Mikko Peurala

Automated RF Line Testing

Test setup to execute RF measurements using microwave switches with Python

Metropolia University of Applied Sciences

Bachelor of Engineering

Electrical and Automation Engineering

Bachelor’s Thesis

19.02.2022

Abstract

Author(s):	Mikko Peurala
Title:	Automated RF Line Testing
Number of Pages:	32 pages + 4 appendices
Date:	19th of February 2022
Degree:	Bachelor of Engineering
Degree Programme:	Electrical and Automation Engineering
Professional Major:	Automation Technology
Instructor(s):	Erkki Räsänen, Principal Lecturer
	Russell Lake, Director Quantum Applications

This thesis describes the automation of RF line measurements and processing of measurement data using Python. The thesis includes the requirements of the devices, the design and programming of the software, and the structure of the Python code.

The purpose of the thesis work was to speed up and standardize the measurement process. The goal was to create software that would be able to perform measurements, store data, and process it in a single interface. Making RF measurements took a lot of time, and the aim was to reach only a fraction of the time spent.

The test setup contains a vector network analyzer and two switch matrices which are controlled by Python code. The device being tested contains 24 high-frequency transmission lines that are connected to switch matrices, which in turn are connected to the vector network analyzer. Python code performs measurements by activating ports of the switch matrices one at a time and stores the measurement data in a user-defined file location. After measurements, the data can be processed in the same Python interface.

First, all measuring devices were connected to a private network to make their operation possible using Python. Subsequently, the structure of the software was planned and implemented using Python's QCodes module. QCodes utilizes software drivers to control the measurement instruments. Therefore, programming the drivers was essential to the results of the thesis work. The structure of the software was updated and developed as the work progressed according to the objectives set by Bluefors Oy.

The result of the thesis work is a testing station that simplifies and speeds up the test process.

Keywords: Python, RF, VNA, Switch, QCoDeS

Tiivistelmä

Tekijä:	Mikko Peurala
Otsikko:	Automatisoitu RF-linjojen Testaus
Sivumäärä:	32 sivua + 4 liitettä
Aika:	19.02.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Sähkö- ja automaatiotekniikka
Ammatillinen pääaine:	Automaatiotekniikka
Ohjaajat:	Yliopettaja Erkki Räsänen Director Quantum Applications Russell Lake

Tämä opinnäytetyö kuvaa RF-linjojen mittausten suorittamisen ja mittausdatan automatisointia Pythonin avulla Bluefors Oy:lle. Työssä käydään läpi laitteiden vaatimukset, ohjelmiston suunnittelu ja ohjelmointi sekä Python-koodin rakenne.

Opinnäytetyön tarkoituksena oli nopeuttaa ja standardisoida prosessia. Tavoitteena oli luoda ohjelmisto, joka pystyisi suorittamaan mittaukset, tallentamaan datan ja käsittelemään sen yhdessä käyttöliittymässä. RF-mittausten tekeminen vei paljon aikaa ja tavoitteena oli päästä vain murto-osaan käytetystä ajasta.

Testiasema sisältää vektoripiirianalysaattorin sekä kaksi kytkinmatriisia, joita ohjataan Python-koodin avulla. Testattava laite sisältää 24 korkeataajuuksista siirtolinjaa, jotka kytketään kytkinmatriiseihin, jotka puolestaan ovat kytkettynä vektoripiirianalysaattoriin. Python-koodi suorittaa mittaukset aktivoimalla kytkinmatriisien portteja yksi kerrallaan ja tallentaa mittausdatan käyttäjän määrittelemään tiedostosijaintiin. Mittausten jälkeen data voidaan käsitellä samassa Pythonin käyttöliittymässä.

Aluksi kaikki mittalaitteet kytkettiin yksityiseen verkkoon, jotta niiden operointi olisi mahdollista Pythonin avulla. Tämän jälkeen ohjelmiston rakennetta alettiin suunnitella ja toteuttaa Pythonin QCoDeS-moduulin avulla. QCoDeS tarvitsee ajurit toimiakseen, joten niiden ohjelmointi oli oleellista työn tuloksen kannalta. Ohjelmiston rakennetta päivitettiin ja kehitettiin työn edetessä Bluefors Oy:n asettamien tavoitteiden mukaisesti.

Opinnäytetyön tuloksena syntyi testausasema, jolla testausprosessi yksinkertaistui ja nopeutui huomattavasti.

Avainsanat: Python, Radiotaajuus, VNA, Kytkin, QCoDeS

Contents

List of Abbreviations

DUT:	Device Under Test
dB:	Decibel
dBm:	Decibel-Milliwatt
HDW:	High-Density Wiring
IDE:	Integrated Development Environment
Hz:	Hertz
IF:	Intermediate Frequency
RF:	Radio Frequency
SCPI:	The Standard Commands for Programmable Instruments
SMA:	Sub Miniature version A
SMPM:	Sub Miniature Push-On Micro
TDR:	Time-Domain Reflection
VNA:	Vector Network Analyzer
VSWR:	Voltage Standing Wave Ratio

1 Introduction

Bluefors Oy manufactures dilution refrigerator measurement systems for different research fields in science and technology. The most rapidly growing field is quantum computing which requires experimental wiring inside the cryostat, such as high frequency coaxial lines for drive and readout lines of the qubits. Bluefors offers different wiring options that are tested to operate reliably in cryogenic temperatures. In this thesis work, the focus was on Radio Frequency (RF) lines and High-Density Wiring (HDW). Picture of a HDW with 72 lines can be seen in figure 1.

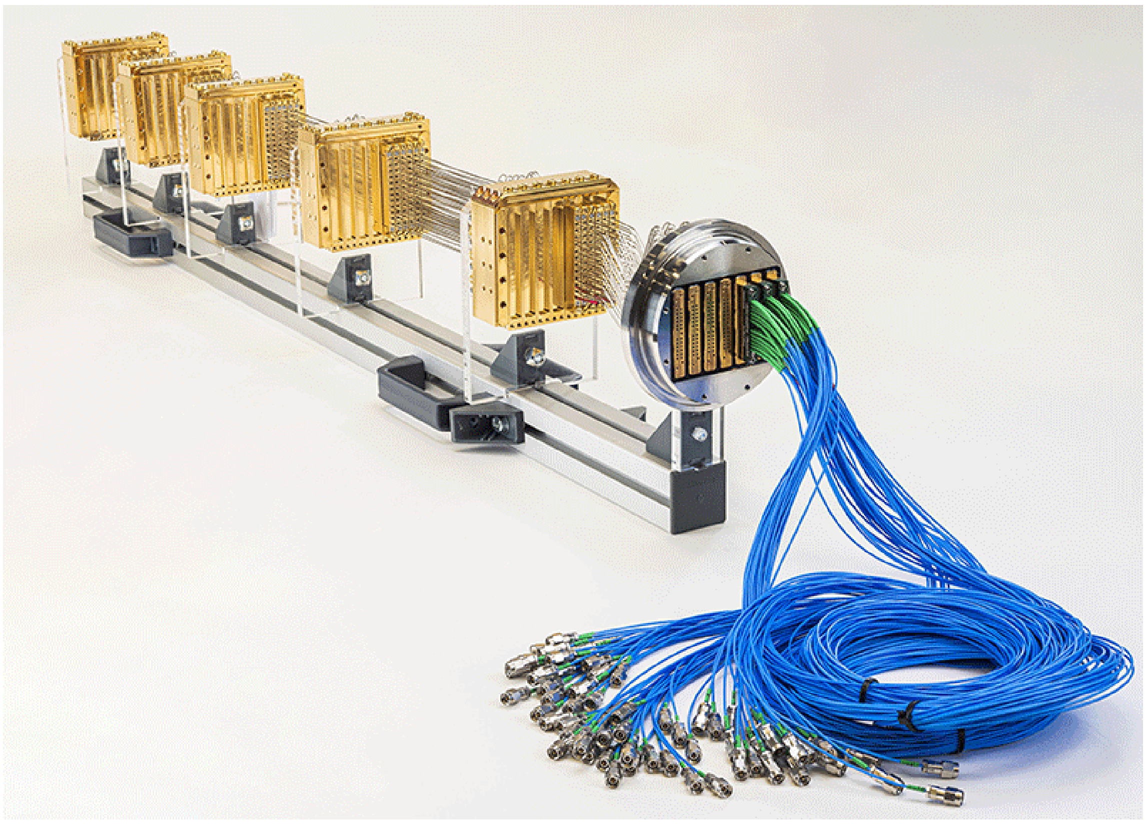


Figure 1. Bluefors High-Density Wiring [1].

HDW is an experimental wiring option for a modular XLD side-loader (XLDsl) dilution refrigerator measurement system [2]. An XLDsl can be equipped with a maximum of 1008 high frequency RF lines divided into 6 ports. High frequency RF lines are used to carry input and output signals to the device under test (DUT). The need for such many lines comes from the evolution of quantum chips. Two years ago, people were talking about 25 qubits but now the leaders of the quantum race have more than 100. And the more qubits there are, the more RF lines are needed.

All RF lines are individually tested after installation, and the possibility to have such many lines in one system increases workload greatly. The purpose of this thesis work was to design and build an automated test setup to test multiple lines with a single attachment. The main goal was to make the testing process faster, easier, and more reliable. Having consistent data would also help data analysis when developing and improving RF products in Bluefors.

2 Specification of Requirements

Testing HDW had two main problems: it was taking too much time and process flow was illogical. All lines were tested one by one with VNA, and the test data was saved to a USB stick. Testing lines one by one caused problems to test cables because they could not handle so many mating cycles and they had to be replaced frequently. Test data needed to be verified by plotting it on a computer with Python script and this was the reason why data was saved to a USB stick. If data indicated that something was broken, the technician needed to use a different Python script to perform time-domain reflection (TDR) analysis. Flowchart of the old style can be seen in appendix 1.

Requirements for this project were specified in the project plan for Bluefors as following:

- Ability to test 24 lines with a single attachment at room temperature to make testing faster.
- Easier and more logical workflow for a technician.
- Python code for executing the measurements.

HDW is divided into modules of 24 RF lines, so it would be convenient to test the full module at once. This was easily achieved since the HDW design has a multi-connector feature that allows you to attach multiple lines to the end interfaces of HDW.

One of the main goals was to reduce the steps technicians needed to make. Saving and naming files could be automated, and test data could be saved directly into a computer where data could be analyzed. All this should be operated in one user interface which makes the testing easier for the technician. A flowchart of the new style can be seen in appendix 2.

The biggest reason to do this project was time-saving possibilities with automation. The old way of testing required a lot of manual work from technicians e.g., plugging test cables, saving data, naming files, analyzing results, and so on. Estimation of how much time could be saved by automating the test setup can be found in appendix 3.

3 Transmission Line

3.1 Definition of Transmission Line

A transmission line is typically a cable that conducts electromagnetic waves. A basic electrical cable can carry low frequency alternating current but cannot carry currents in radio frequency. Special cable structure and impedance matching enable transmission lines to carry high frequency signals without power losses. It has a uniform shape all the way which defines its characteristics impedance and prevents return losses.

Frequency is measured in hertz (Hz), and it tells how many cycles per second the wave takes. RF lines are often categorized into three classes: low, medium, and high frequency. The spectrum band of these lines ranges from 30 Hz to 300 GHz. High frequency RF lines are defined to operate from 3MHz upwards. [3.]

To clarify what a high frequency line means, it is called a transmission line because it acts differently than a low frequency line. This means that the high frequency line must be analyzed based on its characteristics of signal propagation. All the high frequency interconnects are not automatically categorized as transmission lines; length needs to be bigger than 25% of the signal wavelength to be called so. [4.]

3.2 Coaxial Cable

The most common transmission line is a coaxial cable. Coaxial cable is composed of two concentric conductors and in between of them is dielectric material where electromagnetic field propagates [5, 72]. Characteristic impedance for coaxial cable can vary from 32 to 124 Ω but the typical value for the transmission line is 50 Ω. This was studied to be the best value in Bell Labs 1929 [6; 7].

Coaxial cables can be divided into three different groups: flexible, semi-rigid, and rigid cables. The biggest discrepancy between flexible and rigid cables is that flexible cable has a braided shield when rigid cables have solid metal outer surface. Solid metal surface provides better performance especially in higher frequencies because it can create 100% RF shielding. [8.]

Coaxial cable can be called a two-port network. Both ports can reflect, pass, and/or absorb RF energy. To measure these values, we need to understand scattering parameters, more commonly known as S-parameters. There are four S-parameters in a two-port network: S_{11}, S_{21}, S_{12}, and S_{22}. S_{11} and S_{22} represent reflection coefficients and S_{12} and S_{21} represent transmission coefficients. The reflection coefficient means how much power is reflected into the output port and the transmission coefficient indicates gain or loss between input and output port. Indexation of S-parameter follows the rule: S_{xy}, where $_x$=output port and $_y$=input port. [9.]

3.3 Cryogenic Coaxial Cable

Bluefors manufactures its own coaxial cables to meet the strict requirements to operate reliably in the cryogenic temperatures and survive repeating thermal cycling [10]. Materials of the wires are specified to have low thermal conductance and all wires are thermalized to different temperature stages inside Bluefors cryostat with attenuators. Coaxial lines must be attenuated to block thermal radiation, otherwise the cryostat would not cool down to its base temperature so well.

4 Python

Python is an interpreted programming language with superior features for data analysis. Python is easy to use and read, and you can accomplish difficult tasks with only a few lines of code. However, the main reason to choose Python for this project was the enormous library of packages. Using these powerful libraries saves you lots of time and you can focus on the core task. There is also a big community working with Python so you can easily find help online if needed. [11.]

Codes for plotting the data and TDR analysis were already made with Python, so it was easy to integrate these into this test setup.

4.1 Anaconda

Anaconda is a distribution of Python which is widely used in data analysis and industry. A huge advantage of Anaconda is that it comes with lots of pre-installed packages needed for data analysis, so it saves a lot of time for the end-user. Also, Anaconda's package manager conda is useful because it simplifies package management and saves a lot of time when installing new or updating old packages. [12.]

4.2 Jupyter Notebook

Jupyter Notebook is an open-source computational environment that runs a web application, and it comes with Anaconda. It is constructed from two components: cell and kernel. The user writes the code into a cell and when running it, Jupyter Notebook runs it in the back-end kernel and returns the result. In Jupyter Notebook you can combine multiple different file types in one document, for example, code, text, and picture which is not possible in a normal integrated development environment (IDE). Jupyter Notebook works with various programming languages, such as Julia, Python and R. Name Jupyter is formed out of these three languages. [13.]

Jupyter Notebook was selected for this thesis work due to its superior usability in creating and debugging your code. You do not need to run your whole code because Jupyter Notebook is made of cells that can be run individually. This cell structure allows you to quickly test different approaches and decide which works best for you. Also, the fact that you can include text and pictures in your notebook makes it handy to be used in production. For example, you can easily write instructions on how to use the code inside the notebook instead of having a separate instruction sheet.

4.3 QCoDeS

QCoDeS is a software framework for data acquisition, and it is mainly used in experimental quantum physics. It is a Python-based open-source project developed by Copenhagen, Delft, Sydney, and Microsoft quantum computing consortium. It has lots of useful features from multiple drivers for lab instruments to data storage and free documentation. Using QCoDeS will make things easier for the user and it is a very modern software project which updates all the time thanks to the big community working with it. [14.]

The main reason to select QCoDeS for this thesis work was the driver for the VNA. Even though there was not the exact model driver available, the existing driver could be modified to be suitable for this. Using a driver makes reading and writing commands to VNA easier than going through the manual to find the right standard commands for programmable instruments (SCPI). Another powerful feature that QCoDeS has is called station. A station is a set of instruments that you have in your setup. When you have your station set up, you can use automatic logging whenever you run it. Logging will save all the log messages with timestamps of your station, and this might be useful in case debugging is needed in the future. With the station, you can also print a snapshot of your whole station. Snapshot means that QCoDeS will list all the devices and their parameters to the output panel for easy scrutiny.

5 Hardware Selection

5.1 Network Analyzer

A network analyzer is a device used to perform microwave measurements. A typical network analyzer consists of a display, a signal generator, one or more receivers, and a measuring element. Two basic types of network analyzers are scalar network analyzer and vector network analyzer (VNA). In this thesis work, a VNA was selected to be used since it can measure the magnitude and phase element of the S-parameters when a scalar network analyzer is only capable of measuring magnitude. [5, 816.]

At the time of doing this thesis work, Bluefors had a few different kinds of VNAs. One of these models was selected to be used also in this project, Keysight E5063A. It is a benchtop version that can be mounted to an IT rack, and it has two built-in ports with a dynamic range of 117 decibels (dB), a maximum frequency of 18 GHz, and an output power of 0 decibel-milliwatt (dBm).

5.2 RF Switch Matrix

To measure RF signals between multiple input and output ports an RF switch is needed. RF switch matrix is an array of RF switches. The selected RF switch matrices for this project were LXI Microwave Multiplexers with 36 Sub Miniature version A (SMA) channels. This model can be mounted to an IT rack, and it has a bandwidth of 18 GHz, typical insertion loss is 3 dB at 18 GHz, and the voltage standing wave ratio (VSWR) is 2.2:1 at 18 GHz.

5.3 Cables and Connectors

This setup required lot of different hardware with different connector types. Schematic picture of the devices, cables and connectors can be seen in figure 2.

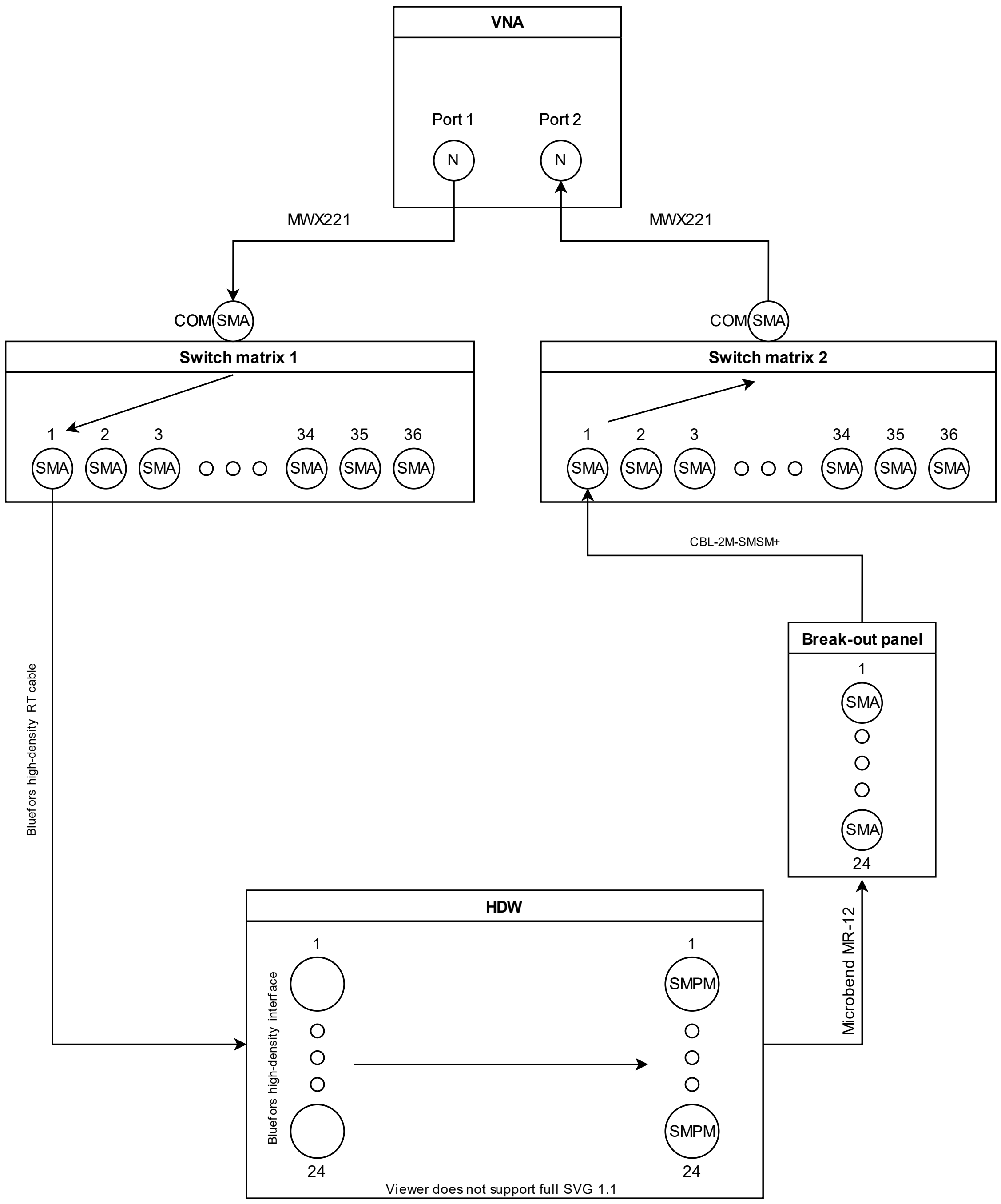


Figure 2. Schematic view of the test setup.

5.3.1 VNA Cables

Keysight VNA has type N connectors and switch matrices have SMA connectors. The cables selected were armored type Junkosha MWX221 with type N plug on the other end and SMA plug on the other. Armoring on the cable prevents it to be over bent but it still maintains its flexibility. Typical insertion loss of the cable is 1.2 dB per meter and VSWR is 1.33:1 at 26.5 GHz.

5.3.2 SMA-SMPM Adapter Cables

HDW uses Sub Miniature Push-On Micro (SMPM) connectors on the other end so to do measurement it is needed to have adapter cable between VNA cable and HDW. The cable selected for this purpose was Huber+Suhner Astrolab Microbend MR-12 with an SMA plug on the other end and an SMPM jack on the other. It is ruggedized and suitable for congested installations due to its small outer diameter. Typical insertion loss of the cable is 1.81 dB and VSWR is 1.45:1 at 18 GHz.

5.3.3 Extension Cables

All the test devices are mounted in a separate IT rack, so it was necessary to have extension cables to reach the test area. The cables selected for this were Mini-Circuits CBL-2M-SMSM+ with an SMA plug on both ends. The cable has extra-rugged construction and strain reliefs on both ends for longer life and steel SMA connectors provide long mating-cycle life. Typical insertion loss of the cable is 4 dB and VSWR is 1.17:1 at 18 GHz.

