopPowerLeveldBm`	The last power to measure.
`frequencyHz`	The frequency to use.
`bandwidthHz`	The bandwidth to use.

4.3 Class: vna::ActiveMeasurement

File:
`include/vna/cxx/ActiveMeasurement.h`

Description:
Represents the ongoing execution of a single `MeasurementConfiguration` on the device.

4.3.1 hasMorePoints

Declaration:
```
bool hasMorePoints() const;
```
Description:
Determine whether there are more points available to read from the object using `getNextPoint()`.

Not to be confused with `isFinished()`, which merely determines whether the measurement process has finished. If it has, the device is ready to do something else (and you may access all the data from the completed measurement).

Parameters:
This function takes no parameters.

4.3.2 getConfig

Declaration:
```
const vna::MeasurementConfiguration& getConfig() const;
```
Description:
Get the `MeasurementConfiguration` object originally used to define this sweep.

Parameters:
This function takes no parameters.

4.3.3 getNextPoint

Declaration:
```
vna::SParameterMeasurementPoint getNextPoint() VNA_THROWS(vna::CommsError);
```
Description:
Read the next data point from this sweep. Blocks until the data is available. This allows you to digest the data as it becomes available, or to manage your own memory.

Parameters:
This function takes no parameters.

4.3.4 tryGetNextPoint

4.3.4.1 tryGetNextPoint overload 1

Declaration:
```
bool tryGetNextPoint(vna::SParameterMeasurementPoint& output) VNA_THROWS(vna::CommsError);
```
Description:
If the next measurement point is already available, write it to `output`.

Parameters:

`output`	The measurement point is written here if it is available.

Returns:
`true` if and only if a point was written out.

4.3.4.2 tryGetNextPoint overload 2 (C++ 20 or greater)

Declaration:
```
std::optional<vna::SParameterMeasurementPoint> tryGetNextPoint() VNA_THROWS(vna::CommsError);
```

Description:
If the next measurement point is already available, return it.

Parameters:
This function takes no parameters.

4.3.5 getAllPoints

Declaration:
```
std::vector<vna::SParameterMeasurementPoint> getAllPoints() VNA_THROWS(vna::CommsError);
```
Description:
Wait for the sweep to finish, and return all remaining data at once. Equivalent to calling `getNextPoint()` in a loop.

Parameters:
This function takes no parameters.

4.3.6 abort

Declaration:
```
void abort() VNA_THROWS(vna::CommsError);
```
Description:
Stop the sweep early.

After stopping the sweep, you can still use `tryGetNextPoint()` to extract any remaining data from its buffer. This function blocks until the sweep has stopped and the device is ready to do other things.

Parameters:
This function takes no parameters.

4.4 Function Group: Time domain

File:
`include/vna/cxx/TimeDomain.h`

Description:
Perform time domain transforms.

4.4.1 struct TimeDomainOptions

Declaration:
```
struct TimeDomainOptions final
{
    double outputStartSeconds = -std::numeric_limits<double>::infinity();
    double outputEndSeconds = std::numeric_limits<double>::infinity();
    double minOutputSampleRate = 0.0;
    int outputPaddingSamplesStart = 0;
    int outputPaddingSamplesEnd = 0;

    vna::TimeDomainResponse response = vna::TimeDomainResponse::STEP;
    vna::TimeDomainMode mode = vna::TimeDomainMode::LOW_PASS;
    vna::TimeDomainWindowFunction window = vna::TimeDomainWindowFunction::RECTANGULAR;
    vna::TimeDomainDCTermination dcTermination = vna::TimeDomainDCTermination::NONE;

    /**
     * If resistive DC termination is in use, this is the resistance value.
     */
```

```
    double dcTerminationResistanceOhms = 50;

    /**
     * If a Kaiser Bessel window function is in use, this value defines its window order.
     *
     * Otherwise, this field does nothing.
     */
    int kbWindowOrder = 0;
};
```

Description:
Used for specifying configuration options when performing time domain transforms.

