The aSpect Device Engine Python API

aSpect Systems

Nov 04, 2024

CONTENTS

Python Software API Manual

> **Note**
>
> Preliminary Version - This is WIP

Meanwhile visit: aSpect Systems

Open Local API documentation

CHAPTER

ONE

INTRODUCTION

This section introduces the idSMU hardware and the main concepts of the API.

1.1 The idSMU hardware

1.1.1 idSMU features

The basic concept of the API is the modelling of the various hardware components of idSMU.
In order to better understand this concept, the hardware components of idSMU will be briefly introduced.

Fig. 1.1: idSMU Module

Features

- Single channel high current (1200mA) voltage source (idSMU1)
- 4 channel voltage and current source (idSMU2)
- Voltage and current measurement

- +/- 12V (idSMU1), +/-11V (idSMU2) default output range
- Asymetric Operation -22V to 24V (idSMU1), -16V to 22V (idSMU2)
- External trigger
- Current clamps / Compliance
- Measurement: 16Bit DAC/ADC, 400uV, 200pA resolution, 250kS/s
- Data rate via USB 3 up to 3.2 Gbit/s
- 26 digital 3.3 IOs
- Optional Pattern generator
- 128 MB DDR RAM

As the name suggests, idSMU is a source measurement unit, i.e. a device with which a voltage (or current) can be forced and measured at the same time. Accordingly, there is at least one SMU unit on an idSMU module. This can have one or more channels.

The idSMU hardware also offers other interesting features that make it a mini test system. In the context of the API, only the Digital IO from idSMU should be mentioned here.
Each idSMU module offers 36 digital input/output channels that can also be controlled via the API. Further hardware is integrated on the modules, e.g. a DRAM. If required, all these components can be used to realise applications that go far beyond forcing and measuring voltages and currents. If required, aSpect Systems can develop customised firmware for various fields of application. One example would be the recording of sensor data, the control of image sensors and the streaming of image data.

> **Note**
>
> If you are interested in these possibilities, please contact *aSpect Systems* or visist the website https://www.aspect-sys.com

The focus of this document is the description of the API which in its standard version enables the control of the SMU modules and additionally the digital IOs.

1.1.2 idSMU variants

There are two main variants of idSMU. These differ essentially in the chip used for forcing and measuring voltage or current.
idSMU1 and the new variant, *idSMU3*, have only one 'SMU channel'. However, this channel can supply up to **1200mA**. This type of idSMU can only force voltages, but no current.
For easier differentiation and following the official designation of the chip used, this type idSMU is also called DPS. No distinction is made between idSMU1 and idSMU3, as there are no significant differences in terms of features in programming from the user's point of view. idSMU3 can generate the necessary voltage supply on the module itself, so that the full current range is available here. For more detailed features, please refer to the relevant product documentation.

The idSMU2 type offers four channels with 70mA each. This type of idSMU is also called SMU.
There are a few differences in the voltage range between SMU and DPS. The digital resources do not differ between the two variants. The names DPS and SMU can be found in the API and are used there as synonyms for the variants mentioned above.

1.1.3 idSMU boards

Mb-X1

In order to operate an idSMU module, it is usually connected to a mainboard.
The API also models this part of the hardware so that several modules can be easily addressed and programmed as a group. With the help of such a board, the various resources and connections can be made available to the user. aSpect Systems already offers ready-made solutions for this.

The *The Mb-X1* offers space for a single idSMU module.

Fig. 1.2: idSMU module on a **Mb-X1** board

This board provides connections for the power supply and USB for communication with the module. The (voltage/current) force lines, trigger, as well as sense lines and digital IOs are connected to the connector on the opposite side.
Both idSMU1 and idSMU2 and the latest idSMU version, idSMU3, can be plugged into this board.
For idSMU1, the high current ranges, 500mA and 1200mA, can only be operated in the positive range and only up to 5V. This restriction does not apply to idSMU3.

aSpect Systems also offers a *housing* for this board.

Fig. 1.3: idSMU module on a **Mb-X1** board with housing

Mb-X16

If you want to operate several modules at the same time, aSpect Systems offers the *The Mb-X16* from aSpect Systems. This offers space for up to 16 idSMU modules. This means that up to 64 SMU channels and 544 digital IOs can be operated.

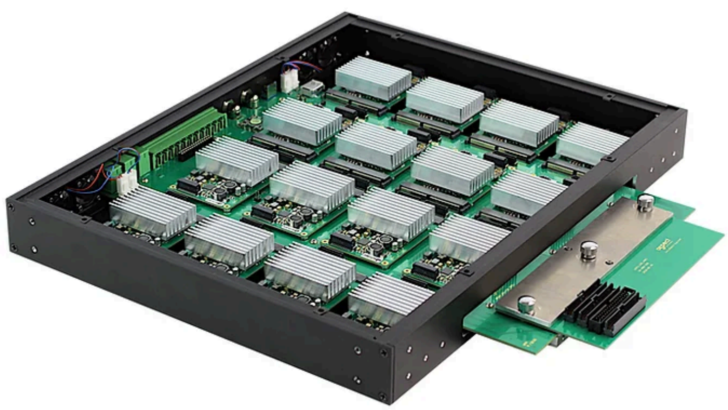


Fig. 1.4: idSMU modules on a **Mb-X16** board

1.2 Api architecture and concepts

1.2.1 The hierarchical hardware model of the API

The core element of the API are the classes that model the idSMU hardware. The part that relates to the measurement and control of current and voltage (SMU) is the most extensive part of the API.
There is also a smaller API part that deals with programming the digital IOs. This part is kept very simple and will be discussed in a later chapter.
Any customised software is not part of the core API and will not be discussed in this document.