5.3.4 SMA Feedthroughs

Extension cables and HDW have SMA plugs so an SMA jack-jack adapter was needed. These adapters were attached to a plate and the plate was attached to the test table to avoid any excess movement of the cables. Selected adapters were Mini-Circuits SF-SF50+. These adapters have a rugged stainless-steel body, typical insertion loss is 0.24 dB at 18 GHz and VSWR is 1.40:1 at 18 GHz.

6 Work Process

6.1 Test Setup

The first thing to do was to set up a connection between the computer and devices. This was done with a LAN connection. The computer had two LAN ports, one for the company network and one for the private network which was created for this setup. Devices on the private network were connected by a network switch. Circuit diagram for the setup can be seen in figure 3.

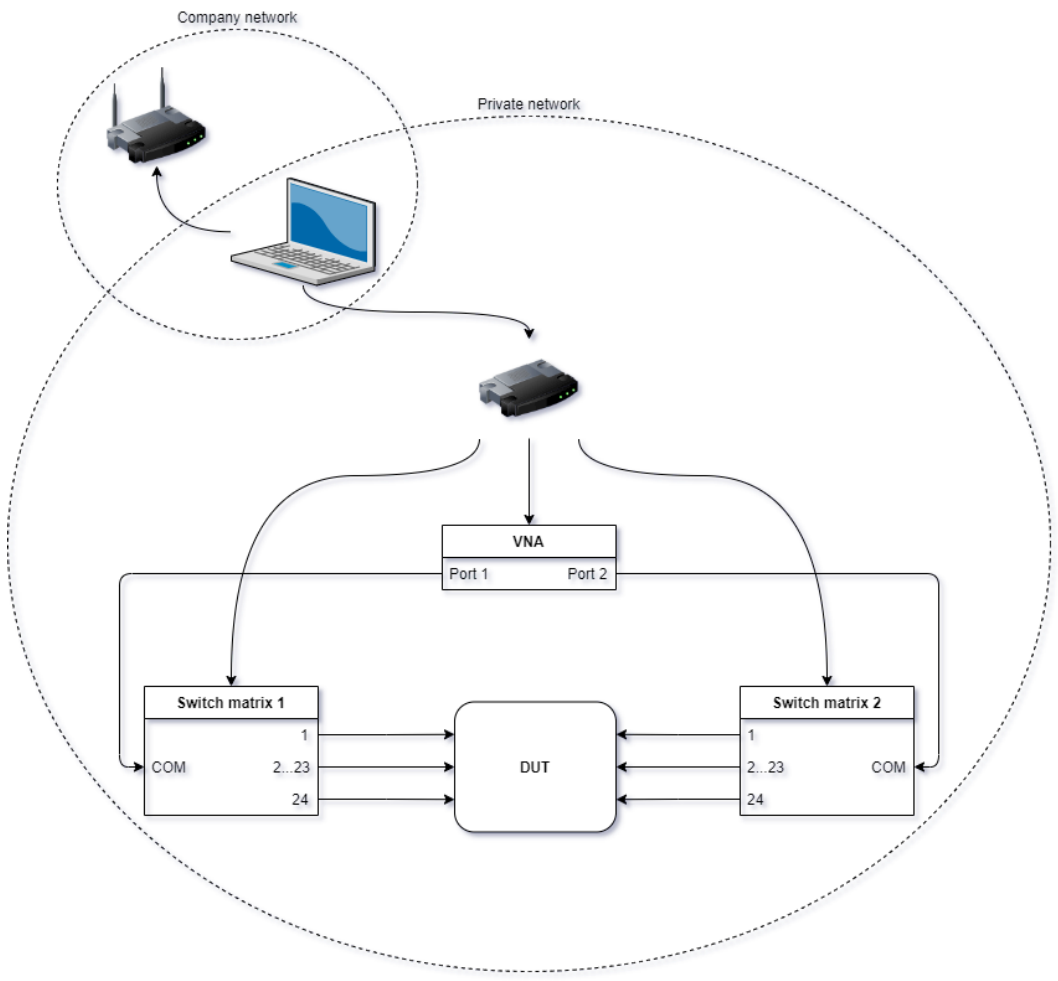


Figure 3. Circuit diagram of the test setup.

This setup required static IP addresses for the computer, VNA, and both switch matrices. These addresses were given and set up by Bluefors IT personnel. Both VNA and switch matrices have a software tool that can be used to set up IP addresses. Keysight Connection Expert was used with VNA and LXI Discovery Tool with switch matrices. After IP addresses were set, the connection could be verified with the same software tools.

To get reliable measurement data, the calibration of the setup needed to be accurate. One option would have been to calibrate each channel of the switch matrices separately and save a calibration file of each of them. Then the correct file would need to be opened in the code every time before the channel is activated. This is possible but it would require a lot of work. In this project, it was decided to do tests first to see how much each channel varies from the other.

VNA was calibrated with Keysight ECal module N4691D and channel 1 open on both switch matrices. Reference planes were needed to be at the end of the SMA connectors on switch matrices. ECal module could not be plugged directly into both channels, so an adapter cable had to be used in between the ECal module and switch matrix 2. After calibration, a port extension function was performed on the VNA that allows you to electrically move the calibration reference plane to get rid of the adapter cable.

Mini-Circuits extension cables were used as test devices because their electrical lengths should be the same. 24 cables were attached to the first 24 channels of the switch matrices and test run was saved as .s2p files. Plotted S-parameters S21 and S11 of this test can be seen in figures 4 and 5. Figures show that cables are not broken and there are no significant differences between them and channels. Figures 6 and 7 show plotted S-parameter S21 and S11 of channels 1 and 2. Here we can see bit better that the uncalibrated channel 2 has < 0.5 dB difference on insertion loss at 18 GHz and it is a bit noisier. Return loss of the channels is < 5dB.

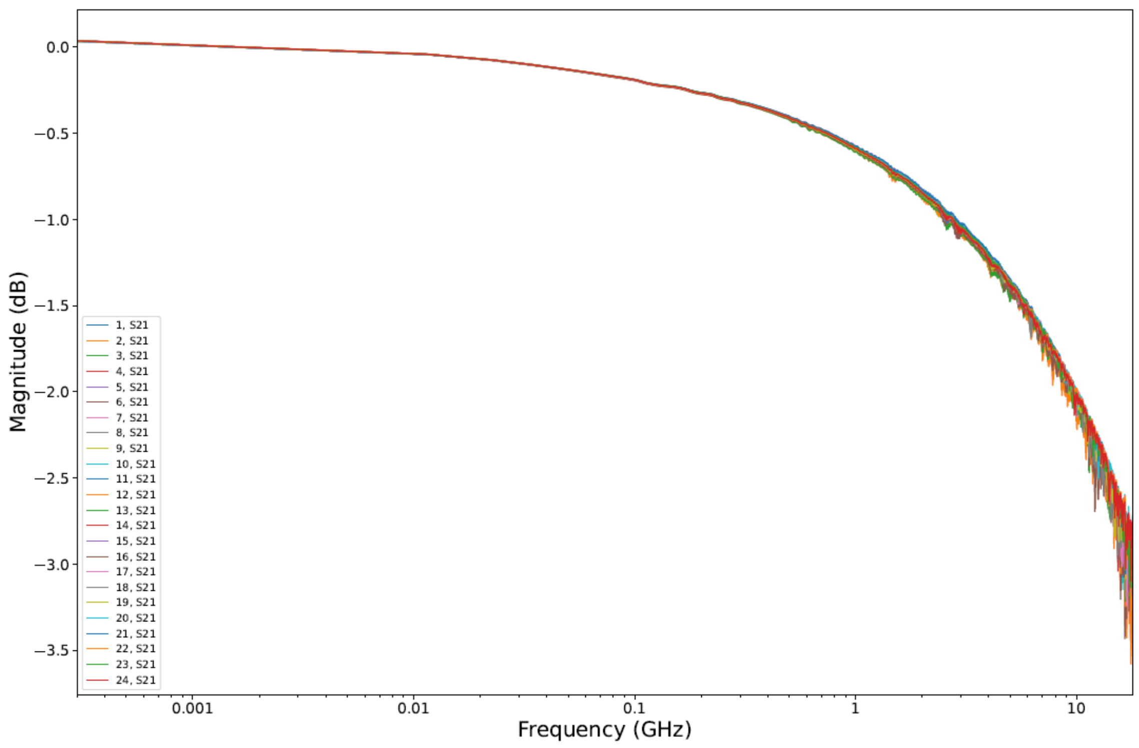


Figure 4. Plotted S-parameter S21 of 24 Mini-Circuits CBL-2M-SMSM+ to compare switch channels between each other's.

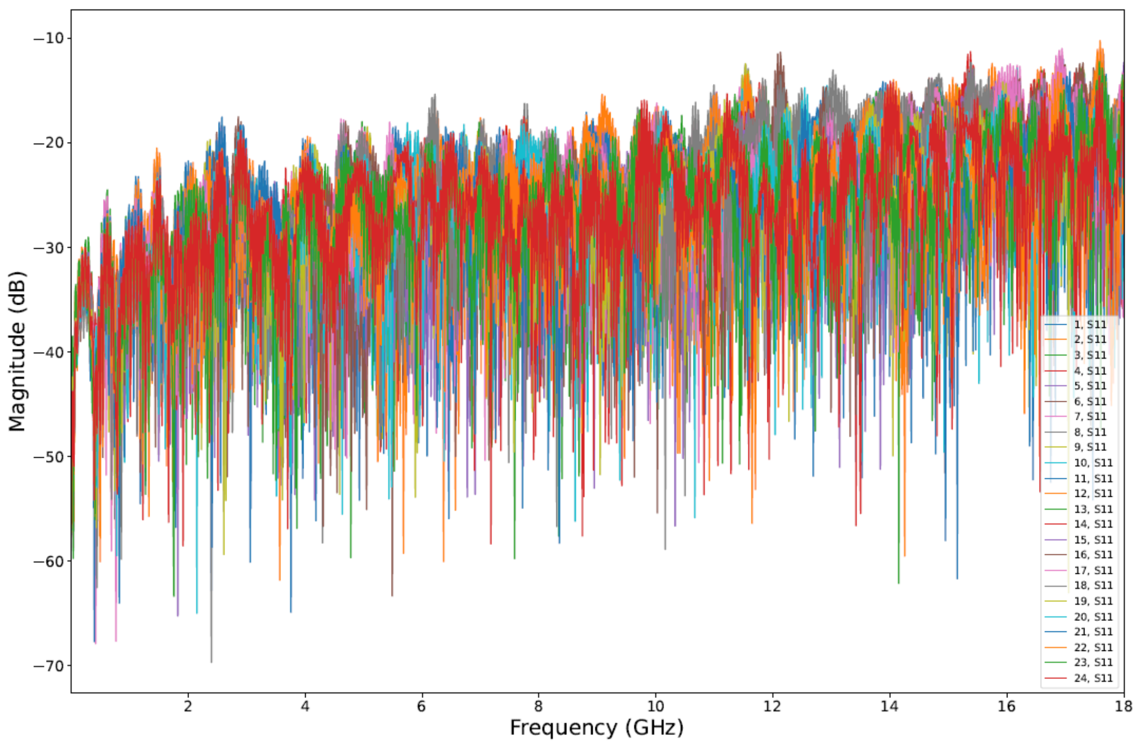


Figure 5. Plotted S-parameter S11 of 24 Mini-Circuits CBL-2M-SMSM+ to compare switch channels between each other's.

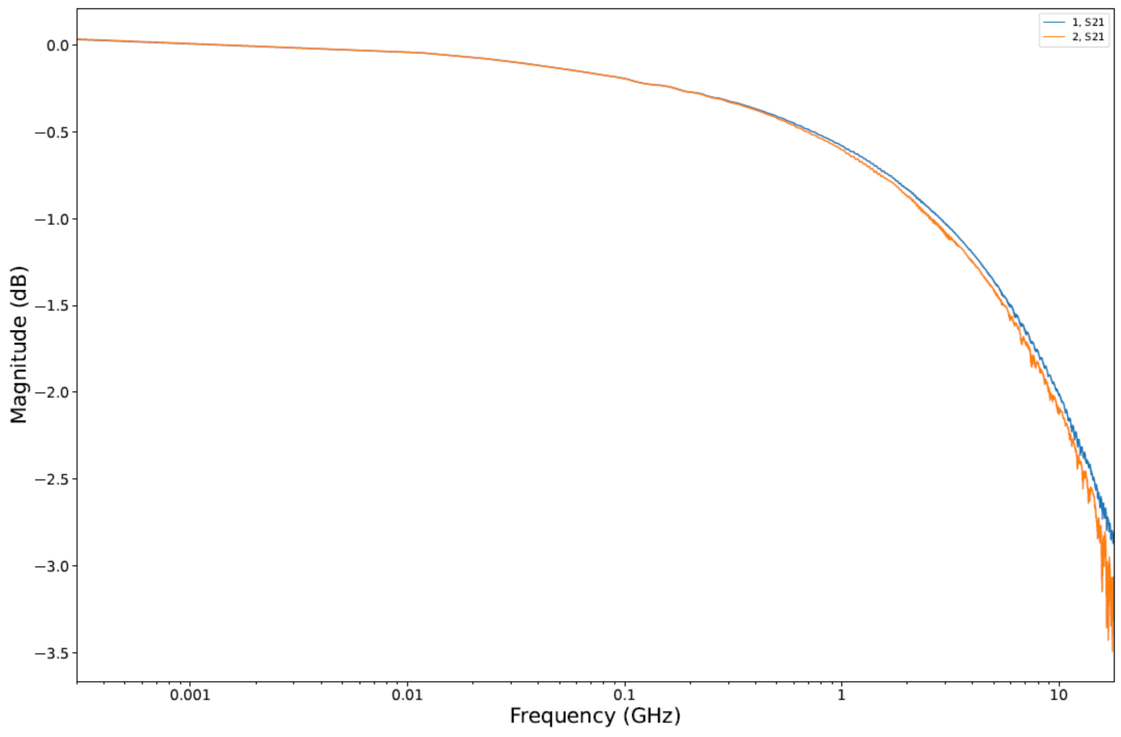


Figure 6. Plotted S-parameter S21 of 2 Mini-Circuits CBL-2M-SMSM+ to compare switch channels between each other’s.

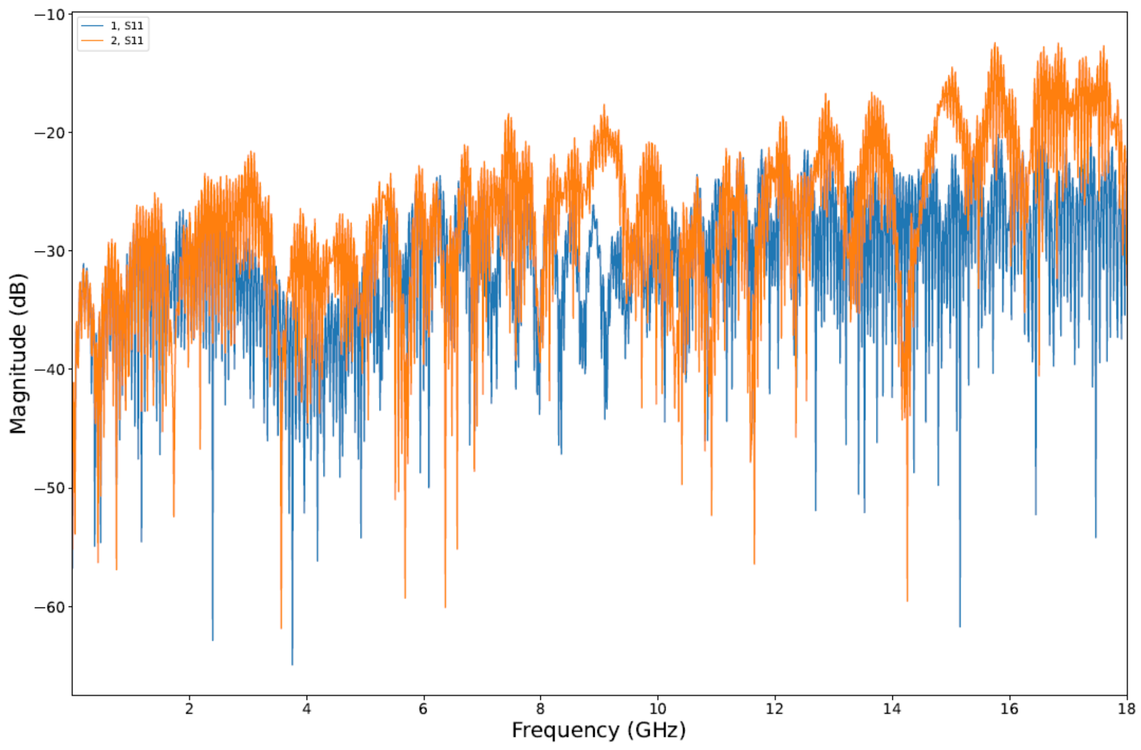


Figure 7. Plotted S-parameter S11 of 2 Mini-Circuits CBL-2M-SMSM+ to compare switch channels between each other’s.

The next step was to find out the electrical length and time delay of the channels. Bluefors had already code to calculate these, so it was used. Figure 8 shows mean time delay, mean electrical length, the standard deviation of time delay, and the standard deviation of electrical length.

```
mean time delay is: 9560.76 ps
standard deviation time delay is: 5.14 ps
mean electrical length is: 2006.37 mm
standard deviation electrical length is: 1.08 mm
```

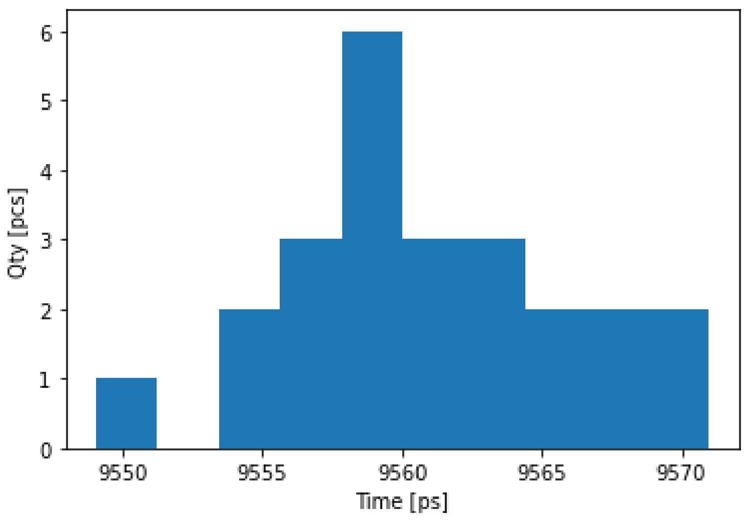


Figure 8. Deviation of the cables in time.

The biggest deviation in electrical length between channels was calculated to be 4,6 millimetres and in time delay the biggest deviation was 17 picoseconds. This result was good enough to proceed with just having the calibration on one channel and using it on the others.

6.2 QCoDeS Drivers

QCoDes has premade drivers for various lab instruments which help users to operate them. When using the driver, basic communication works with a `set()` and `get()` commands. These are standardized methods to perform `Instrument.write()` and `Instrument.ask()` commands in QCoDeS. Operating instruments with SCPI commands would take much more time and the code itself would become hard to understand. Own functions can be also defined into the driver. Using functions makes the code easier to read and modify.

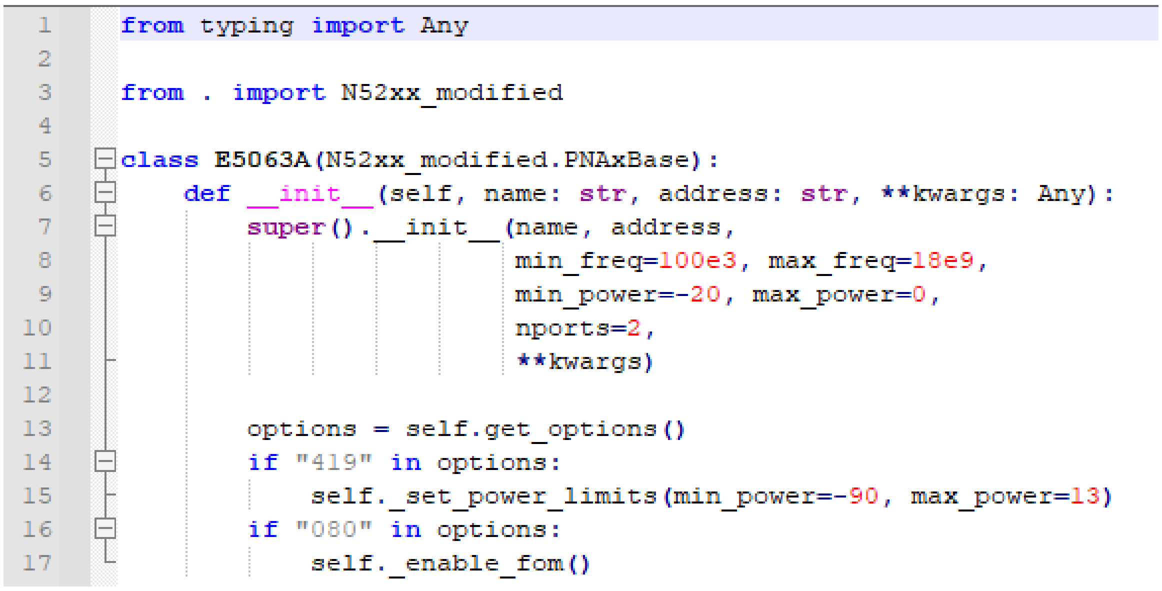
In this case, there was not the exact driver for Keysight E5063A VNA available so an existing driver for a different model Keysight VNA needed to be modified. The selected base driver from QCoDeS Keysight instrument drivers' catalog was N52xx and the specified driver for the VNA was N5245A.

RF switch matrix did not have a driver that could have been used, so a new one needed to be done. Making a driver from the scratch is a big task so it was decided to do just a bare minimum to get it working.

6.2.1 Driver for Network Analyzer

QCoDeS driver for N52xx was used as a base for the new driver called N52xx_modified. There were few issues in the driver with `def traces(self) -> ChannelList:` section which was supposed to update channel list with active traces and return the new list. To fix this, some of the commands were commented out. Some other smaller sections in the original driver were also commented out. All new functions and changes made to existing ones in the driver will be presented in chapter 6.3. The whole N52xx_modified driver can be seen in appendix 4.