4.4.2 struct TimeDomainSample

Declaration:

```
struct TimeDomainSample final
{
    double time;
    double sample;

    TimeDomainSample():
    TimeDomainSample(std::make_pair<double,double>(0.0,0.0))
    {}

    TimeDomainSample(std::pair<double, double> p):
        time(p.first), sample(p.second)
    {}

    operator std::pair<double, double>()
    {
        return {time, sample};
    }
};
```

4.4.3 transform

4.4.3.1 transform overload 1

Declaration:

```
std::vector<TimeDomainSample> transform(
    const TimeDomainOptions& opts,
    vna::MeasurementParameter whichMp,
    const SParameterMeasurementPoint* mp,
    int numPoints
) VNA_THROWS(vna::InvalidParameterException);
```

Description:
Perform the time domain transform on the given list of points, assuming these points form the entire sweep.

4.4.3.2 transform overload 2

Declaration:

```
std::vector<TimeDomainSample> transform(
    const TimeDomainOptions& opts,
    vna::MeasurementParameter whichMp,
    const std::vector<SParameterMeasurementPoint>& mps
) VNA_THROWS(vna::InvalidParameterException);
```

Description:
Perform the time domain transform on the given list of points, assuming these points form the entire sweep.

4.4.3.3 transform overload 3 (C++ 20 or greater)

Declaration:
```
std::vector<TimeDomainSample> transform(
    const TimeDomainOptions& opts,
    vna::MeasurementParameter whichMp,
    std::span<const SParameterMeasurementPoint> mps
) VNA_THROWS(vna::InvalidParameterException);
```
Description:
Perform the time domain transform on the given list of points, assuming these points form the entire sweep.

4.4.3.4 transform overload 4

Declaration:
```
std::vector<TimeDomainSample> transform(
    const TimeDomainOptions& opts,
    vna::MeasurementParameter whichMp,
    const SParameterMeasurementPoint* mp,
    int numPoints,
    const MeasurementConfiguration& mc
) VNA_THROWS(vna::InvalidParameterException);
```
Description:
Perform the time domain transform on the given list of points, assuming these points are the result of partially completing the sweep configuration described by `mc`.

4.4.3.5 transform overload 5

Declaration:
```
std::vector<TimeDomainSample> transform(
    const TimeDomainOptions& opts,
    vna::MeasurementParameter whichMp,
    const std::vector<SParameterMeasurementPoint>& mps,
    const MeasurementConfiguration& mc
) VNA_THROWS(vna::InvalidParameterException);
```
Description:
Perform the time domain transform on the given list of points, assuming these points are the result of partially completing the sweep configuration described by `mc`.

4.4.3.6 transform overload 6 (C++ 20 or greater)

Declaration:
```
std::vector<TimeDomainSample> transform(
    const TimeDomainOptions& opts,
    vna::MeasurementParameter whichMp,
    const std::span<const SParameterMeasurementPoint> mps,
    const MeasurementConfiguration& mc
) VNA_THROWS(vna::InvalidParameterException);
```
Description:
Perform the time domain transform on the given list of points, assuming these points are the result of partially completing the sweep configuration described by `mc`.

4.4.3.7 transform overload 7

Declaration:
```
std::vector<TimeDomainSample> transform(
    const TimeDomainOptions& opts,
    const std::complex<double>* mp,
    int numPoints,
    double startFreq,
    double stopFreq
) VNA_THROWS(vna::InvalidParameterException);
```
Description:
Perform the time domain transform on the given list of points, assuming these points form the entire sweep. These versions operate on complex numbers rather than directly on the SParameterMeasurementPoint objects obtained from the measurement APIs.

4.4.3.8 transform overload 8

Declaration:
```
std::vector<TimeDomainSample> transform(
    const TimeDomainOptions& opts,
    const std::vector<std::complex<double>>& mps,
    double startFreq,
    double stopFreq
) VNA_THROWS(vna::InvalidParameterException);
```
Description:
Perform the time domain transform on the given list of points, assuming these points form the entire sweep. These versions operate on complex numbers rather than directly on the SParameterMeasurementPoint objects obtained from the measurement APIs.

4.4.3.9 transform overload 9 (C++ 20 or greater)

Declaration:
```
std::vector<TimeDomainSample> transform(
    const TimeDomainOptions& opts,
    const std::span<const std::complex<double>> mps,
    double startFreq,
    double stopFreq
) VNA_THROWS(vna::InvalidParameterException);
```
Description:
Perform the time domain transform on the given list of points, assuming these points form the entire sweep. These versions operate on complex numbers rather than directly on the SParameterMeasurementPoint objects obtained from the measurement APIs.