The lowest in this hardware model that the user has to deal with are the channels. A large part of the parameterisation of the SMU properties of idSMU is carried out in the channel model. For example, the channel model has the property 'output foce value' or 'clamp enabled'.
The next hierarchical level is the 'Analogue Device Model'. This models the device on idSMU that is responsible for generating and measuring currents and voltages, i.e. the actual SMU (or DPS) chip.
The analogue device model can contain one or more channels, depending on the type of idSMU. In addition to the channels, the device itself also has some programmable parameters. An example of this is an internal resistor that can be switched on if required. Otherwise, the user has very little contact with other properties of the device model.
One level further up you will find the idSMU device model. This contains the analogue device model and the digital IO in the standard API. The idSMU device model is the container that contains the various hardware components of idSMU.
In idSMU3 there is also a power module at this level. This is not shown in the following images for the sake of simplicity.
The board model is located at the top level. There can be one or more idSMU devices on a board.

1.2.2 Brief introduction to the internal mode of operation

The internal functionality of the device engine is briefly presented here. The hardware of idSMU is parameterised internally via registers. This applies to the channel parameters, the analogue device parameters or the digital IOs.
If a user sets a channel parameter using a method such as 'SetVoltage', this is converted into a register value and saved in a register model.
At the same time, this change is communicated to the board model, which also serves as a control unit. Communication with the hardware is command-orientated.
To stay with the example of setting a voltage, a command is assembled that contains the channel to be programmed as an address and the register value that represents this voltage. This command is assembled by the board model and sent to other parts of the device engine, which finally communicate with the hardware via USB.
These register models and the commands normally remain hidden from the user. Instead, they can work with the descriptive methods and properties of the hardware models.

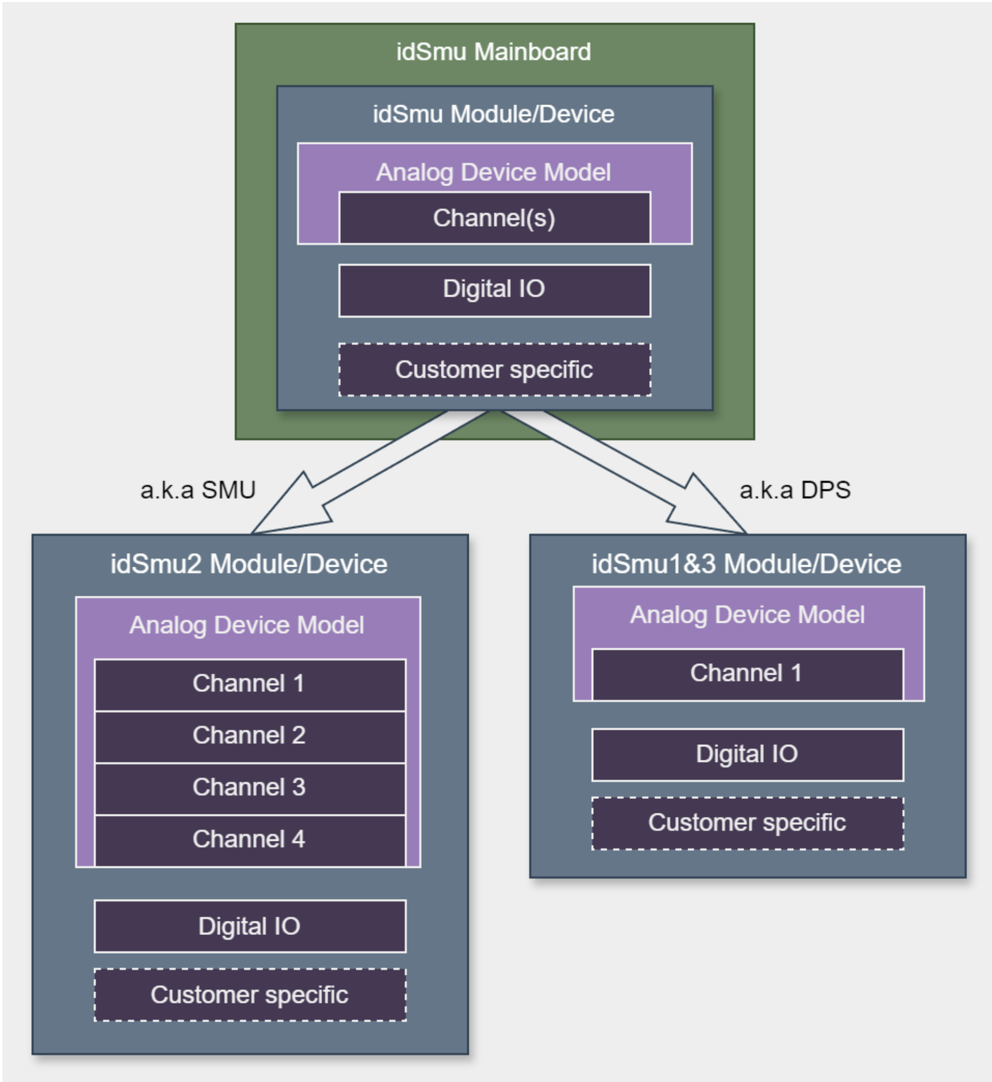


Fig. 1.5: High level view on the API architecture

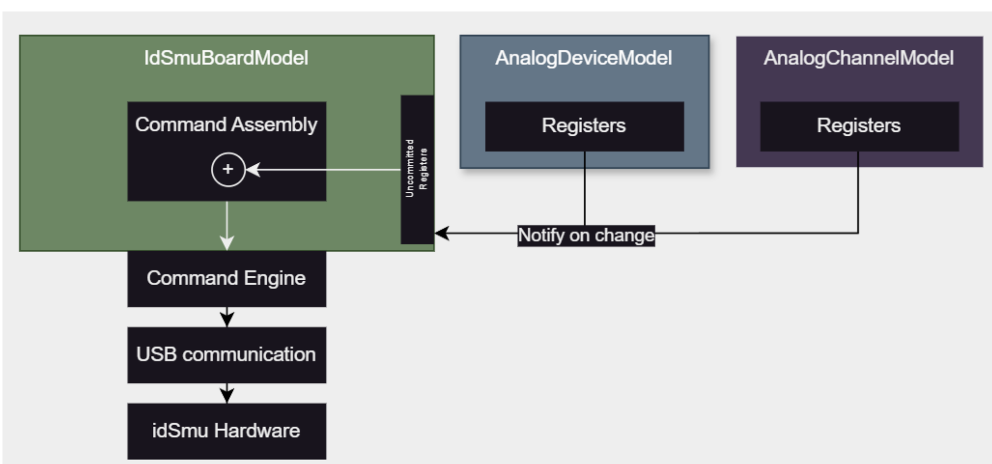


Fig. 1.6: The state of the hardware is modeled through registers

CHAPTER

TWO

API PROGRAMMING

This section describes the programming of the Device Engine API.