A specified driver for Keysight 5063A was created by using the base of the existing driver of N5245A. Frequency, power limits and the number of ports were changed to correct ones based on the VNA model used. Also, the imported base file was changed to be the modified one. This driver can be seen in figure 9.



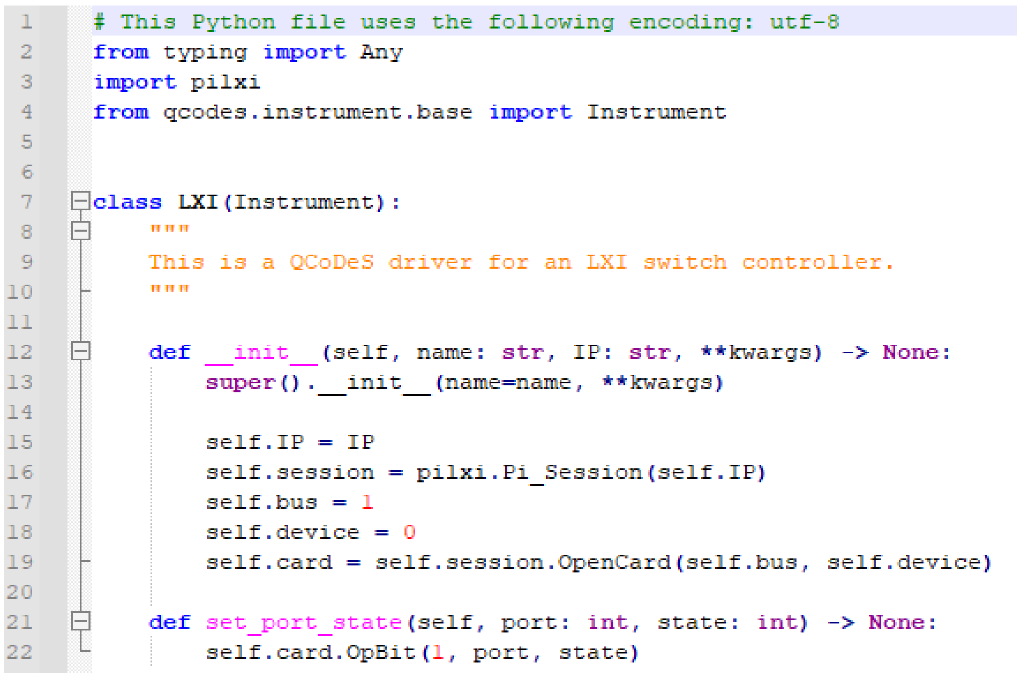
```
from typing import Any

from . import N52xx_modified

class E5063A(N52xx_modified.PNAxBase):
    def __init__(self, name: str, address: str, **kwargs: Any):
        super().__init__(name, address,
                         min_freq=100e3, max_freq=18e9,
                         min_power=-20, max_power=0,
                         nports=2,
                         **kwargs)

        options = self.get_options()
        if "419" in options:
            self._set_power_limits(min_power=-90, max_power=13)
        if "080" in options:
            self._enable_fom()
```

Figure 9. Driver for E5063A.

6.2.2 Driver for Switch Matrices

Creating driver for switch matrices (figure 10) was done based on the way how the driver was made for 5063A. Imported dependencies for switches are the Python library for LXI Driver `pilxi` from Pickeringtest and `Instrument` from QCoDeS. Pickeringtest has examples for different languages on how to operate the switches. Examples for Python were studied to create this driver.

A function called `set_port_state` was created for the driver. This function specifies the state of the channel, and it takes two arguments: port and state. Port defines which channel on the switch is activated and the state defines whether the channel is on (1) or off (0). Usage of this function can be seen in figure 20.



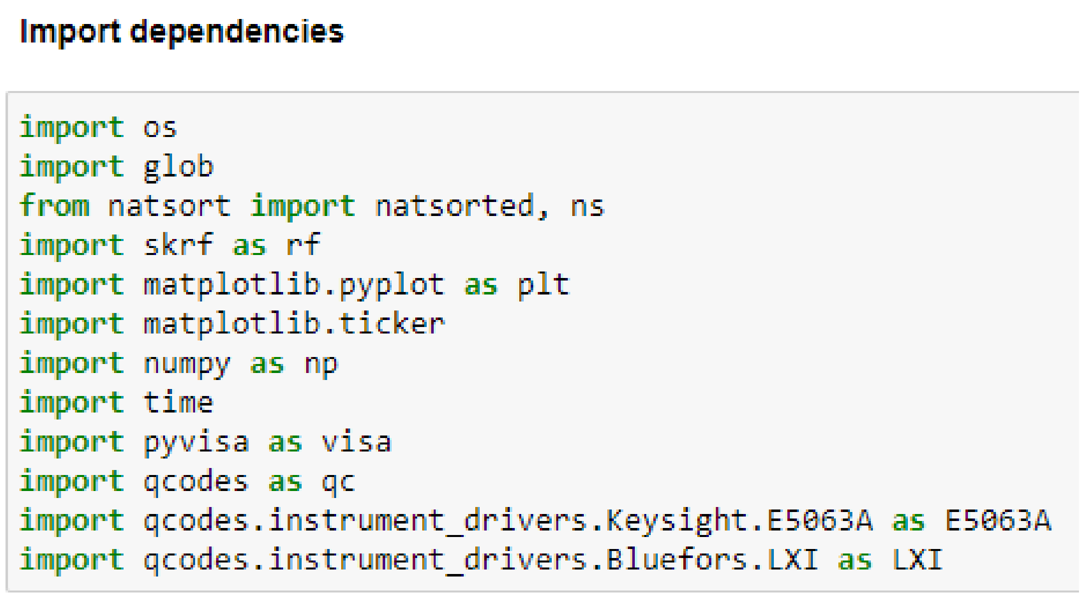
```python
# This Python file uses the following encoding: utf-8
from typing import Any
import pilxi
from qcodes.instrument.base import Instrument


class LXI(Instrument):
    """
    This is a QCoDeS driver for an LXI switch controller.
    """

    def __init__(self, name: str, IP: str, **kwargs) -> None:
        super().__init__(name=name, **kwargs)

        self.IP = IP
        self.session = pilxi.Pi_Session(self.IP)
        self.bus = 1
        self.device = 0
        self.card = self.session.OpenCard(self.bus, self.device)

    def set_port_state(self, port: int, state: int) -> None:
        self.card.OpBit(1, port, state)
```

Figure 10. Driver for switch matrix.

6.3 Programming

Programming was made in Jupyter Notebook with an activated QCoDeS environment. This section describes how these steps were made and what is their function in the whole code.

First, all necessary dependencies are imported (figure 11). In Python, importing works with the `import` -command. If the whole dependency is not needed, it can be imported using `from X import Y` -command. To make the coding easier, packages are often imported with the `import as` -command which allows you to use the package with a shorter and easier name. All dependencies will be described in more detail later when they are used.

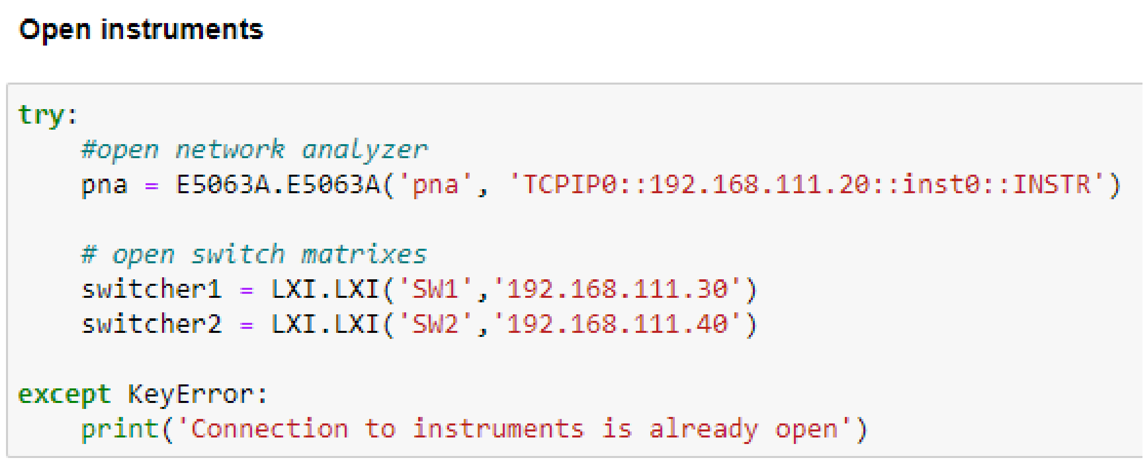


Import dependencies

```
import os
import glob
from natsort import natsorted, ns
import skrf as rf
import matplotlib.pyplot as plt
import matplotlib.ticker
import numpy as np
import time
import pyvisa as visa
import qcodes as qc
import qcodes.instrument_drivers.Keysight.E5063A as E5063A
import qcodes.instrument_drivers.Bluefors.LXI as LXI
```

Figure 11. List of imported dependencies.

In the next cell, a connection between the computer and instruments is opened (figure 12). This is done by creating new variables called `pna, switcher1,` and `switcher2`. These variables will use imported dependencies 5063A or LXI. The reason why the variable name was pna instead of ena is that the 5063A driver is made for a PNA network analyzer, not for ENA which is used in this project.



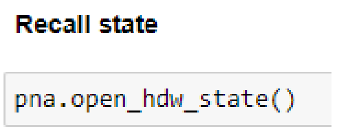
Open instruments

```
try:
    #open network analyzer
    pna = E5063A.E5063A('pna', 'TCPIP0::192.168.111.20::inst0::INSTR')

    # open switch matrixes
    switcher1 = LXI.LXI('SW1','192.168.111.30')
    switcher2 = LXI.LXI('SW2','192.168.111.40')

except KeyError:
    print('Connection to instruments is already open')
```

Figure 12. Opening connection to the instruments.

In both dependencies' driver files, there is a `def __init__` -section which is a constructor for a class of the instrument. This constructor demands two arguments: name and address. Given names are pna, SW1, and SW2 and addresses are static IP addresses for the devices. If the connection is already opened to the network analyzer, a KeyError will be raised, and this is the reason why opening the instruments section was placed inside a `try-except` structure. If the error is raised, the user will only see the text "Connection to instruments is already open" and can continue to the next cell.

The next cell is a function that recalls a correct state to the VNA (figure 13). When recalling a file, VNA opens a state that has all the parameters set and active calibration.

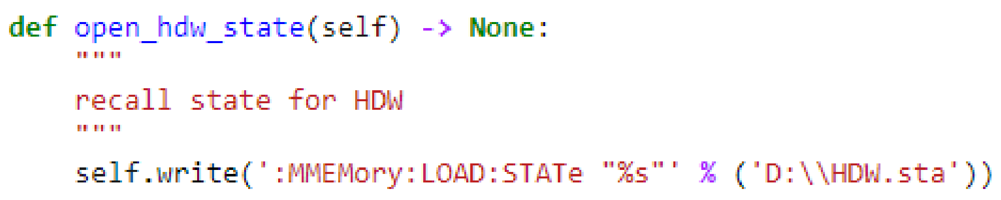


Recall state

```
pna.open_hdw_state()
```

Figure 13. Recall state function.

Function `open_hdw_state` is defined in the driver (figure 14). This function was not included in the original driver, so it was made for this project. It writes a SCPI command to open a file called HDW.sta from root D in VNA.

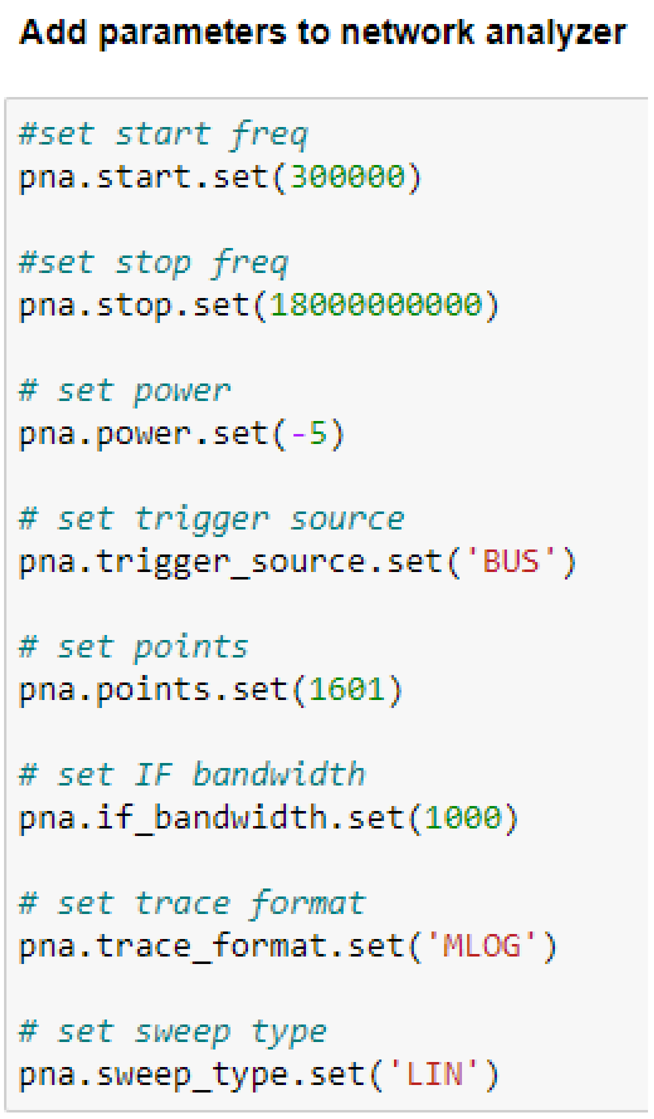


```
def open_hdw_state(self) -> None:
    """
    recall state for HDW
    """
    self.write(':MMEMory:LOAD:STATe "%s"' % ('D:\\HDW.sta'))
```

Figure 14. Function to recall state in VNA.

The power of QCoDeS can be seen in figure 15 where VNA parameters can be modified with simple commands. These parameters are already set in the opened state file but in this cell, the user can easily try different settings if it is needed. In this case, the settings are:

- start frequency = 300 MHz
- stop frequency = 18 GHz
- output power = -5 dBm
- trigger source = bus
- number of points = 1601
- intermediate frequency (IF) bandwidth = 1 KHz
- trace format = logarithmic magnitude
- sweep type = linear frequency.

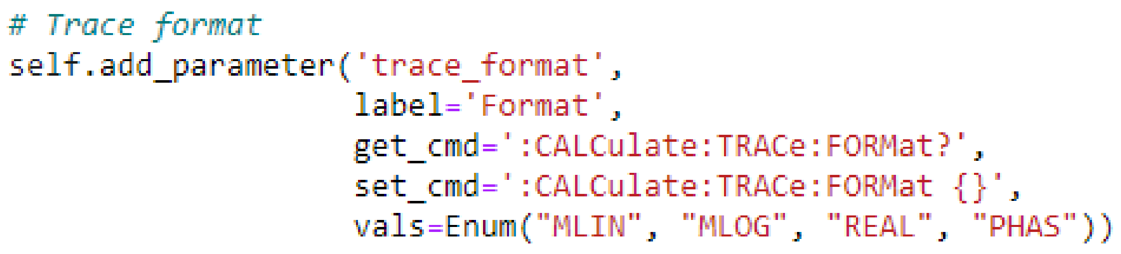


Add parameters to network analyzer

```
#set start freq
pna.start.set(300000)

#set stop freq
pna.stop.set(18000000000)

# set power
pna.power.set(-5)

# set trigger source
pna.trigger_source.set('BUS')

# set points
pna.points.set(1601)

# set IF bandwidth
pna.if_bandwidth.set(1000)

# set trace format
pna.trace_format.set('MLOG')

# set sweep type
pna.sweep_type.set('LIN')
```

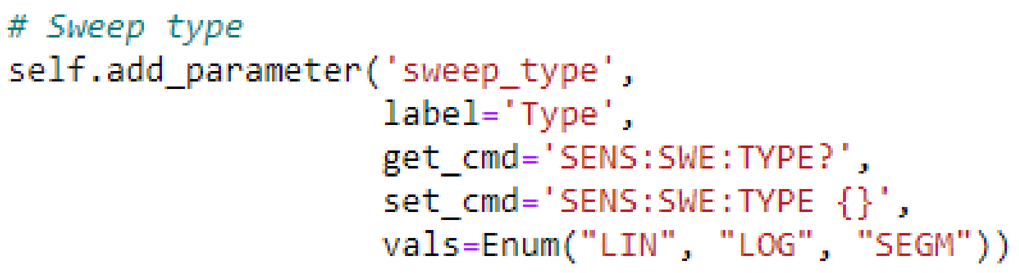
Figure 15. List of added parameters to the network analyzer.

All the commands were in the original driver except the ones for trace format and sweep type. These were added as new parameters to the driver (figures 16 and 17). `Get_cmd` and `set_cmd` commands are defined with correct SCPI commands and a variable `vals` has trace formats and sweep types that are available in VNA.



```
# Trace format
self.add_parameter('trace_format',
                   label='Format',
                   get_cmd=':CALCulate:TRACe:FORMat?',
                   set_cmd=':CALCulate:TRACe:FORMat {}',
                   vals=Enum("MLIN", "MLOG", "REAL", "PHAS"))
```

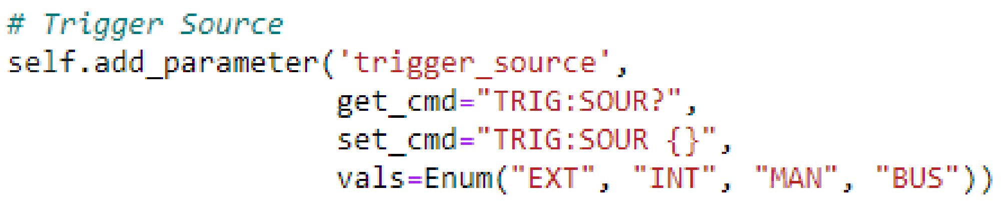
Figure 16. New parameter for trace format.



```
# Sweep type
self.add_parameter('sweep_type',
                   label='Type',
                   get_cmd='SENS:SWE:TYPE?',
                   set_cmd='SENS:SWE:TYPE {}',
                   vals=Enum("LIN", "LOG", "SEGM"))
```

Figure 17. New parameter for sweep type.

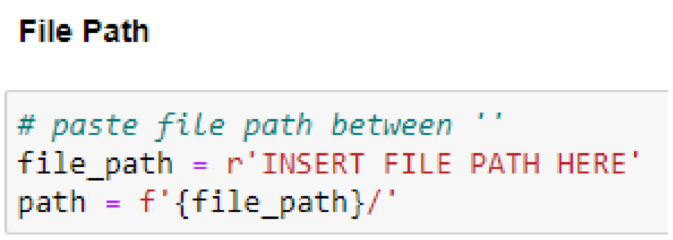
Also trigger source was missing the needed type “BUS” which means that the trigger is activated by software. This was added to variable `vals` (figure 18).



```
# Trigger Source
self.add_parameter('trigger_source',
                   get_cmd="TRIG:SOUR?",
                   set_cmd="TRIG:SOUR {}",
                   vals=Enum("EXT", "INT", "MAN", "BUS"))
```

Figure 18. Updated parameter for trigger source.

Before running the actual measurement, the user needs to specify the file directory where the data will be saved, and this can be seen in figure 19. The file path can be copied straight from the file explorer and pasted between quotations marks. Letter r before quotes means that Python creates a raw string from the pasted file path. Raw string treats backlashes as literal characters, not as escape characters as they are in a normal string. After this, a new variable called `path` is created that uses the f-string method to add one backlash to the end of the file path.

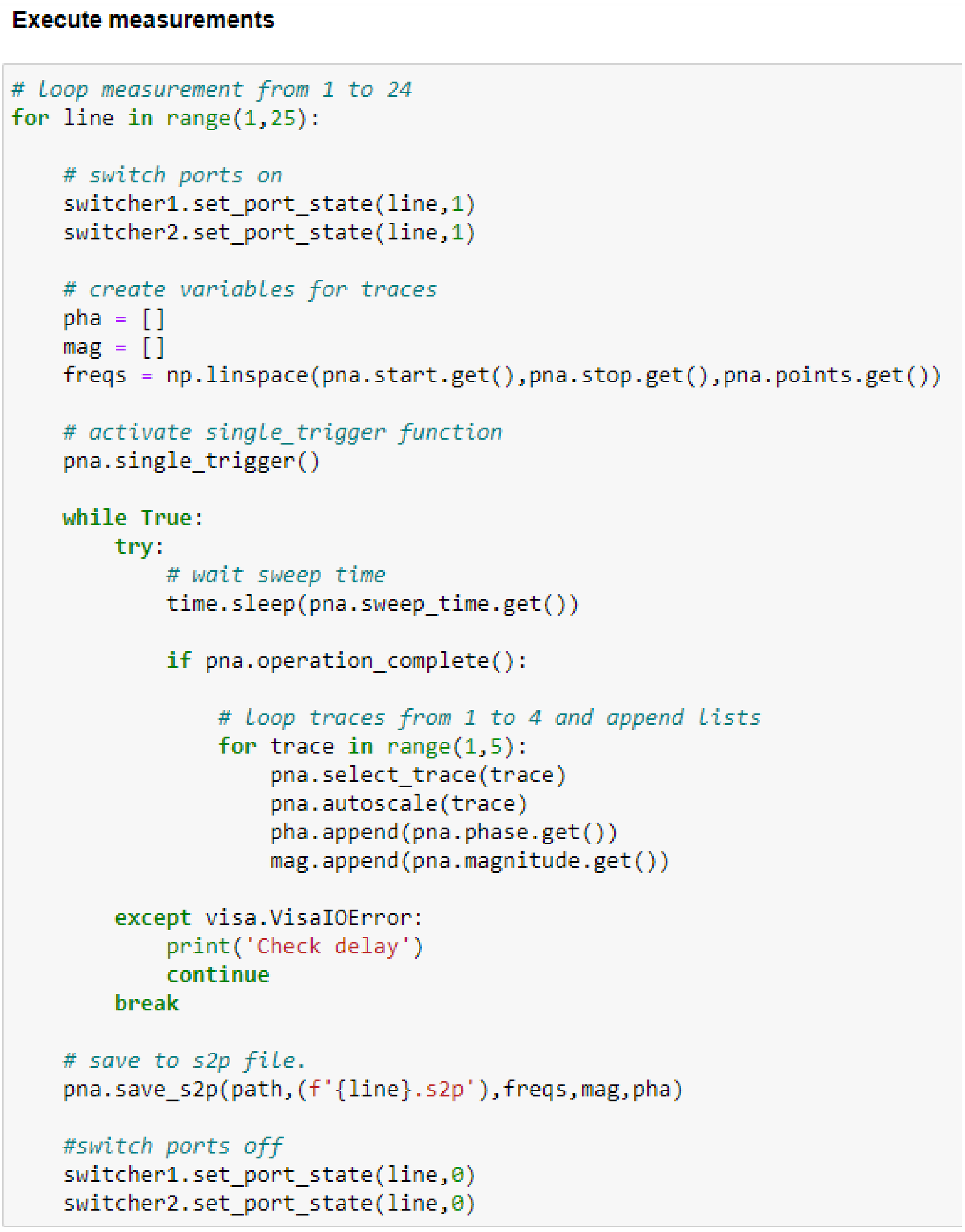


File Path

```
# paste file path between ''
file_path = r'INSERT FILE PATH HERE'
path = f'{file_path}/'
```

Figure 19. Defining a file directory.