4.4.3.10 transform overload 10

Declaration:
```
std::vector<TimeDomainSample> transform(
    const TimeDomainOptions& opts,
    const std::complex<double>* mp,
    int numPoints,
    const MeasurementConfiguration& mc
) VNA_THROWS(vna::InvalidParameterException);
```
Description:
Perform the time domain transform on the given list of points, assuming these points are the result of partially completing the sweep configuration described by `mc`. These versions operate on complex numbers rather than directly on the `SParameterMeasurementPoint` objects obtained from the measurement APIs.

4.4.3.11 transform overload 11

Declaration:
```
std::vector<TimeDomainSample> transform(
    const TimeDomainOptions& opts,
    const std::vector<std::complex<double>>& mps,
    const MeasurementConfiguration& mc
) VNA_THROWS(vna::InvalidParameterException);
```
Description:
Perform the time domain transform on the given list of points, assuming these points are the result of partially completing the sweep configuration described by `mc`. These versions operate on complex numbers rather than directly on the `SParameterMeasurementPoint` objects obtained from the measurement APIs.

4.4.3.12 transform overload 12 (C++ 20 or greater)

Declaration:
```
std::vector<TimeDomainSample> transform(
    const TimeDomainOptions& opts,
    const std::span<const std::complex<double>> mps,
    const MeasurementConfiguration& mc
) VNA_THROWS(vna::InvalidParameterException);
```
Description:
Perform the time domain transform on the given list of points, assuming these points are the result of partially completing the sweep configuration described by `mc`. These versions operate on complex numbers rather than directly on the `SParameterMeasurementPoint` objects obtained from the measurement APIs.

4.5 Function group: Enhancement

File:
`include/vna/cxx/Enhancement.h`

Description:
Facilities for enhancing the data after extracting it from the device.

4.5.1 applySmoothing

4.5.1.1 applySmoothing overload 1

Declaration:
```
void applySmoothing(std::complex<double>* dst,
                    const std::complex<double>* src,
                    uint32_t count,
                    double amount)
VNA_THROWS(vna::Exception);
```
Description:
Apply a smoothing algorithm to an array of complex numbers. This is intended to smooth S-parameters.

Parameters:

`dst`	The output.
`src`	The input.
`count`	The size of the input and output.
`amount`	The amount of smoothing. This is a number between 0.0 (least smoothing) and 1.0 (most smoothing). Note that this is different from the UI, which ranges from 0.0 to 10.0.

4.5.1.2 applySmoothing overload 2 (C++ 20 or greater)

Declaration:
```
void applySmoothing(std::span<std::complex<double>> buffer, double amount)
VNA_THROWS(vna::Exception);
```
Description:
Apply smoothing in-place.

Parameters:

`buffer`	The input, which is overwritten with the result.
`amount`	The amount of smoothing. This is a number between 0.0 (least smoothing) and 1.0 (most smoothing). Note that this is different from the UI, which ranges from 0.0 to 10.0.

4.5.1.3 applySmoothing overload 3

Declaration:
```
void applySmoothing(std::vector<std::complex<double>>& buffer, double amount)
VNA_THROWS(vna::Exception);
```
Description:
Apply smoothing in-place.

Parameters:

`buffer`	The input, which is overwritten with the result.
`amount`	The amount of smoothing. This is a number between 0.0 (least smoothing) and 1.0 (most smoothing). Note that this is different from the UI, which ranges from 0.0 to 10.0.

4.6 Namespace: vna

4.6.1 getConnectedDeviceInfo

Declared in:
`include/vna/cxx/Device.h`

Declaration:
```
std::vector<vna::DeviceInfo> getConnectedDeviceInfo()
```
Description:
Get device information for all connected VNAs.

This function can be used to choose between your VNAs based on their hardware properties, model number, etc.

Since this function queries the hardware info of each device, it is much slower than `getConnectedDeviceSerials()`. This may be an issue if you have a large number of VNAs attached. To handle this problem:

- Store a `deviceSerial` → `info` mapping somewhere in your application so you don't need to run this function more than once ever, and can simply `vna::open(theCorrectSerial)` when needed.
- If you don't care about device properties and just want to load-balance (eg. if all your VNAs are identical), use `getConnectedDeviceSerials()` to list them and pass those IDs to `open()` according to your load-balance logic.

Returns:
Connected devices are returned ordered lexicographically by serial number.