2.1 Initialization

2.1.1 Introduction

The first steps when working with the API consist of detecting and initializing the hardware.
During hardware detection, the device engine searches for connected idSMU hardware. If this is found, a hardware model is created. This model has already been briefly described in the introductory chapters.
However, detection of the hardware is not sufficient to communicate with the hardware. In a second step, the hardware must be initialized: In this step, for example, all default register values are set, the necessary power supply for the SMU/DPS chips is set and calibration data is read from a memory.

2.1.2 IdSmuService

The IdSmuService is the class that takes over the task of device detection and triggers the construction of the hardware model for each idSMU board found. This service class stores a reference to each board model and the user can then access it.

2.1.3 IdSmuServiceRunner

The IdSmuServiceRunner class encapsulates the rather complicated construction and lifetime management of the IdSmuService object.
The API user therefore has the option of instantiating this object instead. It is recommended to instantiate the IdSmuService via this object.
As soon as the IdSmuServiceRunner goes out of scope, it ensures that all necessary connections to the hardware are automatically closed cleanly.
When using the IdSmuService alone, the user would be responsible for this clean-up work. If this is forgotten, the software cannot be restarted without restarting the hardware.

```
srunner = IdSmuServiceRunner()
idsmu_service = srunner.get_idsmu_service()
```

2.1.4 Device detection

> **Note**
>
> Normally, the 'detection' and 'initialization' steps can be combined. Nevertheless, the individual steps are briefly described here.

The `detect_devices()` method of IdSmuService can be used to trigger the device detection. This method returns a CommandReply object. This is an object which, if devices were detected, contains a list of idSMU devices found.
If desired, the user can examine this object very quickly with his 'to_json' method. However, as the IdSmuService automatically creates a hardware model when hardware is found, the user has the option of accessing this directly.
After detection, some information can be retrieved:
With `get_board_addresses()` a list of all board addresses is returned. The `print_device_information()` returns a string table displaying some usefull information about the detected hardware.

2.1.5 Device initialization

Initialization takes place for all idSMU devices on a board (remember: a board is the container for one or more idSMU devices).
Once the hardware has been detected, this board can be initialized using the board address.

Example initializing a board with the address 'M1':

```
idsmu_service.initialize_board('M1')
```

Similar to the `detect_devices()` method, this method returns the result of this process. This time it returns a list of CommandReply objects, one entry for each idSMU device that has been initialized.

Full example detection and initialization:

```
smu_service = srunner.get_idsmu_service()
# detection
smu_service.detect_devices()
print(smu_service.print_device_information())
# initialization
print(smu_service.initialize_board('M1')[0].to_json())
# since the CommandReply content is quite verbose,
# we check the status of the device with the print_device_information again
print(smu_service.print_device_information())
```

2.1.6 Combining detection and initialization

Separate detection and initialization is rarely necessary (e.g. for special power settings).
It is therefore possible to combine both steps in one.
The method `detect_and_initialize_devices()` does what the name says: It detects the hardware and then initializes it.
The method `get_first_board()` is very similar with the difference that it returns the first detected and initialized board. This is particularly useful for single module boards.

Full example detection and initialization:

```
smu_service = srunner.get_idsmu_service()
myboard = smu_service.get_first_board()
```

2.1.7 Summary: Methods for device detection and initialization

IdSmuService methods:

Method	Explenation
detect_devices()	detects all connected idSmu devicesand builds a hardware model
get_board_addresses()	returns a list of all board addresses of the detected hardware
print_device_information()	returns a formatted string tablecontaining useful device information
initialize_board(<board_address>)	Initializes a board (the devices on it)
detect_and_initialize_devices()	Detects and initializes all devices
get_first_board()	Detects and initializes all devicesand returns the first board

2.2 The Board Model

Python	C++	Comment
idSmu1Modul	idSmu1Modul	Proxy object representing allidSMU 1(3) modules on the boardallowing to select a device with bracketoperator
idSmu2Modul	IdSmu2Modul	Proxy object representing allidSMU 2 modules on the boardallowing to select a device with bracketoperator

CHAPTER

THREE

TUTORIALS

3.1 Part 1: Basics

This is the introductory overview of programming the Aspect Device Engine Python API.
This document is available as pdf and interactive jupyter notebook. The introduction includes the following objectives:

- The few steps to initializing software and hardware
- How to retreive information about the hardware
- Providing a basic understanding of the structure of the API
- The programming of essential channel parameters
- Setting a voltage and measuring voltage and current

The following spoiler shows a python code snippet and few lines of code that are necessary with the API to generate and measure a voltage:

```
from aspectdeviceengine.enginecore import IdSmuServiceRunner
from aspectdeviceengine.enginecore import IdSmuService, IdSmuBoardModel

# 3 lines of code for the setup
srunner = IdSmuServiceRunner()
mbX1 : IdSmuBoardModel = srunner.get_idsmu_service().get_first_board()
channel1 = mbX1.idSmu2Modules['M1.S1'].smu.channels[1]

# 3 lines of code for configuration and measurement
channel1.enabled = True
channel1.voltage = 2
print(channel1.voltage)  # output : ~2.0
```

At the end of this document, this code and some of the background to it should be understandable.