The next cell, seen in figure 20, is making the actual measurements. First, a for loop is created to run measurement 24 times. At each loop, the sequence number is saved to a variable called `line`.



Execute measurements

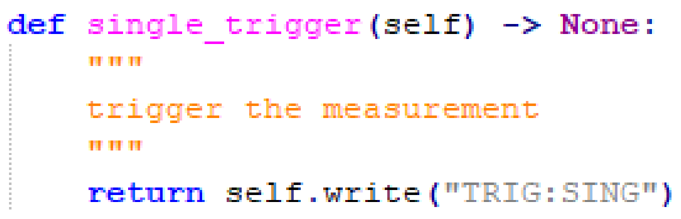
```python
# loop measurement from 1 to 24
for line in range(1,25):

    # switch ports on
    switcher1.set_port_state(line,1)
    switcher2.set_port_state(line,1)

    # create variables for traces
    pha = []
    mag = []
    freqs = np.linspace(pna.start.get(),pna.stop.get(),pna.points.get())

    # activate single_trigger function
    pna.single_trigger()

    while True:
        try:
            # wait sweep time
            time.sleep(pna.sweep_time.get())

            if pna.operation_complete():

                # loop traces from 1 to 4 and append lists
                for trace in range(1,5):
                    pna.select_trace(trace)
                    pna.autoscale(trace)
                    pha.append(pna.phase.get())
                    mag.append(pna.magnitude.get())

        except visa.VisaIOError:
            print('Check delay')
            continue
        break

    # save to s2p file.
    pna.save_s2p(path,(f'{line}.s2p'),freqs,mag,pha)

    #switch ports off
    switcher1.set_port_state(line,0)
    switcher2.set_port_state(line,0)
```

Figure 20. Run measurements for channels 1-24.

At the start of each loop, the `set_port_state` -function is activated. This function was defined in the driver for switch matrix (figure 10). It uses a variable `line` to define the channel which needs to be switched on.

After this, 3 new variables are created: `pha`, `mag`, and `freqs`. `Pha` and `mag` are empty lists, and `freqs` is a list of a start frequency, stop frequency and number of points. These values are processed with the `np.linspace` - method from the imported dependency `numpy` which returns the values evenly spaced. These 3 variables are needed when saving the measurement data.

Next, a function called `single_trigger` is activated. It starts a sweep that stops after the sweep is done. The function was added to the driver, and it writes a SCPI command to VNA to run a single sweep (figure 21).

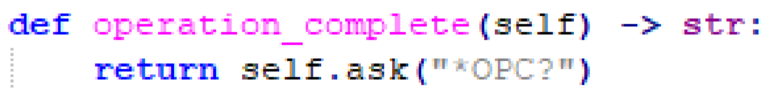


```python
def single_trigger(self) -> None:
    """
    trigger the measurement
    """
    return self.write("TRIG:SING")
```

Figure 21. Function to activate a single trigger.

The next phase is placed inside a `while` loop with Boolean value `True`. This means that the loop will continue if the value is true. Inside `while` loop is a `try-except` structure and command `break`. First Python goes to `try` section and if it can't be completed, it will move to `except`. If it can be completed, `except` will be skipped, and the `break` command will stop the `while` loop. `Except` structure uses imported dependency `pyvisa` to identify `VisaIOError`.

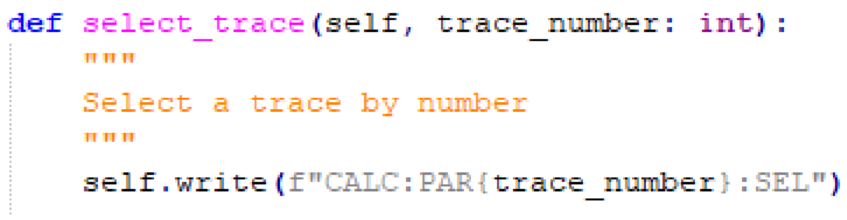
The first thing inside `try` is an operation `time.sleep()` from imported dependency `time` to wait the time it takes to do a single sweep. To get the time required for the single sweep, a QCoDeS command `sweep_time.get()` was used. After this a condition statement `if` was used to check if function `operation_complete()` is ready. This function was added to the driver (figure 22). It returns a string 0 if some operation is still ongoing and 1 when all pending operations are completed. If this function returns 0, Python will move to `except` section that prints a string "Check delay" to the output panel so that the user knows there is something wrong with the wait time. After this, a command `continue` will activate `try` from the beginning.



```
def operation_complete(self) -> str:
    return self.ask("*OPC?")
```

Figure 22. Function to check if an operation is completed.

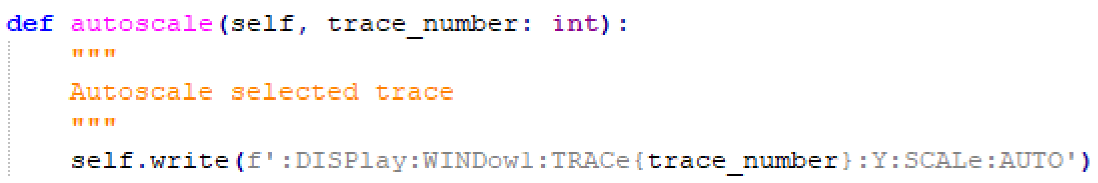
When testing HDW, there are four different traces on the VNA screen: S11, S12, S21, and S22. All these traces are looped through in the next step. First, a function called `select_trace()` selects the trace using variable `trace` which is an integer defined by the running number of the loop. This function did not exist in the driver, so it was created (figure 23). This function requires one argument `trace_number` which needs to be an integer and it writes a SCPI command to select a given trace.



```
def select_trace(self, trace_number: int):
    """
    Select a trace by number
    """
    self.write(f"CALC:PAR{trace_number}:SEL")
```

Figure 23. Function to select trace.

Next, a function called `autoscale()` is activated. It uses the same variable `trace` to identify which trace needs to be scaled. This function didn't exist in the driver, so it was created (figure 24). It requires one argument `trace_number` that needs to be an integer and it writes a SCPI command to auto-scale given trace.

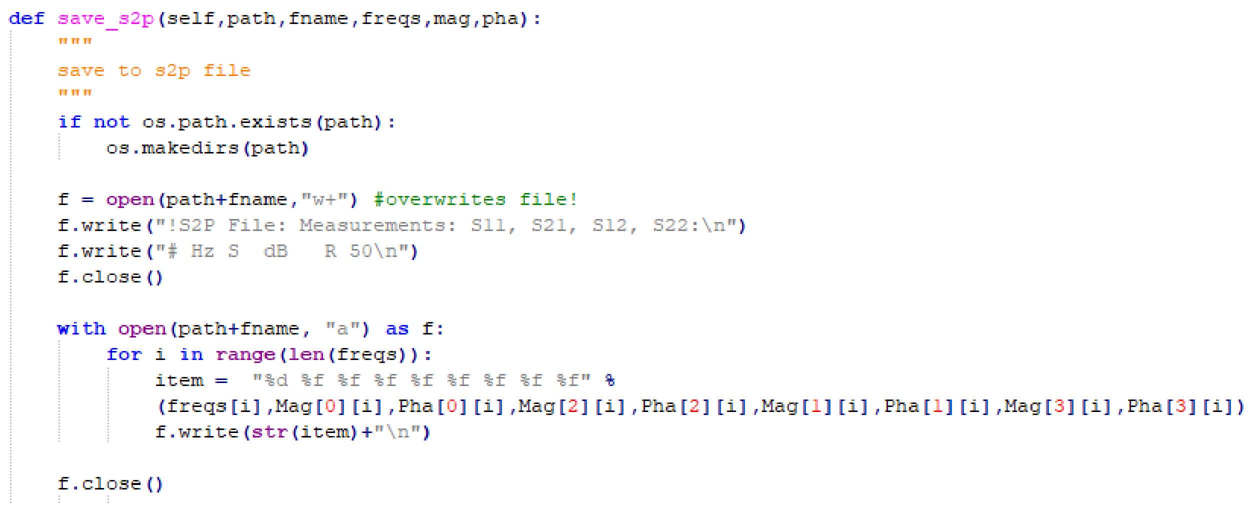


```
def autoscale(self, trace_number: int):
    """
    Autoscale selected trace
    """
    self.write(f':DISPlay:WINDow1:TRACe{trace_number}:Y:SCALe:AUTO')
```

Figure 24. Function to scale selected trace.

The next two steps are saving phase and magnitude information to existing empty lists `pha` and `mag` using the Python list append -method. Saved information is gathered using QCoDeS commands `phase.get()` and `magnitude.get()`.

Data is saved into Touchstone format using a function called `save_s2p` that requires five variables: `path`, `line`, `freqs`, `mag`, and `pha`. The variable line is used with the f-string method to add the string “s.2p” to the end of it to save it in Touchstone format. This function did not exist in the driver, so it was added (figure 25). It requires 5 arguments: `path`, `fname`, `freqs`, `mag`, and `pha`. The path is the file directory where data is saved, and it is defined by the user in a variable `path` (figure 19). Fname is the name of the file, and it is defined in the variable `line`. This means that test results will be saved as .s2p files by number (1.s2p, 2.s2p, 3.s2p etc.). Arguments `freqs`, `mag`, and `pha` are using variables with the same names which are Python lists.

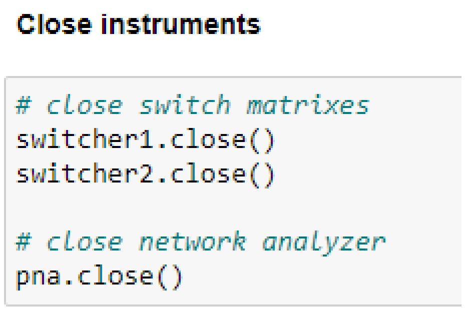


```
def save_s2p(self,path,fname,freqs,mag,pha):
    """
    save to s2p file
    """
    if not os.path.exists(path):
        os.makedirs(path)

    f = open(path+fname,"w+") #overwrites file!
    f.write("!S2P File: Measurements: S11, S21, S12, S22:\n")
    f.write("# Hz S  dB   R 50\n")
    f.close()

    with open(path+fname, "a") as f:
        for i in range(len(freqs)):
            item =  "%d %f %f %f %f %f %f %f %f" %
            (freqs[i],Mag[0][i],Pha[0][i],Mag[2][i],Pha[2][i],Mag[1][i],Pha[1][i],Mag[3][i],Pha[3][i])
            f.write(str(item)+"\n")

    f.close()
```

Figure 25. Function to save data in Touchstone format.

After data is saved, switcher channels are switched off using function `set_port_state` which was defined in the driver (figure 10). The same function was used earlier to switch channels on by giving arguments `line` and integer 1 but this time the second argument is integer 0.

The last cell in the program is to close connections to instruments (figure 26).



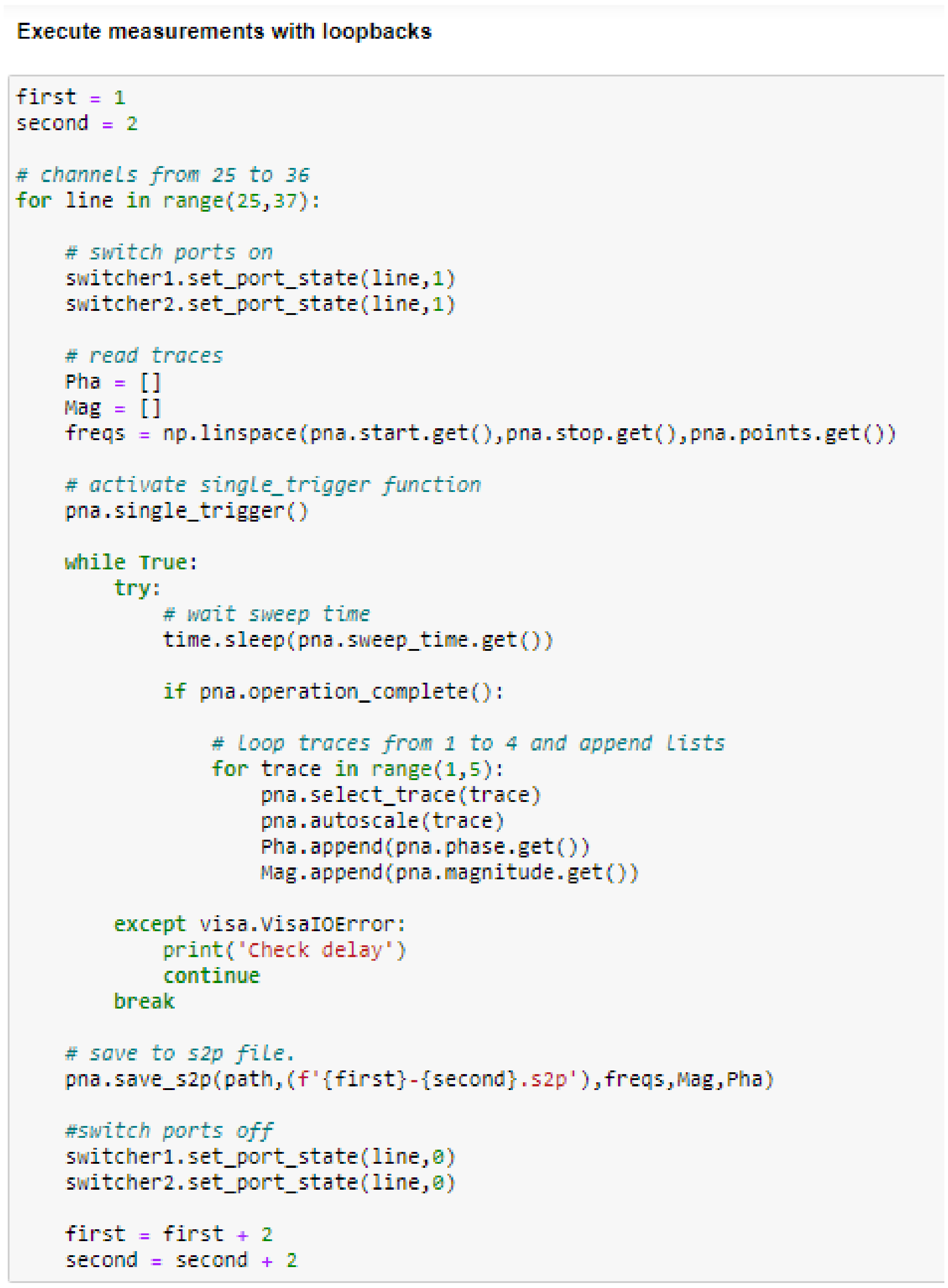
Close instruments

```
# close switch matrixes
switcher1.close()
switcher2.close()

# close network analyzer
pna.close()
```

Figure 26. Closing connection to instruments.

6.3.1 Additional Steps in the Program

There are three additional steps in the program which are used when testing HDW: loopback measurement, data plotting, and TDR. To measure lines inside a cryostat, individual lines need to be connected to pairs on the bottom of the HDW. To measure this setup, the main code was modified to a new code (figure 27). Switch matrices have 36 channels so there were still 12 unused. These 12 channels were decided to be used for the loop measurements. The main idea for the code is the same as in figure 14 but there are few differences. This code starts with defining two variables, `first` and `second`, which are used as counters, and the `for` loop is modified to run measurements from channels 25 to 36. The actual measurement section works the same way, but the difference comes when saving the file. Because we are now measuring two lines at the same time, the file needs to be saved with both line numbers. This is done using f-string in `save_s2p` function to insert variables `first` and `second` with a hyphen in between. The last step is to add integer 2 to counter variables `first` and `second` to measure the next pair in the next round of the loop.



Execute measurements with loopbacks

```
first = 1
second = 2

# channels from 25 to 36
for line in range(25,37):

    # switch ports on
    switcher1.set_port_state(line,1)
    switcher2.set_port_state(line,1)

    # read traces
    Pha = []
    Mag = []
    freqs = np.linspace(pna.start.get(),pna.stop.get(),pna.points.get())

    # activate single_trigger function
    pna.single_trigger()

    while True:
        try:
            # wait sweep time
            time.sleep(pna.sweep_time.get())

            if pna.operation_complete():

                # loop traces from 1 to 4 and append lists
                for trace in range(1,5):
                    pna.select_trace(trace)
                    pna.autoscale(trace)
                    Pha.append(pna.phase.get())
                    Mag.append(pna.magnitude.get())

        except visa.VisaIOError:
            print('Check delay')
            continue
        break

    # save to s2p file.
    pna.save_s2p(path,(f'{first}-{second}.s2p'),freqs,Mag,Pha)

    #switch ports off
    switcher1.set_port_state(line,0)
    switcher2.set_port_state(line,0)

    first = first + 2
    second = second + 2
```

Figure 27. Run measurements for channels 25-36 in pairs.

For plotting the data and running TDR analysis, Bluefors made scripts were used. Imported dependencies `glob`, `natsort`, `skrf`, and `matplotlib` are used for these. Scripts were modified to use the same variables to read saved data from the right file directory and save the plotted figures there. Both scripts are in their cells in Jupyter Notebook for easy operation.

6.3.2 Version Controlling

This project was created over three different computers so proper version control was needed. USB stick can get lost or corrupted, so version controlling was chosen to be done via Git.

Git is a version control system that tracks changes made to the files and allows users to go back to older files if an error was noticed in the new file [15]. With Git I used GitLab that offers cloud storage to store a copy of the repository that contains all the files and revision history.

7 Results

This project was finished and taken into production in August 2021. Some modifications were still made in September when there was a bit more experience from production, but all the main functions and ideas stayed the same. Before starting the project, it was estimated that measuring 24 lines would take 5 minutes and this was proved to be very close to the truth. The actual measurement section takes around 3 minutes and when you add the time that takes attaching test wires, we are close to the estimation. This is a massive time saving compared to the old style which could take up to 60 minutes to perform the same tests.

Another benefit of this project was to simplify and standardize the test process. Now it is easy for a technician to plug in the test cables and run all the measurements in the same user interface. There is no need any more to run multiple tests and save them to USB sticks to get them analyzed. If some error occurs in the validation, a technician can quickly debug it while HDW is still plugged into the test setup.

All required functions were included in this project but during it, there were multiple new improvement ideas for this setup. QCoDeS has a built-in function for logging which was tried to be implemented, but it was noticed to be a too big task for this time frame. Since this setup was using the modified driver for the VNA and switch matrices had the bare minimum driver, logging did not work as it was intended. It is anyhow possible to have, and it would be a great addition to this. Another improvement idea was to improve plotting and TDR with Python module Scikit RF. Now, these functions plot the data for the user, but they could also do the validation automatically. This would require a lot of testing, but this is something that could make employees' work easier.

Hardware for the test setup will be updated after more testing is done and the best practices are found. Now the setup has a bit too long test cables and wrong genders. It would be good to calibrate each channel of the switch matrices after the hardware is updated to perform more accurate measurements.

Based on the results of this thesis work, Bluefors has decided to purchase similar test setup to be used in cryohall where cryostats are built. Final validation of the HDW is done in cryogenic temperatures inside the cryostat so that setup is much needed. It is expected to have similar time saving as in this thesis work and it also standardizes measurements performed in cryohall.

References

1 Bluefors Oy. High-Density Wiring. Digital photograph. https://bluefors.com/wp-content/uploads/2020/02/bluefors-high-density-wiring-side-loading-tree4-e-800-2.png Accessed 16.01.2022.

2 Bluefors Oy. High-Density Wiring. Online. https://bluefors.com/products/high-density-wiring/#product-overview Read 16.01.2022.

3 Scarpati, Jessica. 2021. Radio frequency (RF, rf). Search Networking. Online. https://searchnetworking.techtarget.com/definition/radio-frequency Updated February 2021. Read 22.07.2021.

4 All about circuits. Practical Guide to Radio-Frequency Analysis and Design. EETech Meadia, LLC. Online. https://www.allaboutcircuits.com/textbook/radio-frequency-analysis-design/real-life-rf-signals/what-is-a-transmission-line/ Read 22.07.2021.

5 Sorrentino, Roberto; Bianchi, Giovanni & Chang, Kai. 2010. Microwave and RF Engineering. John Wiley & Sons, Incorporated 2010, p. 72 & 816.

6 Lampen, Steve. 2012. 50 Ohms The Forgotten Impedance. Belden. Online. https://www.belden.com/blogs/broadcast/50-ohms-the-forgotten-impedance/ 27.08.2012. Read 23.07.2021.

7 Techplayon. 2017. Why characteristics impedance of RF transmission lines is kept 50 Ohms? Techplayon. Online. https://www.techplayon.com/characteristics-impedance-rf-transmission-lines-kept-50-ohms/ 30.08.2017. Read 23.07.2021.

8 Customcable.ca. 2012. Flexible vs. Semi Rigid vs. Rigid RF (Coax) Cable Assemblies. Customcable.ca. Online. https://customcable.ca/flexible-semi-rigid-rf-coax-cable-assemblies/ 13.01.2012. Read 23.07.2021.

9 Rohde Schwarz. 2019. Understanding S parameters. Video. https://www.youtube.com/watch?v=-Pi0UbErHTY 18.10.2019. Accessed 26.07.2021.

10 Bluefors Oy. Coaxial Wiring. Online. https://bluefors.com/products/coaxial-wiring/ Read 16.01.2022.

11 Zhidkov, Roman. 2020. Why Python is Essential for Data Analysis. RT Insights. Online. https://www.rtinsights.com/why-python-is-essential-for-data-analysis/ 13.01.2020. Read 27.07.2021.

12 Nishad, Rohit. 2021. What Is Anaconda? Anaconda Vs python Programming (2021). 360 Tech Explorer. Online. https://360techexplorer.com/what-is-anaconda-anaconda-vs-python-programming/ 05.03.2021. Read 29.07.2021.

13 Perkel, Jeffrey M. 2018. Why Jupyter is data scientists' computational notebook of choice? Nature. Online https://www.nature.com/articles/d41586-018-07196-1 30.10.2018. Read 15.08.2021.

14 QuantAcademy. 2020. QCoDeS – Intro and why to use it. Video. https://www.youtube.com/watch?v=5r4vBAsN6hY 20.05.2020. Accessed 29.07.2021.

15 Noble Desktop. 2018. What Is Git & Why Should You Use It? Noble Desktop. Online. https://www.nobledesktop.com/blog/what-is-git-and-why-should-you-use-it 21.09.2018. Read 21.08.2021.

Appendices

Appendix 1. Flowchart of the Old Style

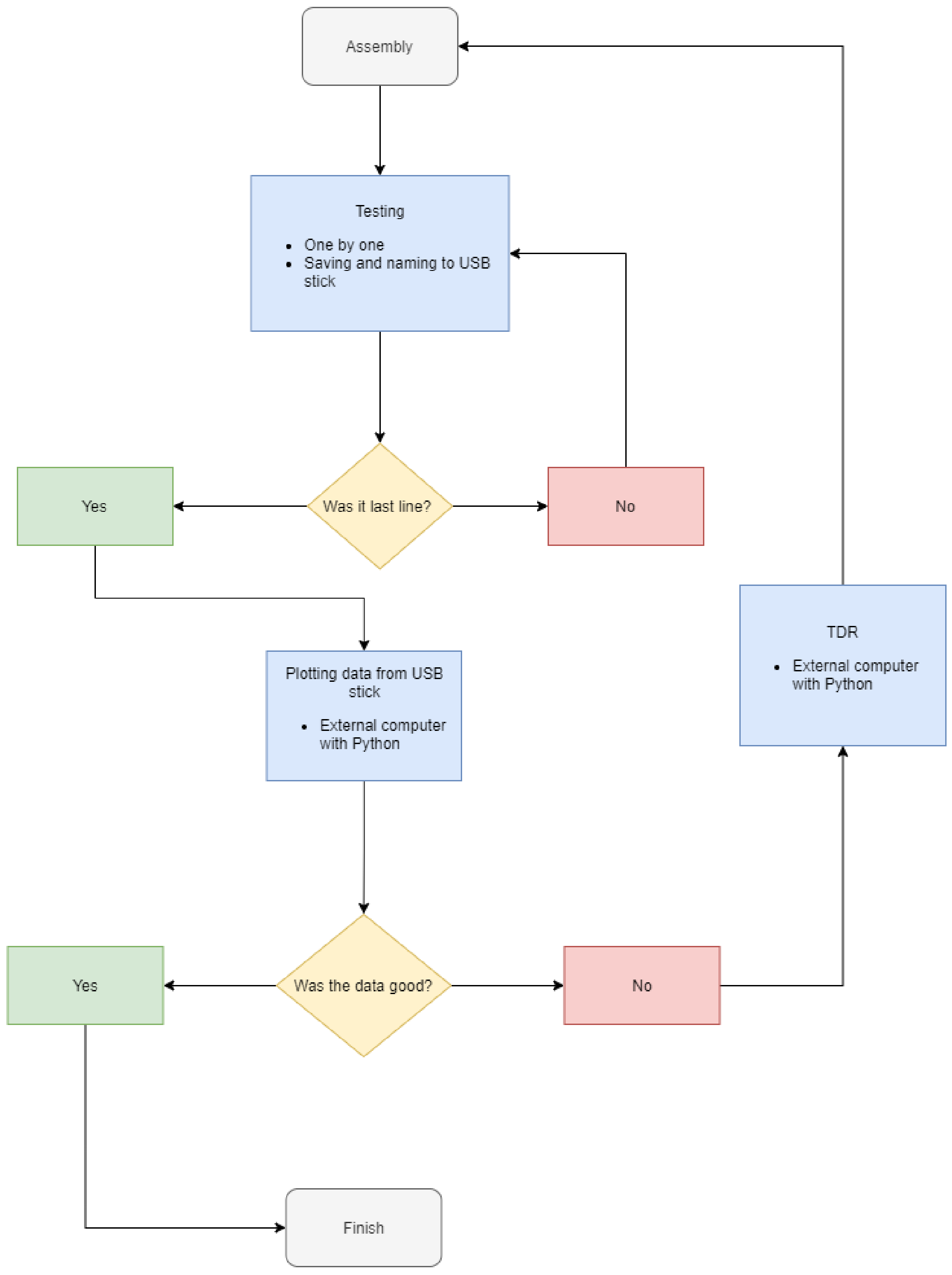

Assembly
Testing
One by one
Saving and naming to USB stick
Was it last line?
Yes
No
Plotting data from USB stick
External computer with Python
TDR
External computer with Python
Was the data good?
Yes
No
Finish

Figure 28. Flowchart showing how testing was performed before this project.

Appendix 2. Flowchart of the New Style

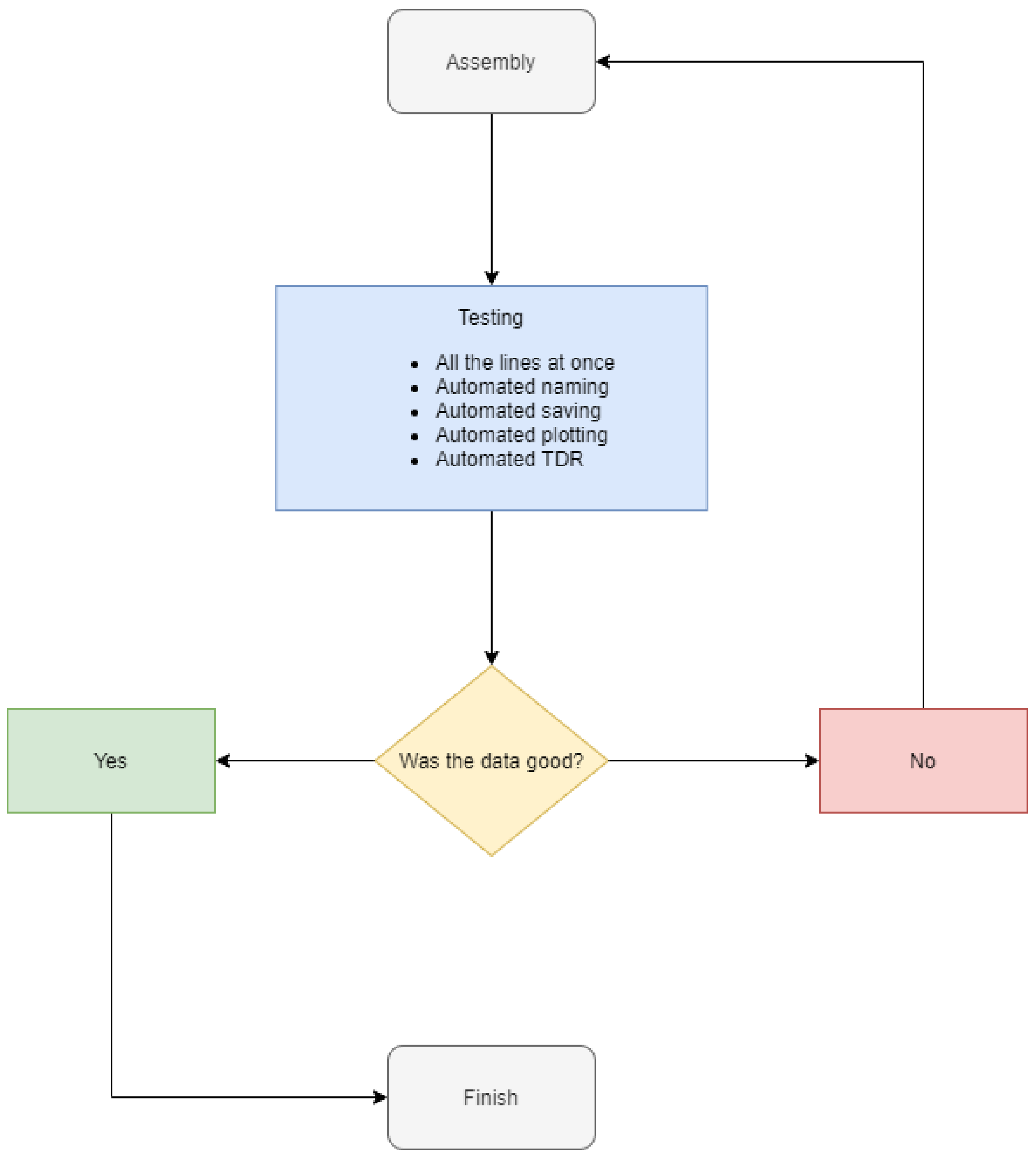


Figure 29. Flowchart showing how testing was performed after this project.

Appendix 3. Time Estimation

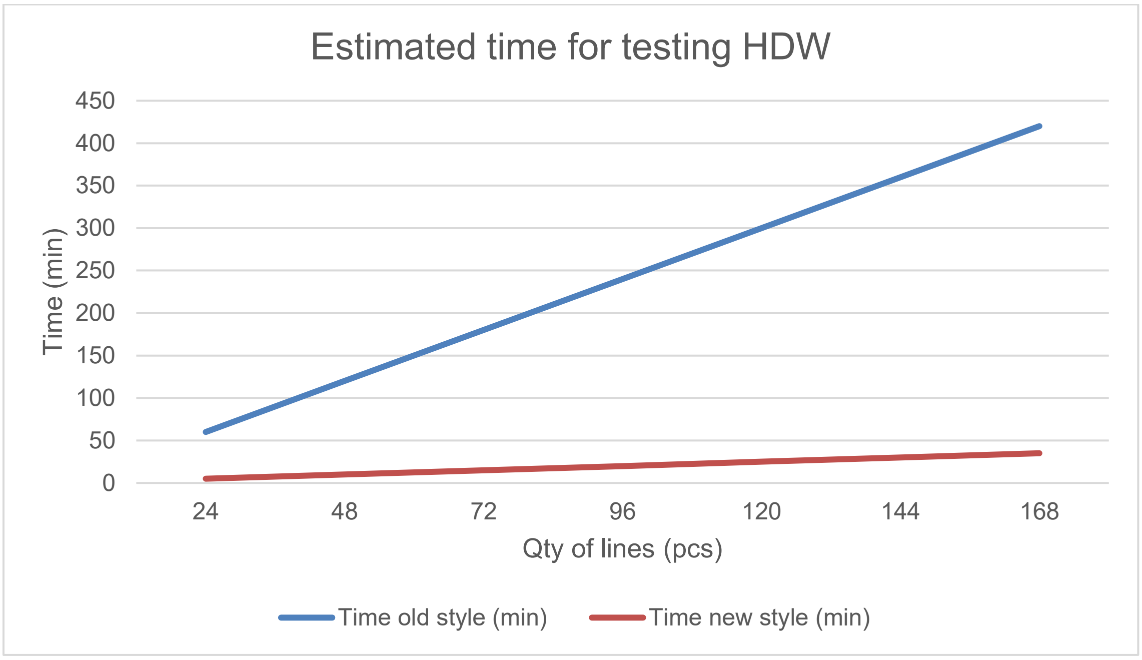
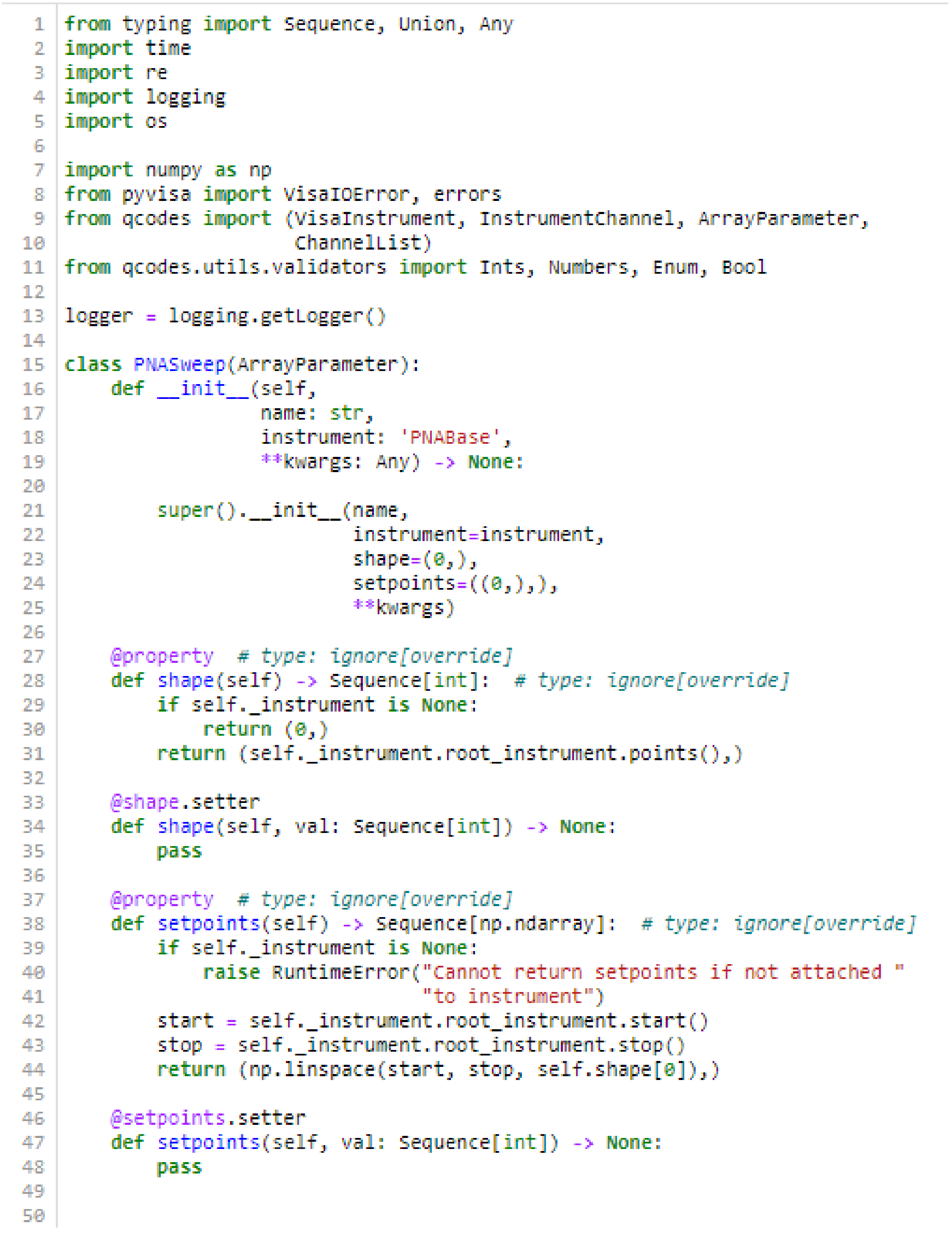


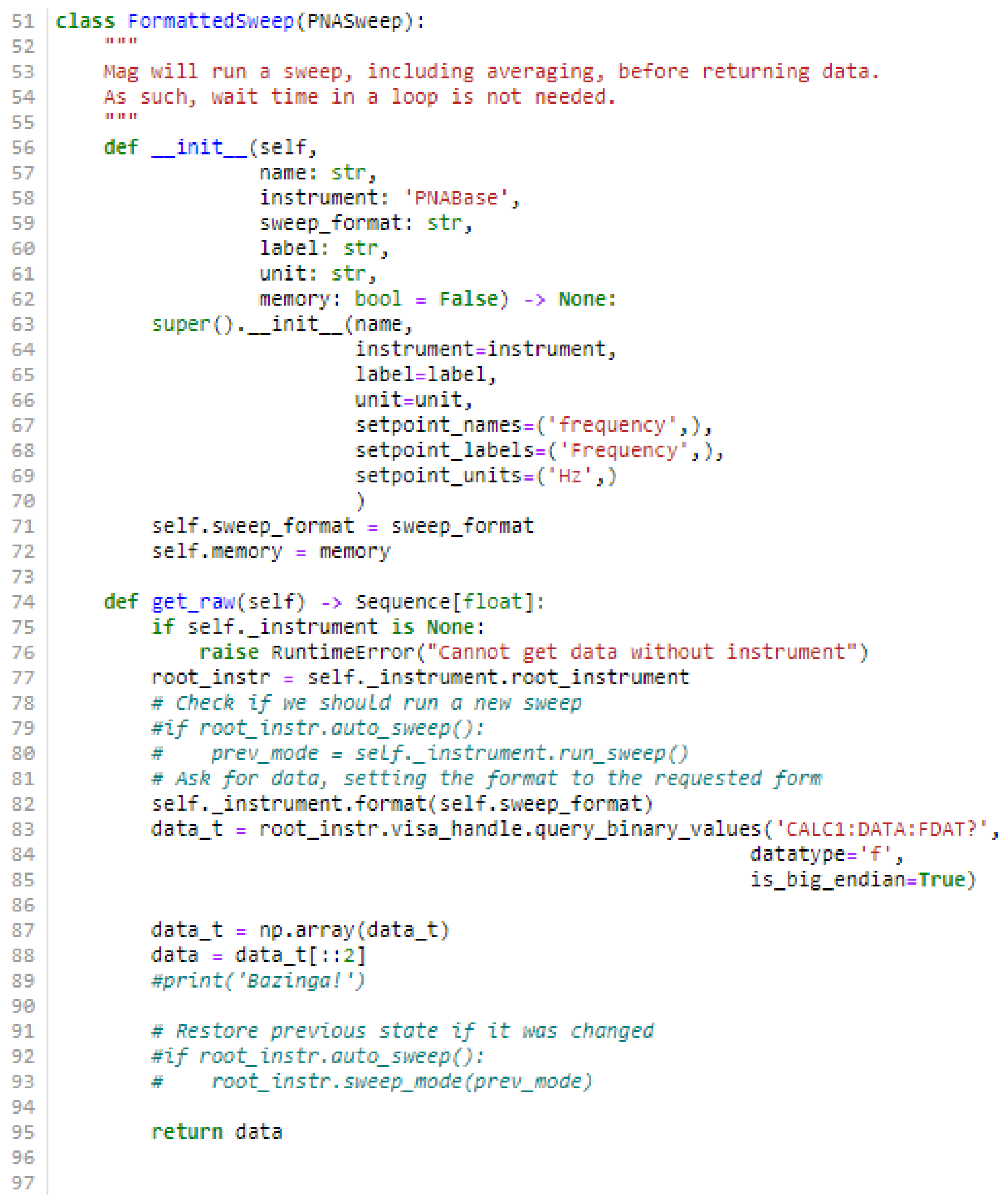
Figure 30. Estimated time for testing 1 full port of HDW.

Appendix 4. N52xx_modified



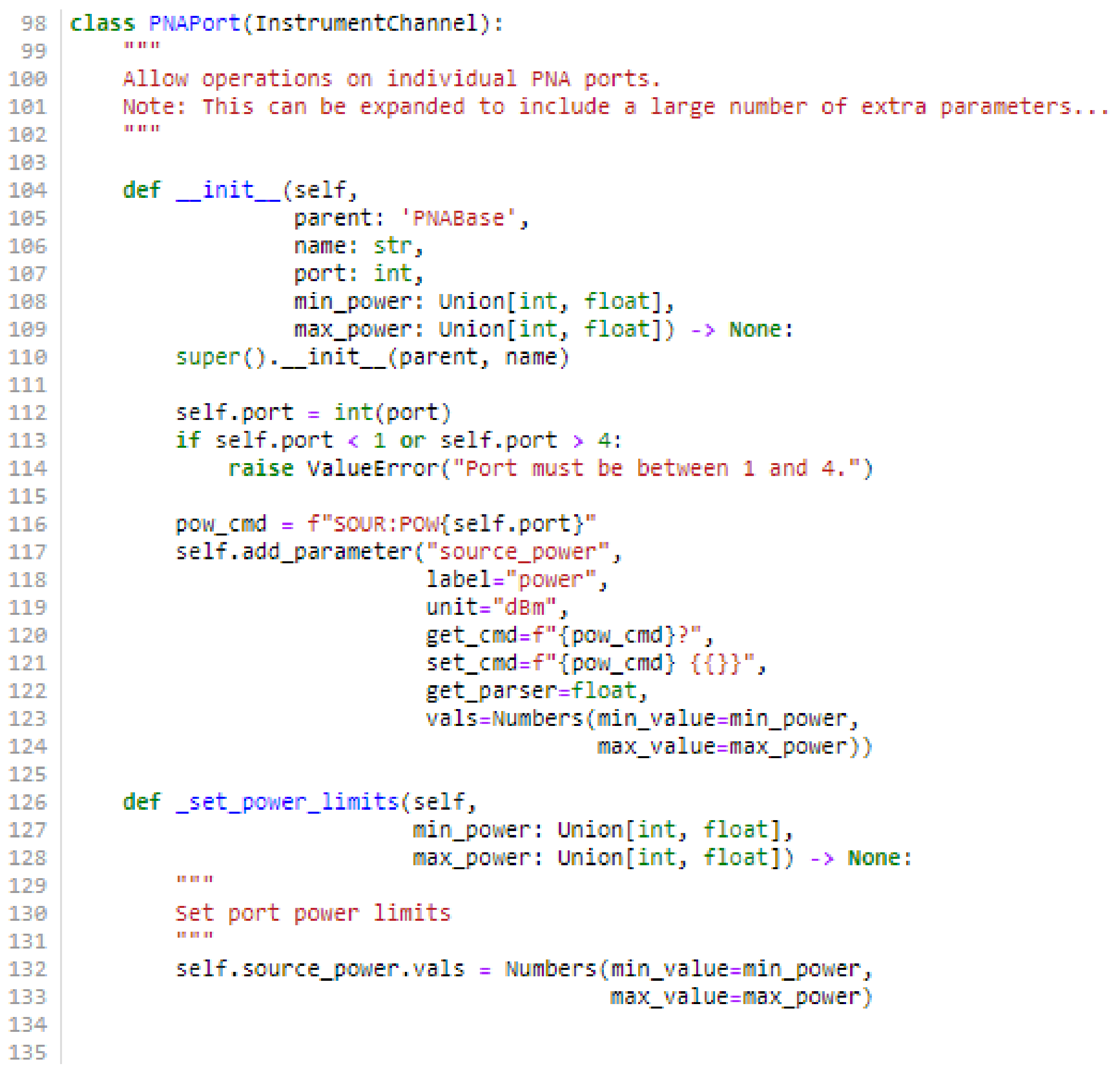
```python
from typing import Sequence, Union, Any
import time
import re
import logging
import os

import numpy as np
from pyvisa import VisaIOError, errors
from qcodes import (VisaInstrument, InstrumentChannel, ArrayParameter,
                    ChannelList)
from qcodes.utils.validators import Ints, Numbers, Enum, Bool

logger = logging.getLogger()

class PNASweep(ArrayParameter):
    def __init__(self,
                 name: str,
                 instrument: 'PNABase',
                 **kwargs: Any) -> None:

        super().__init__(name,
                         instrument=instrument,
                         shape=(0,),
                         setpoints=((0,),),
                         **kwargs)

    @property  # type: ignore[override]
    def shape(self) -> Sequence[int]:  # type: ignore[override]
        if self._instrument is None:
            return (0,)
        return (self._instrument.root_instrument.points(),)

    @shape.setter
    def shape(self, val: Sequence[int]) -> None:
        pass

    @property  # type: ignore[override]
    def setpoints(self) -> Sequence[np.ndarray]:  # type: ignore[override]
        if self._instrument is None:
            raise RuntimeError("Cannot return setpoints if not attached "
                               "to instrument")
        start = self._instrument.root_instrument.start()
        stop = self._instrument.root_instrument.stop()
        return (np.linspace(start, stop, self.shape[0]),)

    @setpoints.setter
    def setpoints(self, val: Sequence[int]) -> None:
        pass


```

Figure 31. N52xx_modified rows 1-50.