3.1.1 Python imports

There are only a few python imports needed for this introduction. Everthing is imported from the *aspectdeviceengine.engincore* module. Actually, only the IdSmuServiceRunner would be needed since this is the only class that will be instantiated. The other classes are only imported for type hinting. The objects of these types are intantiated by the API services.

```
from aspectdeviceengine.enginecore import IdSmuService, IdSmuServiceRunner,␣
↪IdSmuBoardModel
```

3.1.2 Starting the services and hardware initialization

IdSmuServiceRunner

The *IdSmuServiceRunner* holds the references to the background services. If it goes out of scope, all services are shut down (cleanup processes). The lifetime should therefore be guranteed until the end of the session:

```
srunner = IdSmuServiceRunner()
```

IdSmuService

idSmu devices are detected by the **IdSmuService**. If the *get_first_board()* method is called prior to the detection method, the detection and initialization is performed automatically. This is useful for situations where no specific configuration needs to be done before initialization.

Important note: At the end of a session (be it a jupyter notebook or a python script) the services must be shut down. In the case of the termination of a python script, this happens automatically. When moving from one notebook tutorial to the next, either the kernel must be terminated or the shutdown() method must be executed manually (see last cell in the notebook).

The IdSmuBoardModel

The IdSmuBoardModel is the host and (multiside-)controller for idSmu devices.

```
mbX1 : IdSmuBoardModel = srunner.get_idsmu_service().get_first_board()
```

With this single line of code the Hardware is detected and initialized!
Let's print some basic information about the detected devices for this board.
The most relevant information is the **DeviceId** and the device type. The DeviceId is used as resource identifier/ locator for the different parts of the hardware. The format is \Mx.Sy.Cz, where x is the mainboard address, y is the device/slot number and z is the channel number.
In the case of the API, the terms *Resource-Id* and *Address* are sysnonyms for the same thing.

```
print(mbX1.device_information)
```

```
+----------+------------+-------+----------+-------------+
| DeviceId | IdSmu-Type | Name  | Firmware | Initialized |
+----------+------------+-------+----------+-------------+
| M1.S1    | IdSmu2     | M1.S1 | 0x08191f | true        |
+----------+------------+-------+----------+-------------+
```

3.1.3 Programming the hardware with the API

idSmu Modules/Devices and Channels

With this system of hierarchical resource localization, each resource is uniquely identifiable, even in a multi-board setup. As we can see (in a single idSmu board setup), there is exactly one device with the address "M1.S1".
An idSmu device (sometimes called module) can contain one or more channels. We can obtain more information about these channels, for example their IDs/addresses:

```
for idSmu2 in mbX1.idSmu2Modules.as_list():
    print(f'idSmu2 ID:     {idSmu2.hardware_id}')
    print(f'idSmu2 name:   {idSmu2.name}')
    print(f'channel IDs:   {idSmu2.channel_ids}')
```

```
idSmu2 ID:     M1.S1
idSmu2 name:   M1.S1
channel IDs:   ['M1.S1.C1', 'M1.S1.C2', 'M1.S1.C3', 'M1.S1.C4']
```

IdSmuDeviceModel

The idSmuModules classes are proxy classes that implement the [ ] operator for quick access to a device/module of type IdSmuDeviceModel and the as_list() method to get all devices of the same type on the board. There are implementations for the all types of idSmu.

To access a module we can simply use `idSmu2Modules['address or name of module']`

```
idSmu2 = mbX1.idSmu2Modules['M1.S1']
print(f"The module's id is {idSmu2.hardware_id} and the name is {idSmu2.name}")
```

```
The module's id is M1.S1 and the name is M1.S1
```

Alias names

Devices can be renamed, either programmatically or by so-called parameter settings. Parameter settings are applied during initialization and the changed name can thus be used immediately. (programming via parameter settings is an advanced topic and will not be dealt with here).
The advantage of renaming resources is that you can use an alias name for addressing instead of the rather abstract resource IDs / addresses:

```
idSmu2.name = 'MyFavoriteModule'
# now we can reference the module with the new name
my_fav_module = mbX1.idSmu2Modules['MyFavoriteModule']

print(f"The module's id is still {my_fav_module.hardware_id} and the new name is {my_
↪fav_module.name}")
```

```
The module's id is still M1.S1 and the new name is MyFavoriteModule
```

The board's device information now lists the new name:

```
print(mbX1.device_information)
```

```
+----------+------------+------------------+----------+-------------+
| DeviceId | IdSmu-Type | Name             | Firmware | Initialized |
+----------+------------+------------------+----------+-------------+
| M1.S1    | IdSmu2     | MyFavoriteModule | 0x08191f | true        |
+----------+------------+------------------+----------+-------------+
```

3.1.4 Descending further into the model hierarchy

The idSmu-Hardware is not just a source measurement unit, but combines various hardware components. For example, digital signals can be generated or a RAM memory can be used, depending on the hardware/software support. The software therefore models the hardware in dedicated units. One of the most important elements of the IdSmuDeviceModel mentioned above is the unit with which currents and voltages can be generated and measured. These units are called SMU or DPS, depending on the device type.

```
print(idSmu2.smu)
```

```
<aspectdeviceengine.enginecore.IdSmu2DeviceModel.Smu object at 0x000002056E17D9B0>
```

The channel models

The smu/dps subcomponents are again proxy objects and contain a Channels object.
This Channels object implements the [ ] operator for fast access to the channels of the source measurement unit. There is also a as_list() method again to itearate over the channels. The objects returned by the channels object are of type AnalogChannelModel. This class contains a large part of the methods and properties that you have to deal with in your daily work with the API.

```
for i, channel in enumerate(idSmu2.smu.channels.as_list()):
    print(f"Channel number {i+1} with name {channel.name}"
          f" and identifier {channel.hardware_id}")
    channel.name = f'MyChannel{i+1}'
```

```
Channel number 1 with name M1.S1.C1 and identifier M1.S1.C1
Channel number 2 with name M1.S1.C2 and identifier M1.S1.C2
Channel number 3 with name M1.S1.C3 and identifier M1.S1.C3
Channel number 4 with name M1.S1.C4 and identifier M1.S1.C4
```

After renaming a channel, there are 3 ways to address it:

- through the channel number (starting from 1)
- through the unique channel identifier
- through the channel name

```
print(idSmu2.smu.channels[1].hardware_id)
print(idSmu2.smu.channels["M1.S1.C1"].hardware_id)
print(idSmu2.smu.channels["MyChannel1"].hardware_id)
```

```
M1.S1.C1
M1.S1.C1
M1.S1.C1
```

Importante Note: Attempting to assign the same name to two different channels leads to an exception. Channel names must be unique. The reason is that the engine accepts names as resource identifiers for many operations. If the user were given the option to overwrite this name, bugs that are difficult to identify would be possible:

```
try:
    idSmu2.smu.channels[2].name="MyChannel1"
except Exception as e:
    print(f"Exception: {str(e)}")
```

```
Exception: The alias name MyChannel1 is already associated with the id M1.S1.C1
```

3.1.5 Parameterization and measurements

Now we have a reference to the channel object and can finally parameterize it and take measurements. A channel must be active so that a voltage (or a current) can be output or meaningful measurements can be made.
Let's check if the channel is enabled, and if it is not, enable it:

```
channel1 = idSmu2.smu.channels["MyChannel1"]
print(f'Channel enabled? {channel1.enabled}')
if not channel1.enabled:
    channel1.enabled = True
print(f'Channel enabled? {channel1.enabled}')
```

```
Channel enabled? False
Channel enabled? True
```

Measuring voltage and current

The quickest and easiest way to measure a voltage or a current are the properties `voltage` and `current`

```
print(f'Measured voltage: {channel1.voltage:6f}')
print(f'Measured current: {channel1.current:6f}')
```

```
Measured voltage: 0.000000
Measured current: 0.000002
```

Setting voltage and current

The quickest and easiest way to set a voltage or a current are the setters `voltage` and `current`

```
channel1.voltage = 3.14
print(f'Measured voltage: {channel1.voltage:6f}')

# Setting a current is only usefull if there is a load at the outputs
# wheras voltages can be measured on an open output
# channel1.current = 1E-3
# print(f'Measured current: {channel1.current:6f}')
```

```
Measured voltage: 3.141083
```

Bonus: Getting the maximum output voltage and output current ratings

```
vMin, vMax, iMin, iMax = channel1.output_ranges
print(f'Output voltage range: [{vMin:6f}, {vMax:6f}] V')
print(f'Output current range: [{iMin:6f}, {iMax:6f}] A')
```

```
Output voltage range: [-11.000000, 10.999664] V
Output current range: [-0.075000, 0.074998] A
```

```
srunner.shutdown()
```

3.2 Part 1b: Basics - Logging

This is the introductory overview of programming the Aspect Device Engine Python API.
This document is available as pdf and interactive jupyter notebook. The introduction includes the following objectives:

- How to log to the console and file

3.2.1 Logging

In case of programming problems with the hardware, it can be helpful to get a detailed text output with the processes in the software.
For this purpose, different levels of detail can be set in this text output, so-called log levels. By default, the log level is set to NONE (0). This means that no log is output.
The log levels can have the values 0 to 5. The API provides an enum value for each of these levels.
These are as follows: None_(0), Error(1), Warning(2), Info(3), Debug(4), Trace(5).
If, for example, the LogLevel Info(3) is set, errors, warnings and some useful information are output. If the 'Trace' level is set, even the communication protocol with the hardware is output.
The more detailed the log is, the lower the performance of the engine. You should therefore only switch on (detailed) logging in the event of problems that are not understood or a suspected bug.
There are two types of logging: console logging and file logging. Both can be switched on and configured independently of each other.
The configuration is relatively straightforward and is briefly described below using code examples.

```
from aspectdeviceengine.enginecore import IdSmuService, IdSmuServiceRunner,␣
↪IdSmuBoardModel, LogLevel
```

3.2.2 Console logging

Console logging can be switched on ready with the construction of the `IdSmuServiceRunner`.
The first parameter in the constructor is the log level as a numerical value from 0-5 (default is 0).
This log level can be set later using the `LogLevel` enum and the `LogService` object.

> Note: The log is displayed in the standard console and not in Jupyter Notebook

```
# Initialisation of the log level via the ServiceRunner in the highest level 'Trace'
srunner = IdSmuServiceRunner(5)
```

```
mbX1 : IdSmuBoardModel = srunner.get_idsmu_service().get_first_board()
```

```
# Check the log level via the console_log_level_property of the LogService object
print(srunner.log_service.console_log_level)
# Set a new console log level via the same property
srunner.log_service.console_log_level = LogLevel.Info
print(srunner.log_service.console_log_level)
```

```
LogLevel.Trace
LogLevel.Info
```

3.2.3 File logging

File logging must be activated in a separate step via the LogService object.
To do this, first tell the Log Service the path to the file to be logged. The file log level is then set.

```
srunner.log_service.file_log_level = LogLevel.Trace
srunner.log_service.set_log_file("C:\\tmp\\log.txt")
print(srunner.log_service.file_log_level)
```

```
LogLevel.Trace
```

```
# We issue a command to produce a log entry
mbX1.set_enable_clamps(True, ["M1.S1.C1"])
```

The output in the file will then look something like this:

> Device: Execution of commands triggered…
> CommandService: Removing command from queue: Name = WriteIdSmuConfigurationCommand, Identifier = 24
> CommandService: Waiting for commands to execute…
> Device: Executing command: Name = WriteIdSmuConfigurationCommand, Identifier = 24
> WriteReadCommand: Executing write read cycles…
> WriteReadCommand: Executing write read cycle number = 0
> FtdiService: Setting FTDI pipe timeout = 5000
> FtdiDevice: Writing to FTDI pipe…
> FtdiDevice: Successfully wrote to FTDI pipe. Bytes written = 36
> FtdiDevice: Word# written 00000: 0x00030000
> FtdiDevice: Word# written 00001: 0x00000001
> FtdiDevice: Word# written 00002: 0x0000001b

```
srunner.shutdown()
```

3.3 Part 2: More about channel measurements

This is the second introductory overview of programming the Aspect Device Engine Python API.
This document is available as pdf and interactive jupyter notebook. The introduction includes the following objectives:

- Get familiar with multiple measurements at multiple channels

---

```
from aspectdeviceengine.enginecore import IdSmuService, IdSmuServiceRunner,␣
↪IdSmuBoardModel
import plotly.graph_objects as go
import numpy as np
srunner = IdSmuServiceRunner()
```

```
mbX1 : IdSmuBoardModel = srunner.get_idsmu_service().get_first_board()

channel1 = mbX1.idSmu2Modules['M1.S1'].smu.channels["M1.S1.C1"]
channel2 = mbX1.idSmu2Modules['M1.S1'].smu.channels["M1.S1.C2"]
channel3 = mbX1.idSmu2Modules['M1.S1'].smu.channels["M1.S1.C3"]
channel4 = mbX1.idSmu2Modules['M1.S1'].smu.channels["M1.S1.C4"]
print(channel1)
```

```
<aspectdeviceengine.enginecore.AD5522ChannelModel object at 0x00000290455AB8B0>
```

3.3.1 Repeated measurements

While measurements can be repeated in the software, e.g. in a loop, as shown in the previous sections, the measurements can also be repeated directly on the hardware.
This has the advantage that the code is much more compact and that the measurements run much faster.
The **measure_voltages()** method takes one parameter: the number of repetitions for the measurement.

```
channel1.enabled = True
channel1.voltage = 1.14
number_of_measurements = 50
voltages_ch1 = channel1.measure_voltages(number_of_measurements)
print(voltages_ch1[1:10])
```

```
[1.138671875, 1.138671875, 1.138336181640625, 1.138336181640625, 1.138336181640625,
 ↪ 1.138336181640625, 1.137664794921875, 1.138336181640625, 1.137664794921875]
```

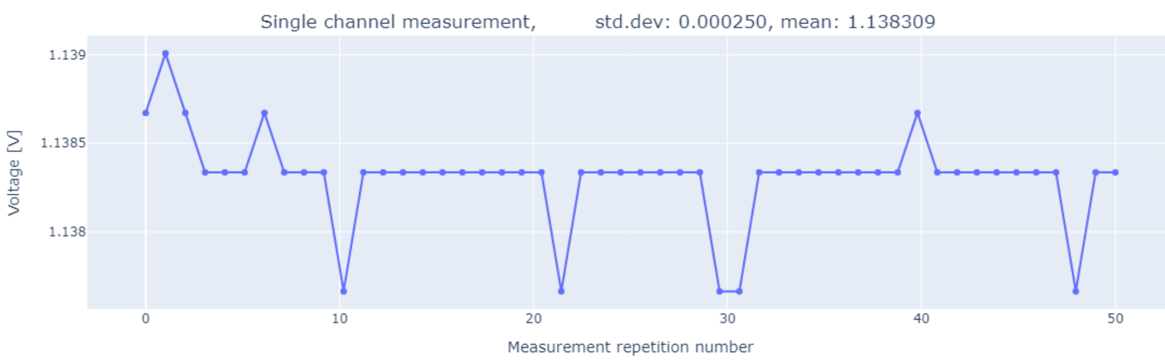
Repeat executing the next cell (Ctrl+Enter) and you will see how the below plot updates.
The plot stays close to 1.4V. Only the usual very small fluctuations due to noise can be seen.

```
number_of_measurements = 50
voltages_ch1 = channel1.measure_voltages(number_of_measurements)

x_ = np.linspace(0, number_of_measurements, number_of_measurements)
# Create traces
fig = go.Figure()
fig.add_trace(go.Scatter(x=x_, y=voltages_ch1,  mode='lines+markers',  name='voltages_
↪ch1'))

fig.update_layout(  title={'text': f"Single channel measurement, \
```

(continues on next page)

(continued from previous page)

```
        std.dev: {np.std(voltages_ch1):.6f}, mean: {np.mean(voltages_ch1):.6f}",  \
        'y':0.9,  'x':0.5, 'xanchor': 'center',  'yanchor': 'top'}, \
        xaxis_title='Measurement repetition number',  yaxis_title='Voltage [V]', \
        margin=dict(l=20, r=20, t=55, b=20))
fig
#uncomment in pure python script:
#fig.show()
```



Now we activate the other channels with an offset of one volt each.

```
channel2.enabled = True
channel2.voltage = 2.14
channel3.enabled = True
channel3.voltage = 3.14
channel4.enabled = True
channel4.voltage = 4.14
```

The measurements for each channel are run sequentally and the results are displayed in a plot:

```
voltages_ch1 = channel1.measure_voltages(50)
voltages_ch2 = channel2.measure_voltages(50)
voltages_ch3 = channel3.measure_voltages(50)
voltages_ch4 = channel4.measure_voltages(50)

x_ = np.linspace(0, 50, 50)
# Create traces
fig = go.Figure()
fig.add_trace(go.Scatter(x=x_, y=voltages_ch1,
                    mode='lines+markers',
                    name='voltages_ch1'))
fig.add_trace(go.Scatter(x=x_, y=voltages_ch2,
                    mode='lines+markers',
                    name='voltages_ch2'))
fig.add_trace(go.Scatter(x=x_, y=voltages_ch3,
                    mode='lines+markers',
                    name='voltages_ch3'))
fig.add_trace(go.Scatter(x=x_, y=voltages_ch4,
                    mode='lines+markers',
                    name='voltages_ch4'))
fig.update_layout(  title={'text': "4 channel measurement",  'y':0.9,  'x':0.5,
↪'xanchor': 'center',  'yanchor': 'top'},
                   xaxis_title='Measurement repetition number',  yaxis_title='Voltage␣
↪[V]', margin=dict(l=20, r=20, t=55, b=20))
```

(continues on next page)

(continued from previous page)

```
fig
#uncomment in pure python script:
#fig.show()
```

Do not forget to shut down the services before proceeding:

```
srunner.shutdown()
```

3.4 Part 3: Multiside and high performance

This is the third introductory overview of programming the Aspect Device Engine Python API.
This document is available as pdf and interactive jupyter notebook. The introduction includes the following objectives:

- Understanding the parallel or multi-side programming of idSMU resources

---

3.4.1 Intoduction

In the previous chapters, the idSMU channels were always programmed individually and sequentially.
This is the simplest and quickest approach to programming the hardware. This approach is sufficient in cases where a high-performance application is not important.
With its multi-threading approach, the aspect device engine also offers the possibility to programme and measure resources efficiently in parallel.

```
from aspectdeviceengine.enginecore import IdSmuService, IdSmuServiceRunner,␣
↪IdSmuBoardModel, MeasurementMode
import plotly.graph_objects as go
import numpy as np
srunner = IdSmuServiceRunner()
```

```
mbX1 : IdSmuBoardModel = srunner.get_idsmu_service().get_first_board()
mbX1.is_board_initialized()
```

```
True
```

The IdSmuBoardModel class is the host and controller for the detected hardware on an idSMU board.
A board can, for example, contain a single idSMU module (MbX-1) or 16 (MbX-16). Parallelisation is optimised for one

board, i.e. all theoretically possible 160 channels of an MbX-16 board could be set to 0V at once and the engine would attempt to carry out this process as efficiently as possible.
Parallelism between several boards is also given in the sense that the programming of a resource is a non-blocking process running in the background.
The most frequently used methods for parameterising and measuring a channel are also available as board methods.
For example, the property `.voltage` of a channel corresponds to the `set_voltages()` method of a board.
The plural in the method name already indicates that several channels can be programmed simultaneously here.

```
# The anatomy of a board method usually consists of the combination parameter-value,
# followed by a list of device or channel names (or identifiers)
# We could set a voltage for each channel separately:
mbX1.set_voltages(3.14, ["M1.S1.C1"])
mbX1.set_voltages(4.13, ["M1.S1.C2"])
# or in parallel:
mbX1.set_voltages(3.14, ["M1.S1.C1", "M1.S1.C2"])

# After assigning some alias names...
mbX1.idSmu2Modules['M1.S1'].smu.channels["M1.S1.C1"].name = "channel1"
mbX1.idSmu2Modules['M1.S1'].smu.channels["M1.S1.C2"].name = "channel2"
# ...the new names can be used in the board methods:
mbX1.set_voltages(3.14, ["channel1", "channel2"])
# Changing the measurement mode and enabling the channels is done in a similar way:
mbX1.set_measurement_modes(MeasurementMode.vsense, ["channel1", "channel2"])
mbX1.set_enable_channels(True,["channel1", "channel2"] )
```

3.4.2 Parallel measurements in detail

```
smuchannel1 = mbX1.idSmu2Modules['M1.S1'].smu.channels['channel1']
smuchannel2 = mbX1.idSmu2Modules['M1.S1'].smu.channels['channel2']
```

```
print(smuchannel1.enabled, smuchannel2.enabled)
```

```
True True
```

```
print(smuchannel1.voltage, smuchannel2.voltage)
```

```
3.13873291015625 3.14007568359375
```

We have set and measured the output voltage using the simple and intuitive method via channel properties, as presented in the last tutorials. These measurements run sequentially. Next, we will learn about the high-performance parallel method.
The IdSmuBoardModel has numerous methods for programming and measuring the resources it manages. One of these methods is `set_voltages()`, which can be used to set any number of channels together to the same voltage value.

```
mbX1.set_voltages(voltage=2.5, channel_names=["channel1", "channel1"])
# for SMU-based modules, set_currents() can be called as well
```

We can still use the "channel-method" to query the voltages:

```
print(smuchannel1.voltage, smuchannel2.voltage)
```

```
2.099090576171875 3.14007568359375
```

3.4.3 Synchronous and asynchronous measurements

When performing a measurement via the board controller (*IdSmuBoardModel*) on one or more channels, you have the choice of either waiting for the result until the measurement has been performed or letting the measurement run in the background. The latter is particularly useful for triggered measurements or for GUI applications or other asynchronous tasks.
This is set with the `wait_for_result` parameter of the `measure_channels()` method.
The `measure_channels()` method also offers the advantage of being able to set other parameters, such as the sample count or the number of measurements. Let's perform a synchronous measurement with 2 repetitions on 2 channels:

```
measresults = mbX1.measure_channels(wait_for_result=True, sample_count=1,␣
↪repetitions=2, channel_names=["channel1", "channel2"])
```

Important Note:

> The user is responsible to provide a valid list of channel names/identifiers. In addition to the valid names, care must also be taken to ensure that the channel names in the list are unique without duplicate entries. Otherwise errors will occur!

Measurement Results

The result of a measurement via a board is a vector (list) of measurement results.
Each of these measurement results relates to an idSmu module/device.
As the `measure_channels()` method can be used to measure several channels on several modules quasi-parallel, this result list can contain more than one entry. As we only measured on one module on two channels, the list only has one entry:

```
print(len(measresults))
measresult0 = measresults[0]
type(measresult0)
```

```
1
```

```
aspectdeviceengine.enginecore.ReadAdcCommandIdSmuResult
```

The elements of the result are of type `ReadAdcCommandIdSmuResult`. Useful properties are `device_id` , `channel_ids`, `channel_names`, `execution_time`:

```
print(f'The results come from the measurement on the device with the id {measresult0.
↪device_id}')
print(f'The total execution time (including data transfer via usb etc) was
↪{measresult0.execution_time} microseconds.')
```

```
The results come from the measurement on the device with the id M1.S1
The total execution time (including data transfer via usb etc) was 669␣
↪microseconds.
```

To further simplify the assignment of the results to the measured resources, we can query the IDs of the channels or their names (the names must have been set before execution, see above)

```
print(measresult0.channel_ids, measresult0.channel_names)
```

```
StringList[M1.S1.C1, M1.S1.C2] StringList[channel1, channel2]
```