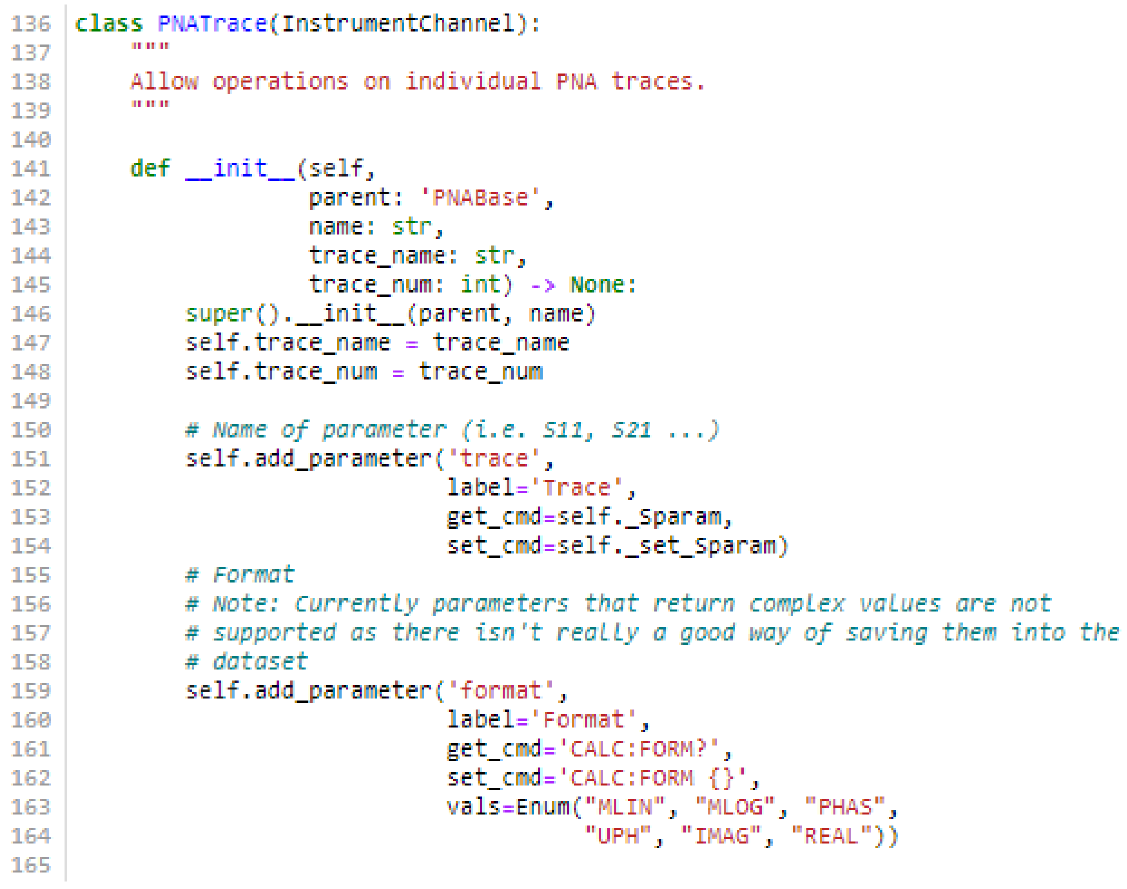
```
51  class FormattedSweep(PNASweep):
52      """
53      Mag will run a sweep, including averaging, before returning data.
54      As such, wait time in a loop is not needed.
55      """
56      def __init__(self,
57                   name: str,
58                   instrument: 'PNABase',
59                   sweep_format: str,
60                   label: str,
61                   unit: str,
62                   memory: bool = False) -> None:
63          super().__init__(name,
64                           instrument=instrument,
65                           label=label,
66                           unit=unit,
67                           setpoint_names=('frequency',),
68                           setpoint_labels=('Frequency',),
69                           setpoint_units=('Hz',)
70                           )
71          self.sweep_format = sweep_format
72          self.memory = memory
73  
74      def get_raw(self) -> Sequence[float]:
75          if self._instrument is None:
76              raise RuntimeError("Cannot get data without instrument")
77          root_instr = self._instrument.root_instrument
78          # Check if we should run a new sweep
79          #if root_instr.auto_sweep():
80          #    prev_mode = self._instrument.run_sweep()
81          # Ask for data, setting the format to the requested form
82          self._instrument.format(self.sweep_format)
83          data_t = root_instr.visa_handle.query_binary_values('CALC1:DATA:FDAT?',
84                                                            datatype='f',
85                                                            is_big_endian=True)
86  
87          data_t = np.array(data_t)
88          data = data_t[::2]
89          #print('Bazinga!')
90  
91          # Restore previous state if it was changed
92          #if root_instr.auto_sweep():
93          #    root_instr.sweep_mode(prev_mode)
94  
95          return data
96  
97  
```

Figure 32. N52xx_modified rows 51-97.



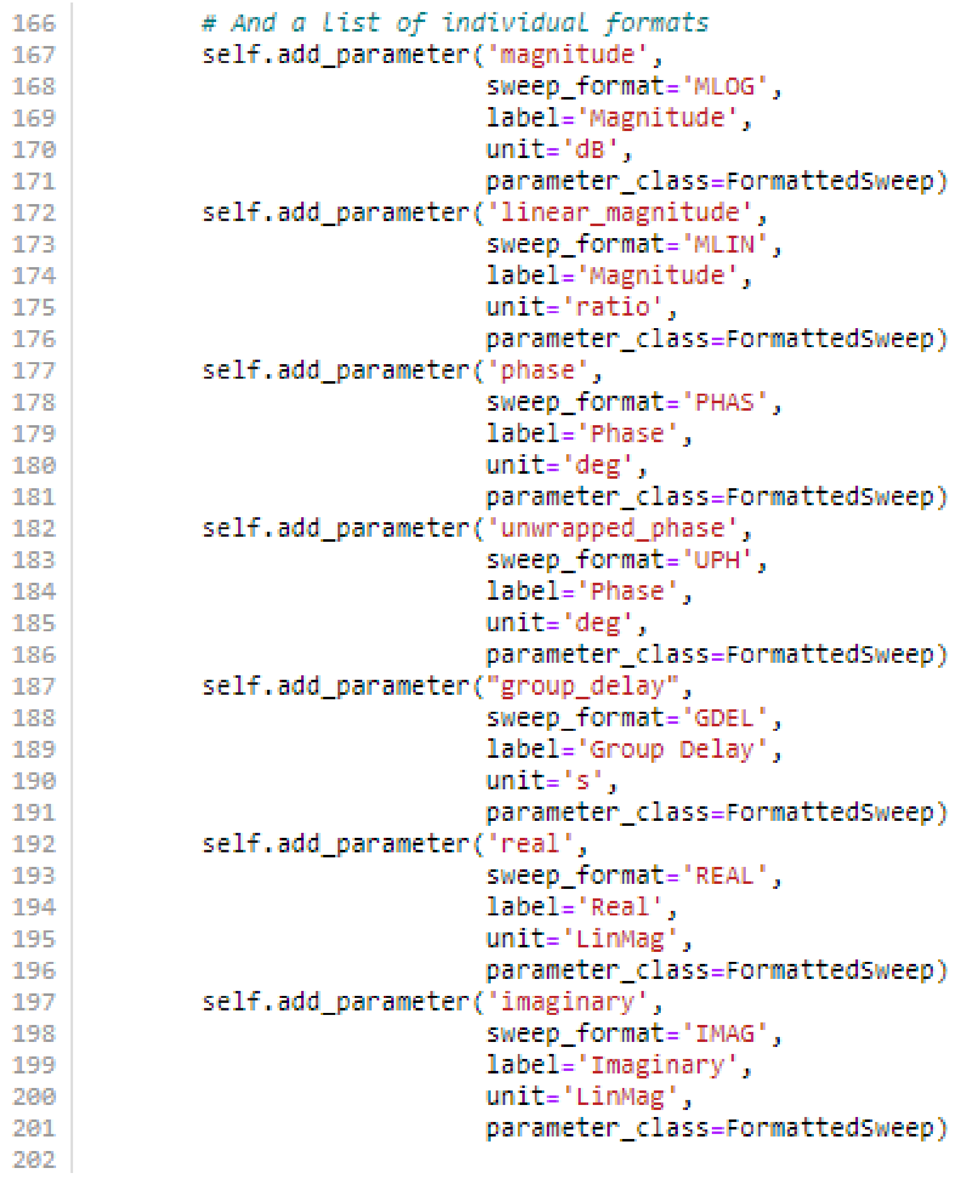
```
98  class PNAPort(InstrumentChannel):
99      """
100     Allow operations on individual PNA ports.
101     Note: This can be expanded to include a large number of extra parameters...
102     """
103
104     def __init__(self,
105                  parent: 'PNABase',
106                  name: str,
107                  port: int,
108                  min_power: Union[int, float],
109                  max_power: Union[int, float]) -> None:
110         super().__init__(parent, name)
111
112         self.port = int(port)
113         if self.port < 1 or self.port > 4:
114             raise ValueError("Port must be between 1 and 4.")
115
116         pow_cmd = f"SOUR:POW{self.port}"
117         self.add_parameter("source_power",
118                            label="power",
119                            unit="dBm",
120                            get_cmd=f"{pow_cmd}?",
121                            set_cmd=f"{pow_cmd} {{}}",
122                            get_parser=float,
123                            vals=Numbers(min_value=min_power,
124                                         max_value=max_power))
125
126     def _set_power_limits(self,
127                           min_power: Union[int, float],
128                           max_power: Union[int, float]) -> None:
129         """
130         Set port power limits
131         """
132         self.source_power.vals = Numbers(min_value=min_power,
133                                          max_value=max_power)
134
135
```

Figure 33. N52xx_modified rows 98-135.



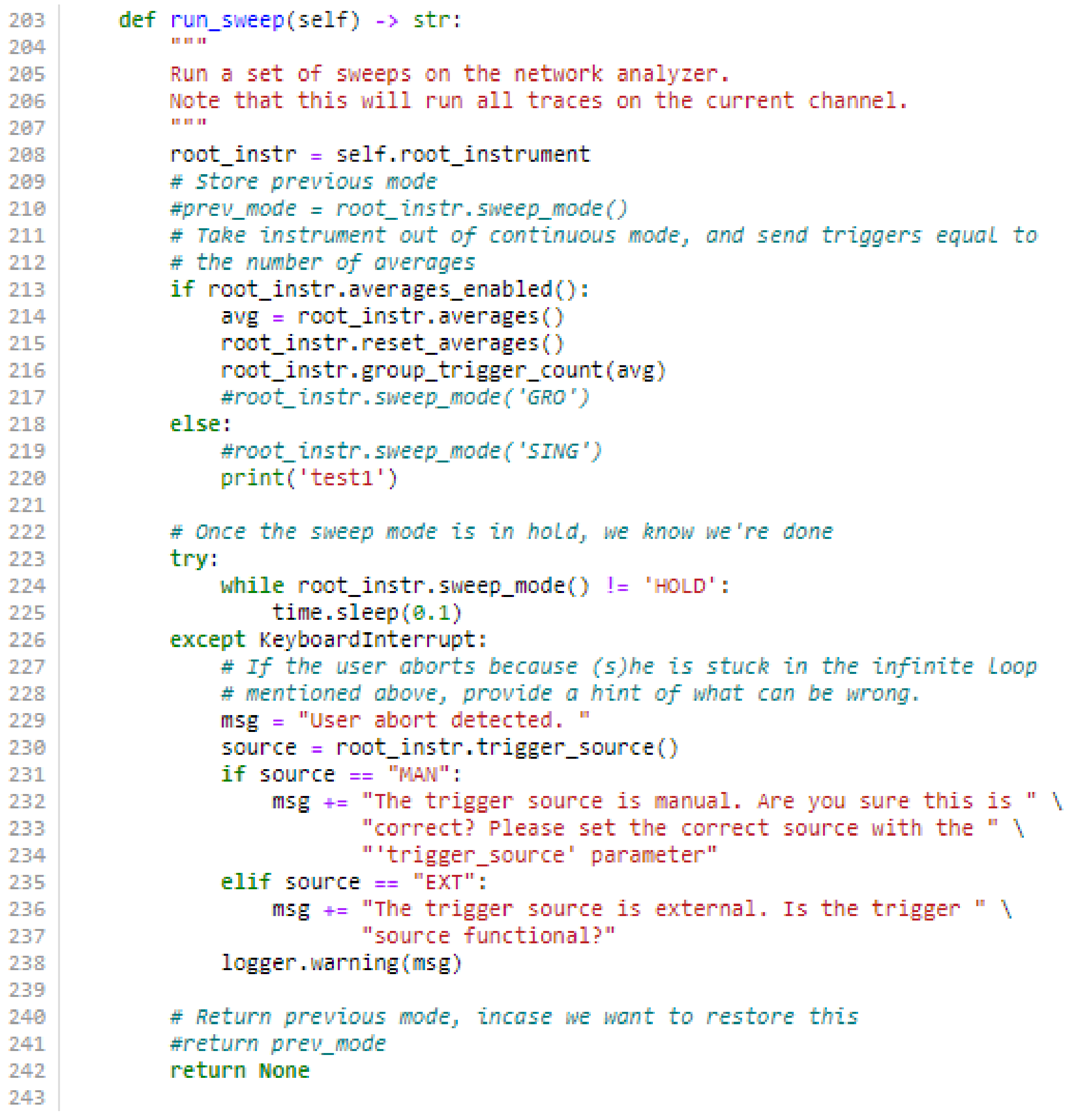
```python
class PNATrace(InstrumentChannel):
    """
    Allow operations on individual PNA traces.
    """

    def __init__(self,
                 parent: 'PNABase',
                 name: str,
                 trace_name: str,
                 trace_num: int) -> None:
        super().__init__(parent, name)
        self.trace_name = trace_name
        self.trace_num = trace_num

        # Name of parameter (i.e. S11, S21 ...)
        self.add_parameter('trace',
                           label='Trace',
                           get_cmd=self._Sparam,
                           set_cmd=self._set_Sparam)
        # Format
        # Note: Currently parameters that return complex values are not
        # supported as there isn't really a good way of saving them into the
        # dataset
        self.add_parameter('format',
                           label='Format',
                           get_cmd='CALC:FORM?',
                           set_cmd='CALC:FORM {}',
                           vals=Enum("MLIN", "MLOG", "PHAS",
                                     "UPH", "IMAG", "REAL"))

```

Figure 34. N52xx_modified rows 136-165.



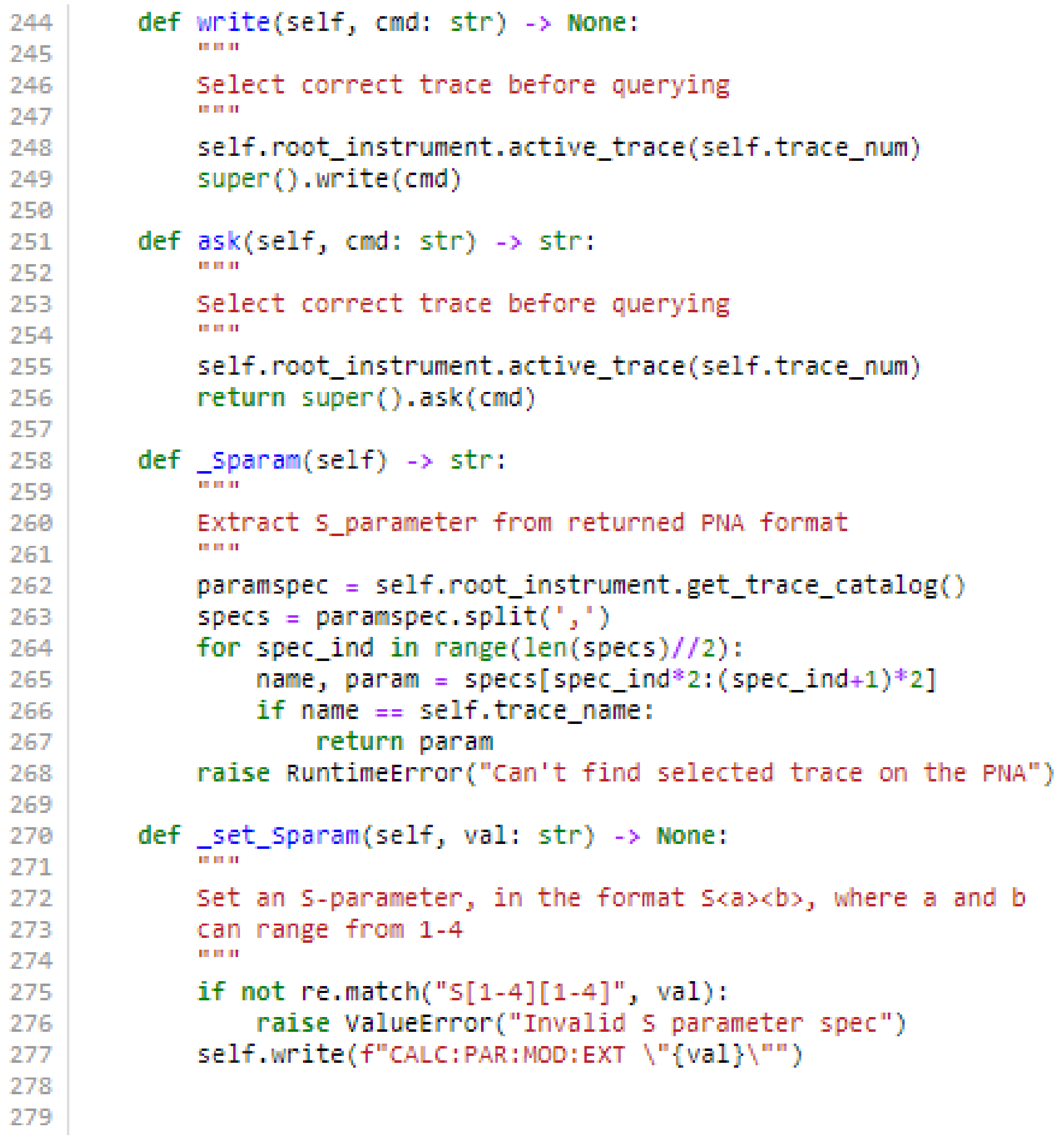
```
166         # And a list of individual formats
167         self.add_parameter('magnitude',
168                            sweep_format='MLOG',
169                            label='Magnitude',
170                            unit='dB',
171                            parameter_class=FormattedSweep)
172         self.add_parameter('linear_magnitude',
173                            sweep_format='MLIN',
174                            label='Magnitude',
175                            unit='ratio',
176                            parameter_class=FormattedSweep)
177         self.add_parameter('phase',
178                            sweep_format='PHAS',
179                            label='Phase',
180                            unit='deg',
181                            parameter_class=FormattedSweep)
182         self.add_parameter('unwrapped_phase',
183                            sweep_format='UPH',
184                            label='Phase',
185                            unit='deg',
186                            parameter_class=FormattedSweep)
187         self.add_parameter("group_delay",
188                            sweep_format='GDEL',
189                            label='Group Delay',
190                            unit='s',
191                            parameter_class=FormattedSweep)
192         self.add_parameter('real',
193                            sweep_format='REAL',
194                            label='Real',
195                            unit='LinMag',
196                            parameter_class=FormattedSweep)
197         self.add_parameter('imaginary',
198                            sweep_format='IMAG',
199                            label='Imaginary',
200                            unit='LinMag',
201                            parameter_class=FormattedSweep)
202
```

Figure 35. N52xx_modified rows 166-202.



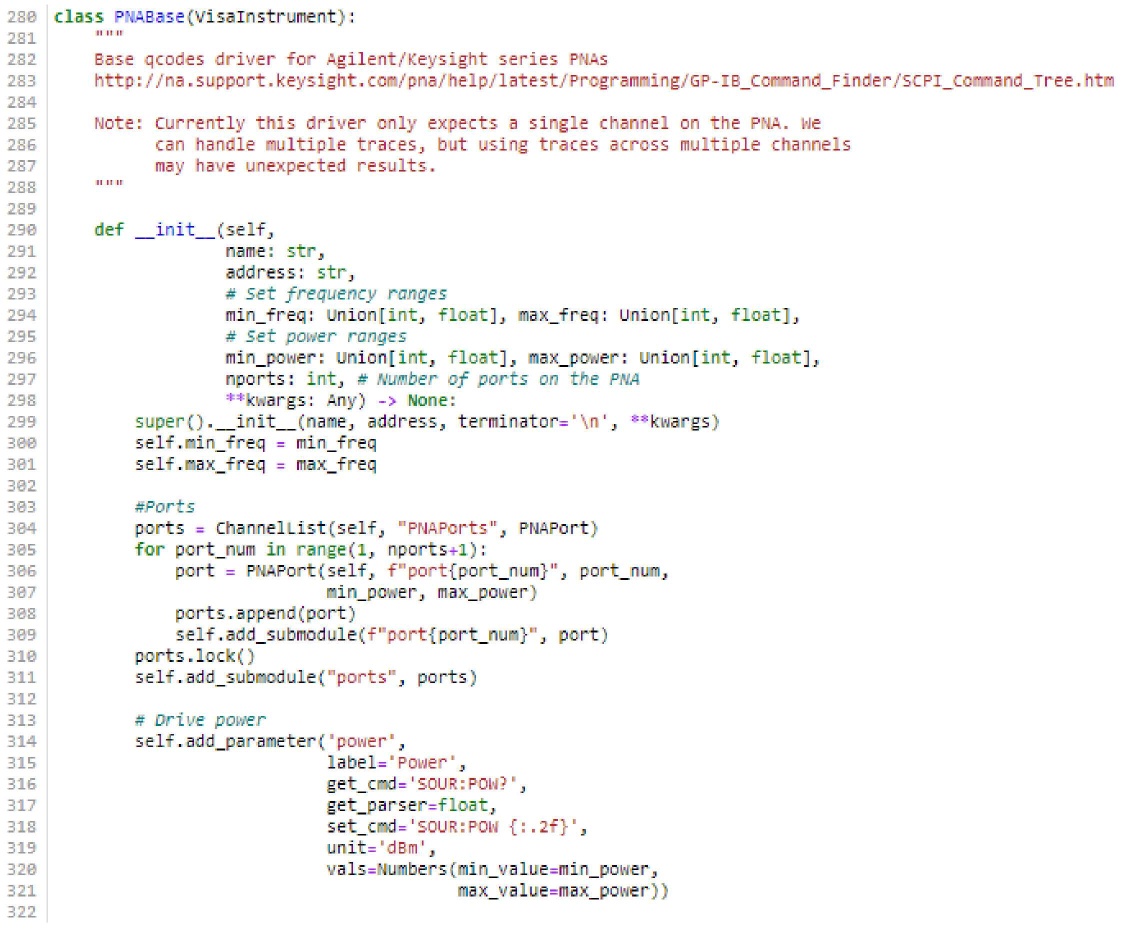
```
203     def run_sweep(self) -> str:
204         """
205         Run a set of sweeps on the network analyzer.
206         Note that this will run all traces on the current channel.
207         """
208         root_instr = self.root_instrument
209         # Store previous mode
210         #prev_mode = root_instr.sweep_mode()
211         # Take instrument out of continuous mode, and send triggers equal to
212         # the number of averages
213         if root_instr.averages_enabled():
214             avg = root_instr.averages()
215             root_instr.reset_averages()
216             root_instr.group_trigger_count(avg)
217             #root_instr.sweep_mode('GRO')
218         else:
219             #root_instr.sweep_mode('SING')
220             print('test1')
221 
222         # Once the sweep mode is in hold, we know we're done
223         try:
224             while root_instr.sweep_mode() != 'HOLD':
225                 time.sleep(0.1)
226         except KeyboardInterrupt:
227             # If the user aborts because (s)he is stuck in the infinite loop
228             # mentioned above, provide a hint of what can be wrong.
229             msg = "User abort detected. "
230             source = root_instr.trigger_source()
231             if source == "MAN":
232                 msg += "The trigger source is manual. Are you sure this is " \
233                        "correct? Please set the correct source with the " \
234                        "'trigger_source' parameter"
235             elif source == "EXT":
236                 msg += "The trigger source is external. Is the trigger " \
237                        "source functional?"
238             logger.warning(msg)
239 
240         # Return previous mode, incase we want to restore this
241         #return prev_mode
242         return None
243 
```

Figure 36. N52xx_modified rows 203-243.



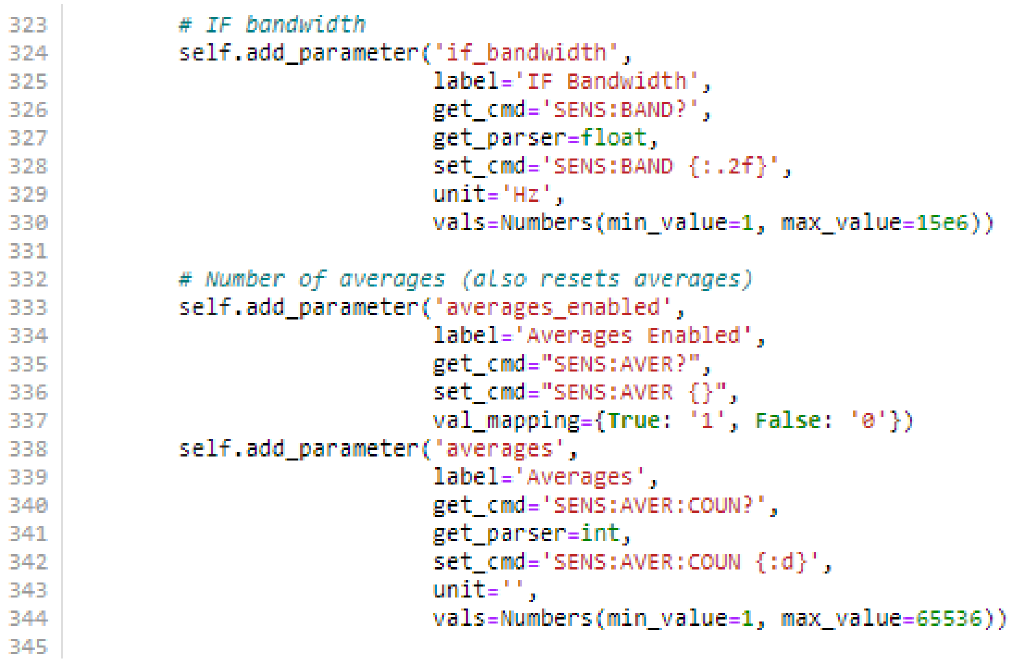
```python
    def write(self, cmd: str) -> None:
        """
        Select correct trace before querying
        """
        self.root_instrument.active_trace(self.trace_num)
        super().write(cmd)

    def ask(self, cmd: str) -> str:
        """
        Select correct trace before querying
        """
        self.root_instrument.active_trace(self.trace_num)
        return super().ask(cmd)

    def _Sparam(self) -> str:
        """
        Extract S_parameter from returned PNA format
        """
        paramspec = self.root_instrument.get_trace_catalog()
        specs = paramspec.split(',')
        for spec_ind in range(len(specs)//2):
            name, param = specs[spec_ind*2:(spec_ind+1)*2]
            if name == self.trace_name:
                return param
        raise RuntimeError("Can't find selected trace on the PNA")

    def _set_Sparam(self, val: str) -> None:
        """
        Set an S-parameter, in the format S<a><b>, where a and b
        can range from 1-4
        """
        if not re.match("S[1-4][1-4]", val):
            raise ValueError("Invalid S parameter spec")
        self.write(f"CALC:PAR:MOD:EXT \"{val}\"")
```

Figure 37. N52xx_modified rows 244-279.



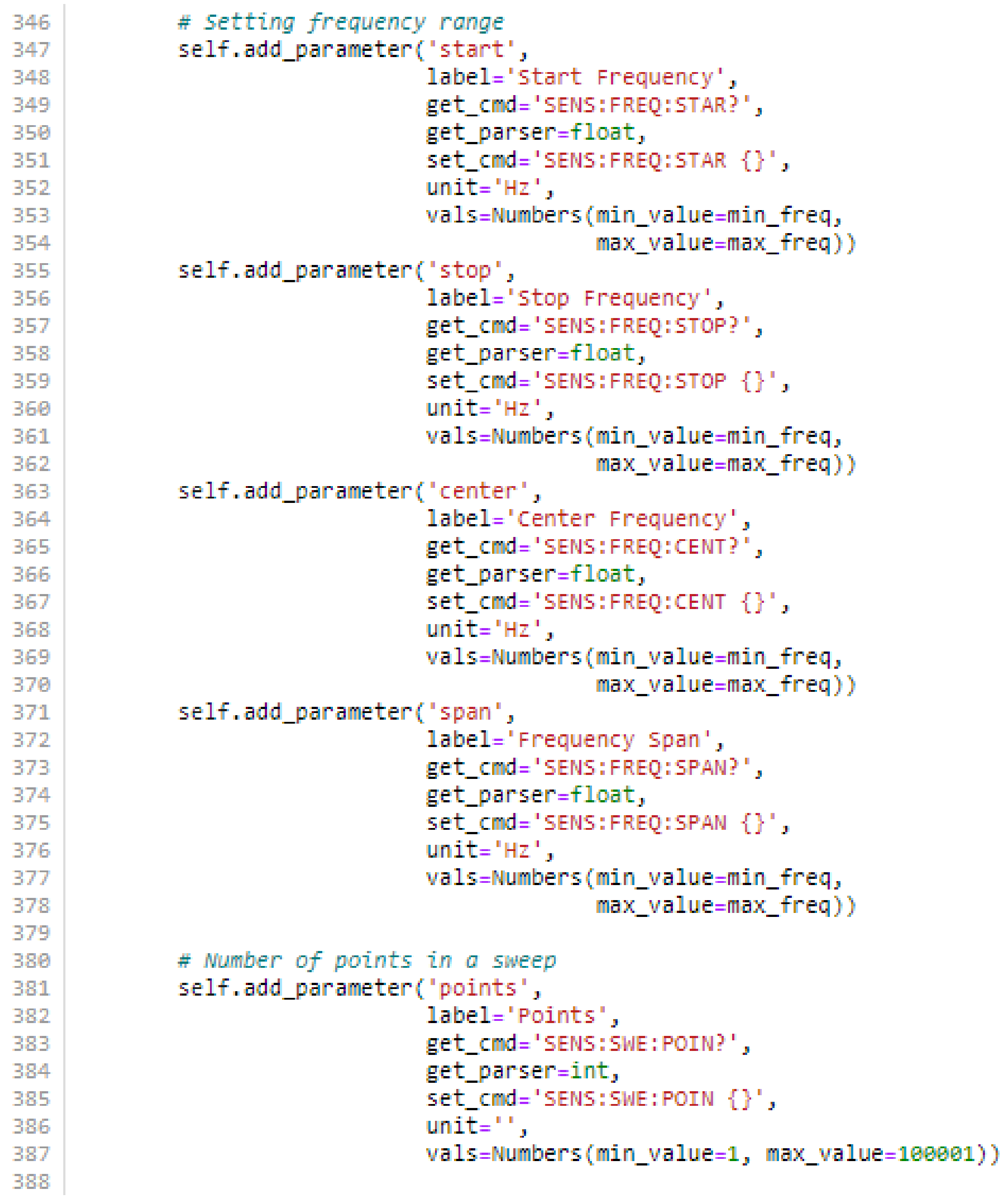
```
class PNABase(VisaInstrument):
    """
    Base qcodes driver for Agilent/Keysight series PNAs
    http://na.support.keysight.com/pna/help/latest/Programming/GP-IB_Command_Finder/SCPI_Command_Tree.htm

    Note: Currently this driver only expects a single channel on the PNA. We
          can handle multiple traces, but using traces across multiple channels
          may have unexpected results.
    """

    def __init__(self,
                 name: str,
                 address: str,
                 # Set frequency ranges
                 min_freq: Union[int, float], max_freq: Union[int, float],
                 # Set power ranges
                 min_power: Union[int, float], max_power: Union[int, float],
                 nports: int, # Number of ports on the PNA
                 **kwargs: Any) -> None:
        super().__init__(name, address, terminator='\n', **kwargs)
        self.min_freq = min_freq
        self.max_freq = max_freq

        #Ports
        ports = ChannelList(self, "PNAPorts", PNAPort)
        for port_num in range(1, nports+1):
            port = PNAPort(self, f"port{port_num}", port_num,
                           min_power, max_power)
            ports.append(port)
            self.add_submodule(f"port{port_num}", port)
        ports.lock()
        self.add_submodule("ports", ports)

        # Drive power
        self.add_parameter('power',
                           label='Power',
                           get_cmd='SOUR:POW?',
                           get_parser=float,
                           set_cmd='SOUR:POW {:.2f}',
                           unit='dBm',
                           vals=Numbers(min_value=min_power,
                                        max_value=max_power))

```

Figure 38. N52xx_modified rows 280-322.



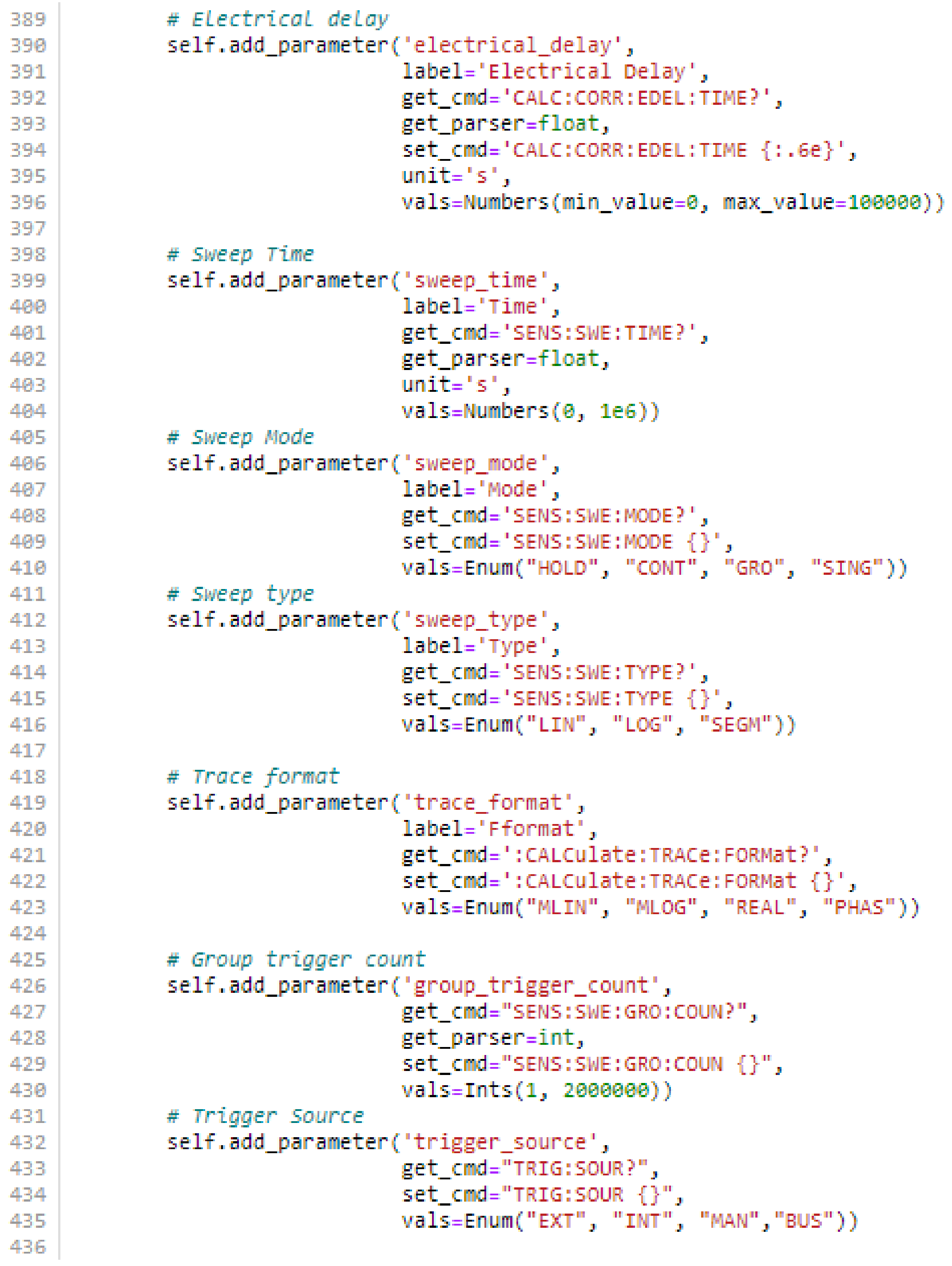
```
        # IF bandwidth
        self.add_parameter('if_bandwidth',
                           label='IF Bandwidth',
                           get_cmd='SENS:BAND?',
                           get_parser=float,
                           set_cmd='SENS:BAND {:.2f}',
                           unit='Hz',
                           vals=Numbers(min_value=1, max_value=15e6))

        # Number of averages (also resets averages)
        self.add_parameter('averages_enabled',
                           label='Averages Enabled',
                           get_cmd="SENS:AVER?",
                           set_cmd="SENS:AVER {}",
                           val_mapping={True: '1', False: '0'})
        self.add_parameter('averages',
                           label='Averages',
                           get_cmd='SENS:AVER:COUN?',
                           get_parser=int,
                           set_cmd='SENS:AVER:COUN {:d}',
                           unit='',
                           vals=Numbers(min_value=1, max_value=65536))

```

Figure 39. N52xx_modified rows 323-345.



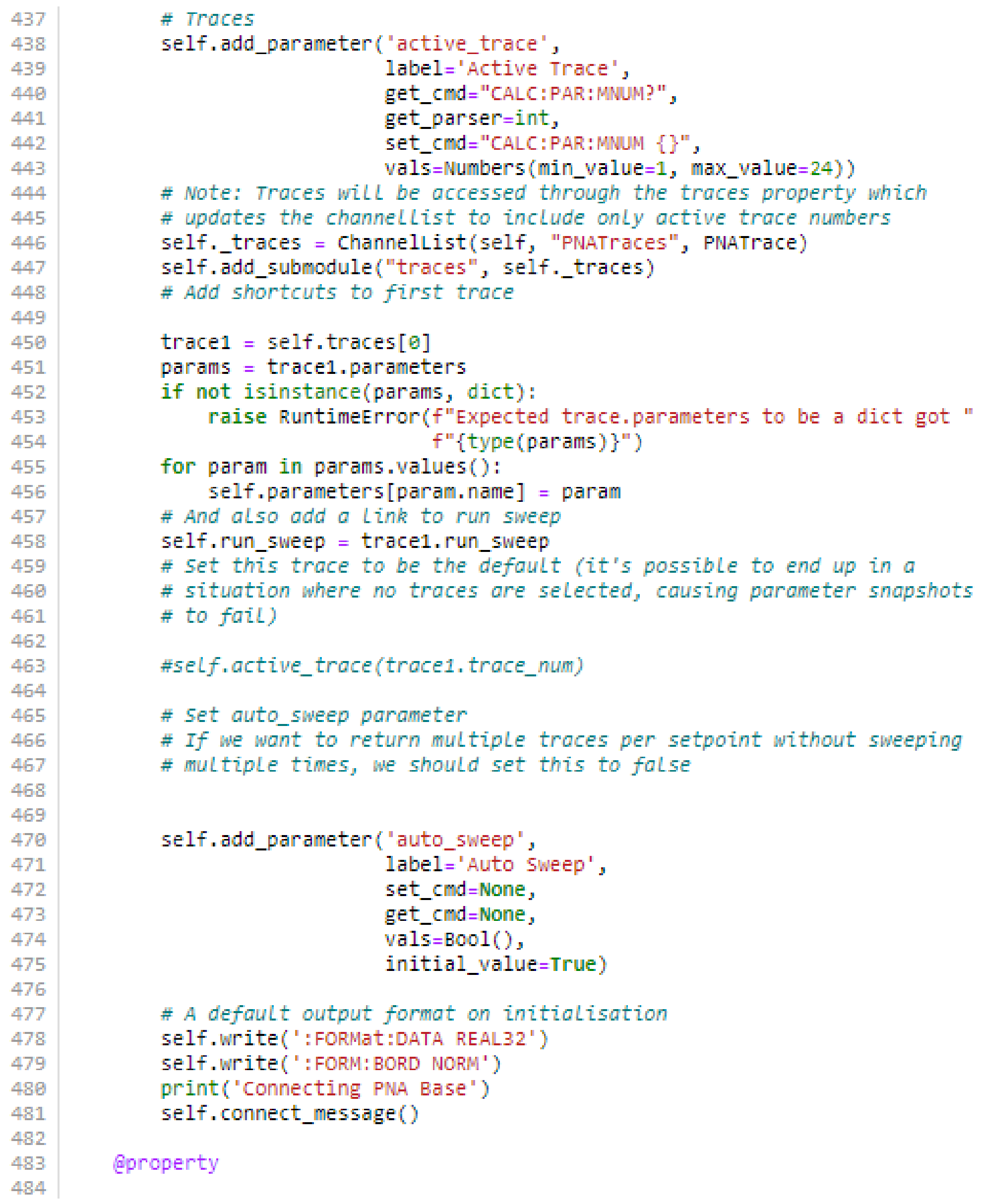
```
        # Setting frequency range
        self.add_parameter('start',
                           label='Start Frequency',
                           get_cmd='SENS:FREQ:STAR?',
                           get_parser=float,
                           set_cmd='SENS:FREQ:STAR {}',
                           unit='Hz',
                           vals=Numbers(min_value=min_freq,
                                        max_value=max_freq))
        self.add_parameter('stop',
                           label='Stop Frequency',
                           get_cmd='SENS:FREQ:STOP?',
                           get_parser=float,
                           set_cmd='SENS:FREQ:STOP {}',
                           unit='Hz',
                           vals=Numbers(min_value=min_freq,
                                        max_value=max_freq))
        self.add_parameter('center',
                           label='Center Frequency',
                           get_cmd='SENS:FREQ:CENT?',
                           get_parser=float,
                           set_cmd='SENS:FREQ:CENT {}',
                           unit='Hz',
                           vals=Numbers(min_value=min_freq,
                                        max_value=max_freq))
        self.add_parameter('span',
                           label='Frequency Span',
                           get_cmd='SENS:FREQ:SPAN?',
                           get_parser=float,
                           set_cmd='SENS:FREQ:SPAN {}',
                           unit='Hz',
                           vals=Numbers(min_value=min_freq,
                                        max_value=max_freq))

        # Number of points in a sweep
        self.add_parameter('points',
                           label='Points',
                           get_cmd='SENS:SWE:POIN?',
                           get_parser=int,
                           set_cmd='SENS:SWE:POIN {}',
                           unit='',
                           vals=Numbers(min_value=1, max_value=100001))

```

Figure 40. N52xx_modified rows 346-388.



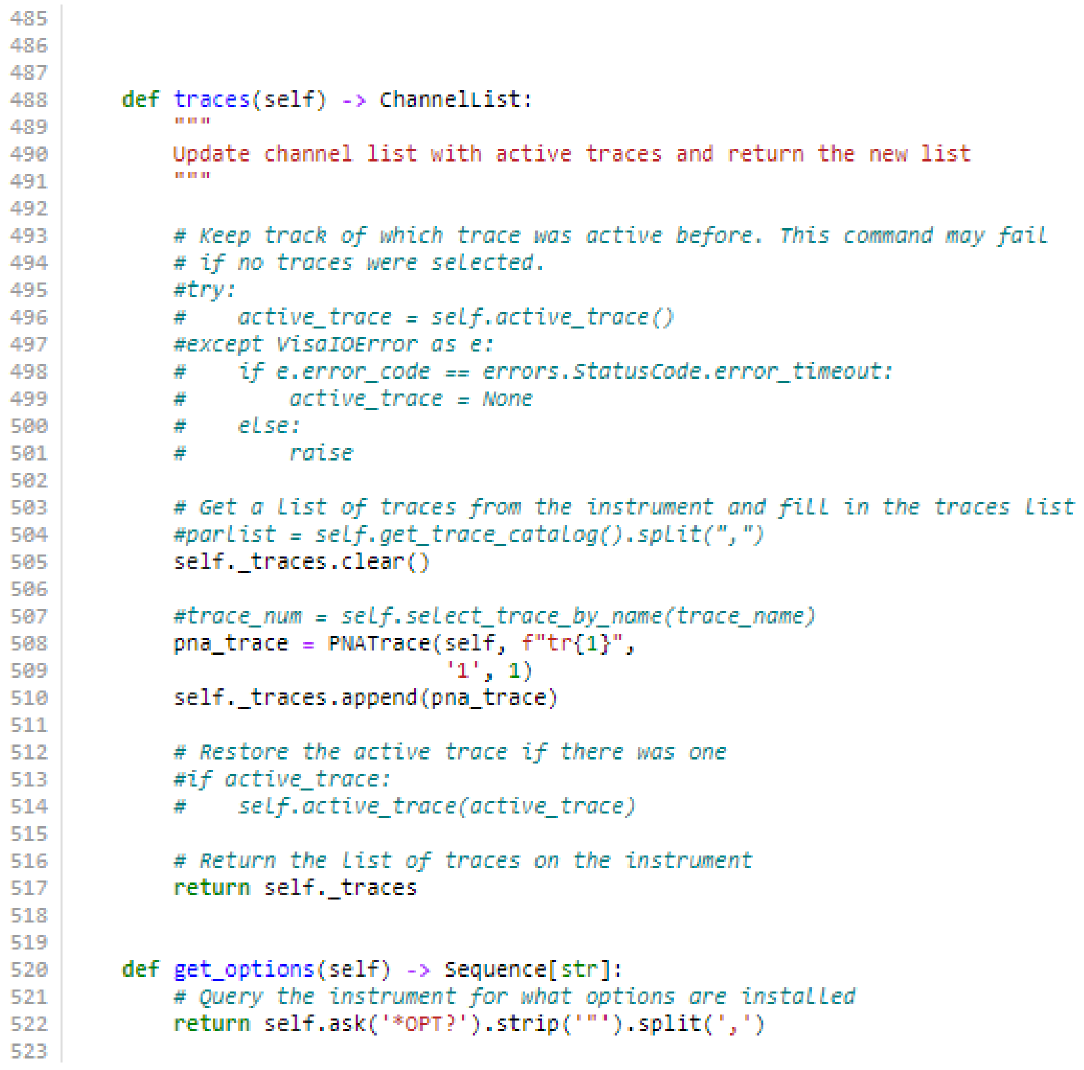
```
389         # Electrical delay
390         self.add_parameter('electrical_delay',
391                            label='Electrical Delay',
392                            get_cmd='CALC:CORR:EDEL:TIME?',
393                            get_parser=float,
394                            set_cmd='CALC:CORR:EDEL:TIME {:.6e}',
395                            unit='s',
396                            vals=Numbers(min_value=0, max_value=100000))
397
398         # Sweep Time
399         self.add_parameter('sweep_time',
400                            label='Time',
401                            get_cmd='SENS:SWE:TIME?',
402                            get_parser=float,
403                            unit='s',
404                            vals=Numbers(0, 1e6))
405         # Sweep Mode
406         self.add_parameter('sweep_mode',
407                            label='Mode',
408                            get_cmd='SENS:SWE:MODE?',
409                            set_cmd='SENS:SWE:MODE {}',
410                            vals=Enum("HOLD", "CONT", "GRO", "SING"))
411         # Sweep type
412         self.add_parameter('sweep_type',
413                            label='Type',
414                            get_cmd='SENS:SWE:TYPE?',
415                            set_cmd='SENS:SWE:TYPE {}',
416                            vals=Enum("LIN", "LOG", "SEGM"))
417
418         # Trace format
419         self.add_parameter('trace_format',
420                            label='Fformat',
421                            get_cmd=':CALCulate:TRACe:FORMat?',
422                            set_cmd=':CALCulate:TRACe:FORMat {}',
423                            vals=Enum("MLIN", "MLOG", "REAL", "PHAS"))
424
425         # Group trigger count
426         self.add_parameter('group_trigger_count',
427                            get_cmd="SENS:SWE:GRO:COUN?",
428                            get_parser=int,
429                            set_cmd="SENS:SWE:GRO:COUN {}",
430                            vals=Ints(1, 2000000))
431         # Trigger Source
432         self.add_parameter('trigger_source',
433                            get_cmd="TRIG:SOUR?",
434                            set_cmd="TRIG:SOUR {}",
435                            vals=Enum("EXT", "INT", "MAN","BUS"))
436
```

Figure 41. N52xx_modified rows 389-436.



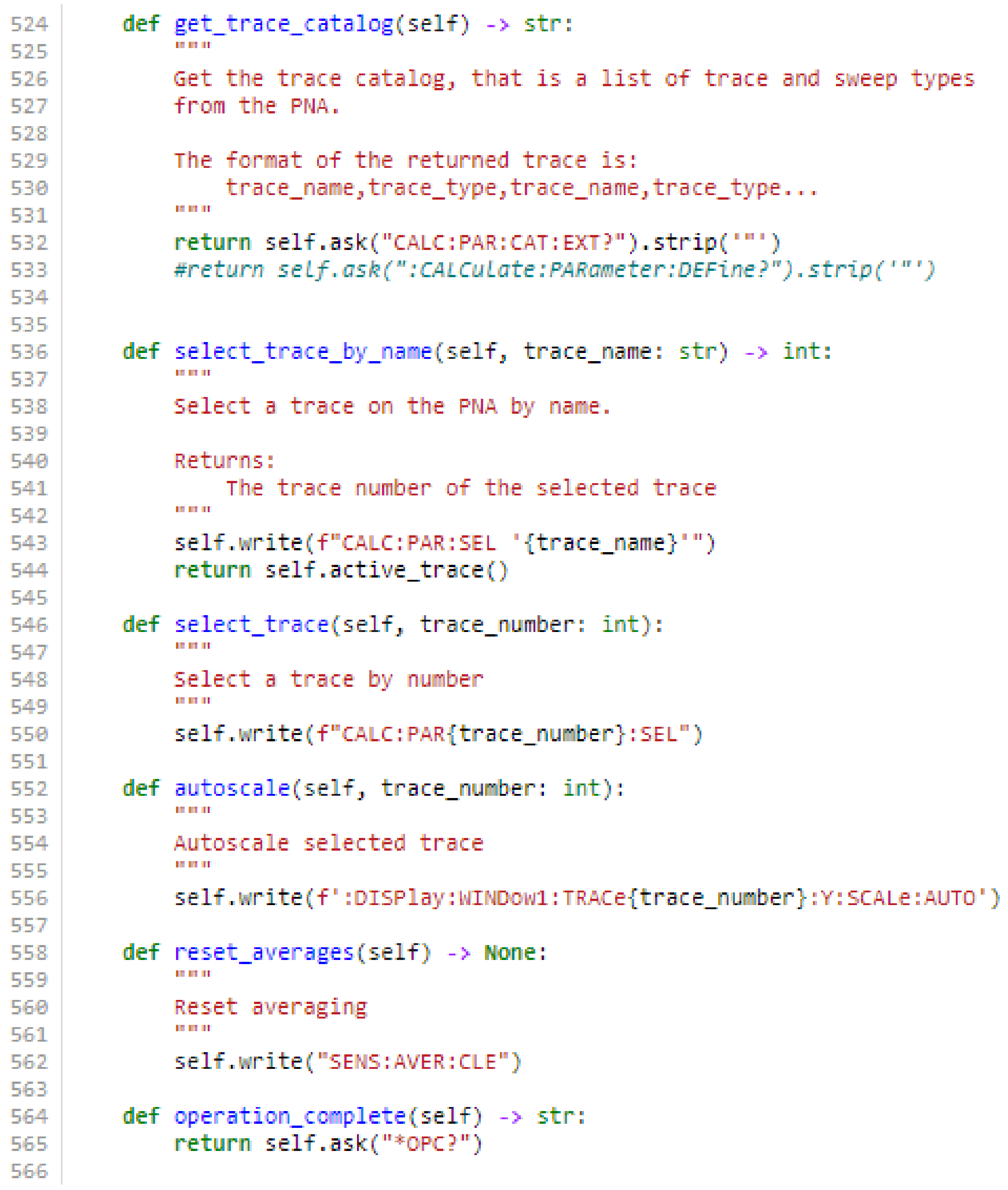
```
        # Traces
        self.add_parameter('active_trace',
                           label='Active Trace',
                           get_cmd="CALC:PAR:MNUM?",
                           get_parser=int,
                           set_cmd="CALC:PAR:MNUM {}",
                           vals=Numbers(min_value=1, max_value=24))
        # Note: Traces will be accessed through the traces property which
        # updates the channellist to include only active trace numbers
        self._traces = ChannelList(self, "PNATraces", PNATrace)
        self.add_submodule("traces", self._traces)
        # Add shortcuts to first trace

        trace1 = self.traces[0]
        params = trace1.parameters
        if not isinstance(params, dict):
            raise RuntimeError(f"Expected trace.parameters to be a dict got "
                               f"{type(params)}")
        for param in params.values():
            self.parameters[param.name] = param
        # And also add a link to run sweep
        self.run_sweep = trace1.run_sweep
        # Set this trace to be the default (it's possible to end up in a
        # situation where no traces are selected, causing parameter snapshots
        # to fail)

        #self.active_trace(trace1.trace_num)

        # Set auto_sweep parameter
        # If we want to return multiple traces per setpoint without sweeping
        # multiple times, we should set this to false


        self.add_parameter('auto_sweep',
                           label='Auto Sweep',
                           set_cmd=None,
                           get_cmd=None,
                           vals=Bool(),
                           initial_value=True)

        # A default output format on initialisation
        self.write(':FORMat:DATA REAL32')
        self.write(':FORM:BORD NORM')
        print('Connecting PNA Base')
        self.connect_message()

    @property
```

Figure 42. N52xx_modified rows 437-484.



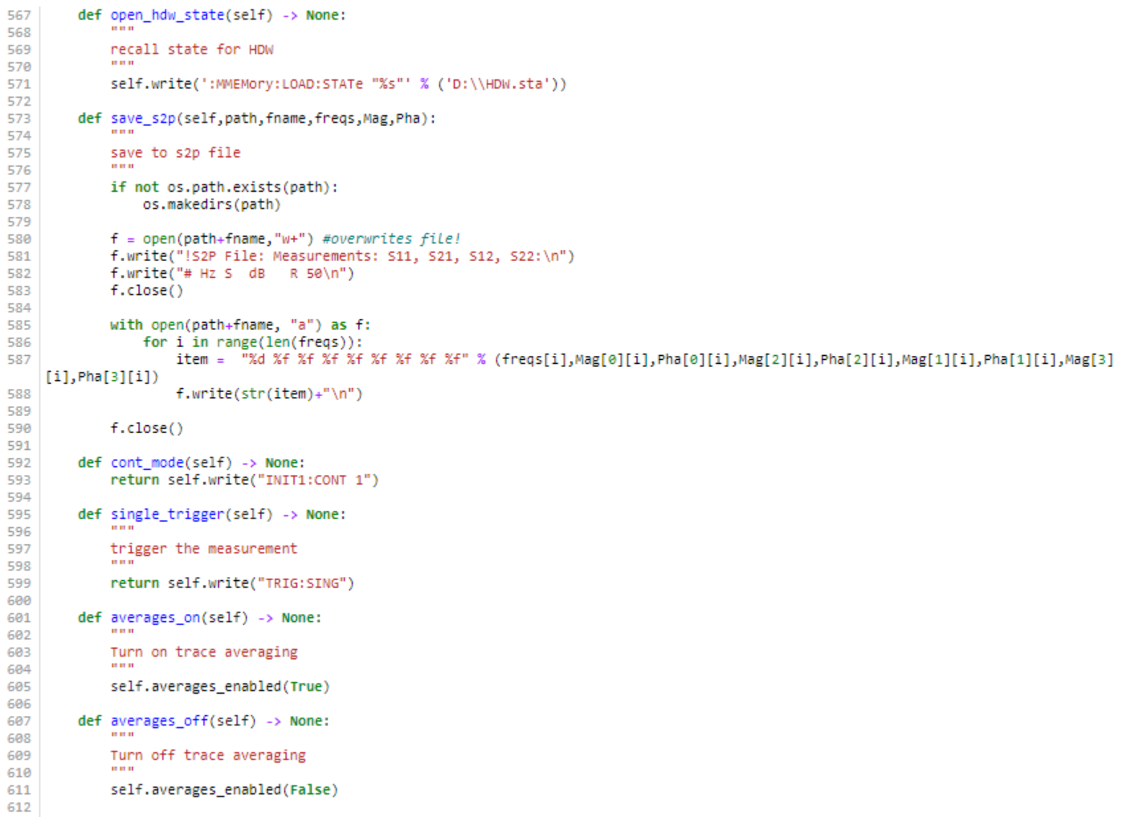
```python


    def traces(self) -> ChannelList:
        """
        Update channel list with active traces and return the new list
        """

        # Keep track of which trace was active before. This command may fail
        # if no traces were selected.
        #try:
        #    active_trace = self.active_trace()
        #except VisaIOError as e:
        #    if e.error_code == errors.StatusCode.error_timeout:
        #        active_trace = None
        #    else:
        #        raise

        # Get a list of traces from the instrument and fill in the traces list
        #parlist = self.get_trace_catalog().split(",")
        self._traces.clear()

        #trace_num = self.select_trace_by_name(trace_name)
        pna_trace = PNATrace(self, f"tr{1}",
                             '1', 1)
        self._traces.append(pna_trace)

        # Restore the active trace if there was one
        #if active_trace:
        #    self.active_trace(active_trace)

        # Return the list of traces on the instrument
        return self._traces


    def get_options(self) -> Sequence[str]:
        # Query the instrument for what options are installed
        return self.ask('*OPT?').strip('"').split(',')

```

Figure 43. N52xx_modified rows 485-523.



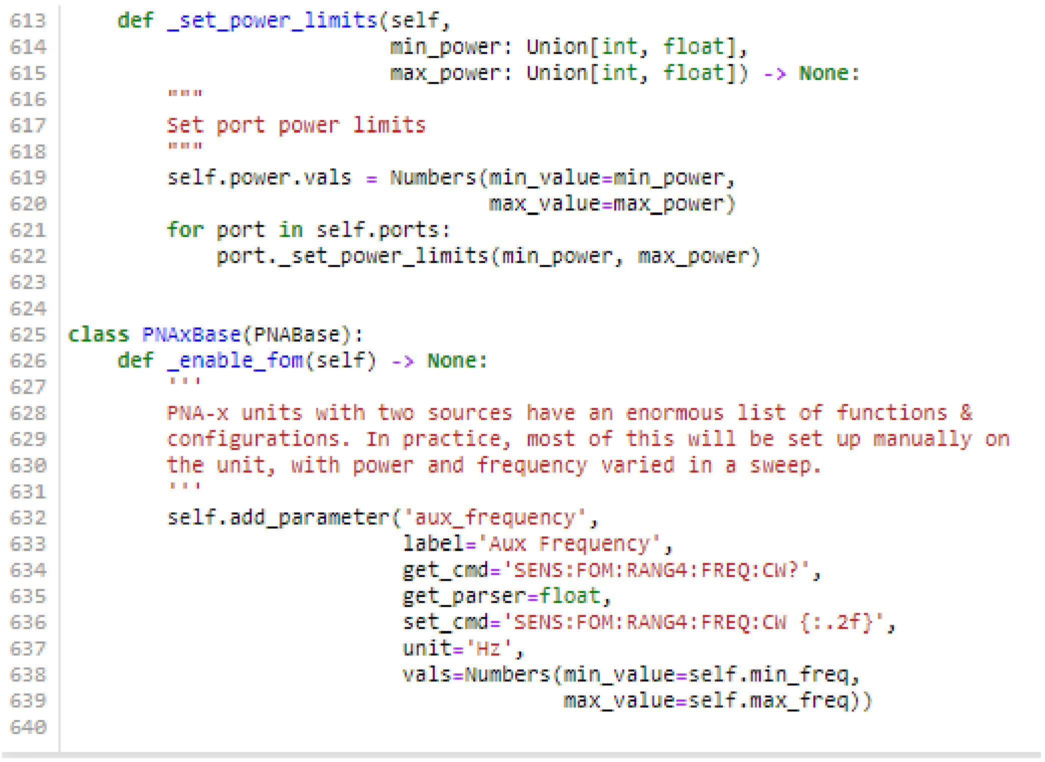
```
524      def get_trace_catalog(self) -> str:
525          """
526          Get the trace catalog, that is a list of trace and sweep types
527          from the PNA.
528
529          The format of the returned trace is:
530              trace_name,trace_type,trace_name,trace_type...
531          """
532          return self.ask("CALC:PAR:CAT:EXT?").strip('"')
533          #return self.ask(":CALCulate:PARameter:DEFine?").strip('"')
534
535
536      def select_trace_by_name(self, trace_name: str) -> int:
537          """
538          Select a trace on the PNA by name.
539
540          Returns:
541              The trace number of the selected trace
542          """
543          self.write(f"CALC:PAR:SEL '{trace_name}'")
544          return self.active_trace()
545
546      def select_trace(self, trace_number: int):
547          """
548          Select a trace by number
549          """
550          self.write(f"CALC:PAR{trace_number}:SEL")
551
552      def autoscale(self, trace_number: int):
553          """
554          Autoscale selected trace
555          """
556          self.write(f':DISPlay:WINDow1:TRACe{trace_number}:Y:SCALe:AUTO')
557
558      def reset_averages(self) -> None:
559          """
560          Reset averaging
561          """
562          self.write("SENS:AVER:CLE")
563
564      def operation_complete(self) -> str:
565          return self.ask("*OPC?")
566
```

Figure 44. N52xx_modified rows 524-566.



```python
    def open_hdw_state(self) -> None:
        """
        recall state for HDW
        """
        self.write(':MMEMory:LOAD:STATe "%s"' % ('D:\\HDW.sta'))

    def save_s2p(self,path,fname,freqs,Mag,Pha):
        """
        save to s2p file
        """
        if not os.path.exists(path):
            os.makedirs(path)

        f = open(path+fname,"w+") #overwrites file!
        f.write("!S2P File: Measurements: S11, S21, S12, S22:\n")
        f.write("# Hz S  dB   R 50\n")
        f.close()

        with open(path+fname, "a") as f:
            for i in range(len(freqs)):
                item =  "%d %f %f %f %f %f %f %f %f" % (freqs[i],Mag[0][i],Pha[0][i],Mag[2][i],Pha[2][i],Mag[1][i],Pha[1][i],Mag[3]
[i],Pha[3][i])
                f.write(str(item)+"\n")

        f.close()

    def cont_mode(self) -> None:
        return self.write("INIT1:CONT 1")

    def single_trigger(self) -> None:
        """
        trigger the measurement
        """
        return self.write("TRIG:SING")

    def averages_on(self) -> None:
        """
        Turn on trace averaging
        """
        self.averages_enabled(True)

    def averages_off(self) -> None:
        """
        Turn off trace averaging
        """
        self.averages_enabled(False)

```

Figure 45. N52xx_modified rows 567-612.



```python
    def _set_power_limits(self,
                          min_power: Union[int, float],
                          max_power: Union[int, float]) -> None:
        """
        Set port power limits
        """
        self.power.vals = Numbers(min_value=min_power,
                                  max_value=max_power)
        for port in self.ports:
            port._set_power_limits(min_power, max_power)


class PNAxBase(PNABase):
    def _enable_fom(self) -> None:
        '''
        PNA-x units with two sources have an enormous list of functions &
        configurations. In practice, most of this will be set up manually on
        the unit, with power and frequency varied in a sweep.
        '''
        self.add_parameter('aux_frequency',
                           label='Aux Frequency',
                           get_cmd='SENS:FOM:RANG4:FREQ:CW?',
                           get_parser=float,
                           set_cmd='SENS:FOM:RANG4:FREQ:CW {:.2f}',
                           unit='Hz',
                           vals=Numbers(min_value=self.min_freq,
                                        max_value=self.max_freq))

```

Figure 46. N52xx_modified rows 613-640.