There are now several ways to obtain the result for a specific channel (as numpy array), which are all equivalent:

```
print(measresult0.get_float_values("M1.S1.C1"))
print(measresult0["M1.S1.C1"])
print(measresult0["channel1"])
print(measresult0[measresult0.channel_names[0]])

# example of averaging the results for each channel:
for channel_name in measresult0.channel_names:
    print(f'{channel_name} with an average value of {np.mean(measresult0[channel_
↪name]):.4f} V')
```

```
[2.09875488 2.09942627]
[2.09875488 2.09942627]
[2.09875488 2.09942627]
[2.09875488 2.09942627]
channel1 with an average value of 2.0991 V
channel2 with an average value of 3.1399 V
```

3.4.4 Asynchronous measurements

With asynchronous measurements, there is no waiting for a measurement. The commanded measurements are started in the background in high-performance C++ threads. The measurement results can be retrieved in python at any time. The result of the non-waiting `measure_channels()` method is therefore an empty array as shown below:

```
measresults = mbX1.measure_channels(wait_for_result=False, sample_count=1,␣
↪repetitions=2, channel_names=["channel1", "channel2"])
print(measresults)
```

```
[]
```

The `get_measurement_results_for_channel()` method returns the result of at least the specified channel. If several channels are measured simultaneously on one device, as in this example, all results are returned (a device cannot return the results separately for each channel). Since *chanel1* and *channel2* channels are on the same device, an object is returned that bundles the results for these two channels. It is of the same type as the element already from the array after the synchronous call of the measurement. To recognize it, we call it “measresult0” again

```
measresult0 = mbX1.get_measurement_results_for_channel("channel1")
```

```
print(measresult0["channel1"])
print(measresult0.timecode)
```

```
[2.10043335 2.10043335]
[0 0]
```

Timecode generation during measurements

In addition to the actual measurement data, a time code can be generated that tracks the exact time at which a measurement was started. The following method is used to activate this (after restarting a device, the default is *disabled*):

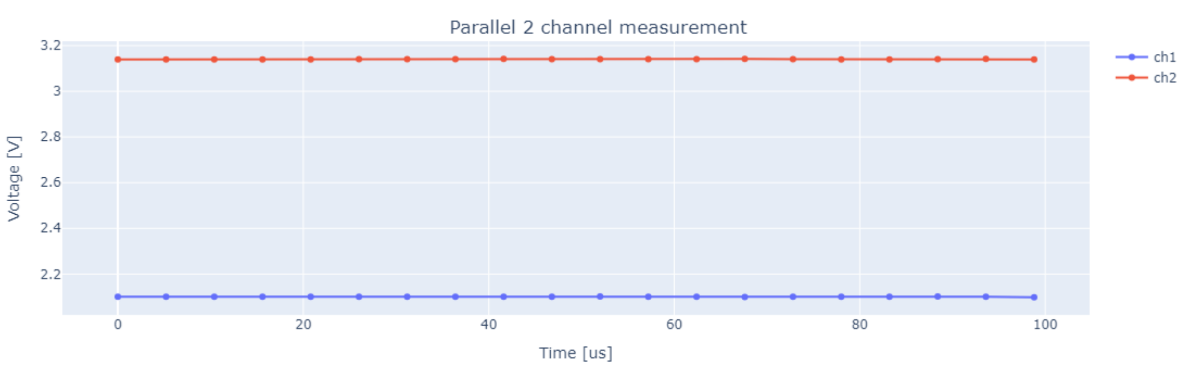
```
mbX1.enable_timecode("M1.S1")
```

The timecode is counted in multiples od 10ns (100Mhz clock). It is generated by counter that always runs when enabled, not only when commands are sent. We substract the first value from the array for an offset of zero and devide by 100 to get the time in units of microseconds.

```
measresults = mbX1.measure_channels(wait_for_result=False, sample_count=1,␣
↪repetitions=20, channel_names=["channel1", "channel2"])
measresult0 = mbX1.get_measurement_results_for_channel("channel1")

timecode = (measresult0.timecode-measresult0.timecode[0])/100
```

Now we can display the measurement results in a plot over an axis that displays the normalized time vs the measurement results

```
x_ = timecode
# Create traces
fig = go.Figure()
fig.add_trace(go.Scatter(x=x_, y=measresult0["channel1"],
                    mode='lines+markers',
                    name='ch1'))
fig.add_trace(go.Scatter(x=x_, y=measresult0["channel2"],
                    mode='lines+markers',
                    name='ch2'))
fig.update_layout(  title={'text': "Parallel 2 channel measurement",  'y':0.9,  'x':0.
↪5, 'xanchor': 'center',  'yanchor': 'top'},
                   xaxis_title='Time [us]',  yaxis_title='Voltage [V]',␣
↪margin=dict(l=20, r=20, t=55, b=20))
fig
#uncomment in pure python script:
#fig.show()
```



Do not forget to shut down the services before proceeding:

```
srunner.shutdown()
```

3.5 Part 4: Clamps, Compliance & Current Range

This is the forth introductory overview of programming the Aspect Device Engine Python API.
This document is available as pdf and interactive jupyter notebook. The introduction includes the following objectives:

- Understanding the current range and clamps/compliance of idSMU devices

---

```
from aspectdeviceengine.enginecore import IdSmuService, IdSmuServiceRunner,␣
↪IdSmuBoardModel, MeasurementMode, SmuCurrentRange, DpsCurrentRange, CurrentRange
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import numpy as np
srunner = IdSmuServiceRunner()
```

```
mbX1 : IdSmuBoardModel = srunner.get_idsmu_service().get_first_board()
print(mbX1.is_board_initialized())
smu_channel = mbX1.idSmu2Modules['M1.S1'].smu.channels["M1.S1.C1"]
smu_channel.name = "ch1"
smu_channel.enabled = True
```

```
True
```

Python can always be used to analyse the properties and methods of an object.
We search for current range and clamp in the list, the topics of this tutorial:

```
# Let's list all properties and methods of the smu channel object
[m for m in dir(smu_channel) if not m.startswith('_')]
```

```
['autorange',
 'clamp_enabled',
 'clamp_high_value',
 'clamp_low_value',
 'current',
 'current_range',
 'enabled',
 'hardware_id',
 'measure_current',
 'measure_currents',
 'measure_voltage',
 'measure_voltages',
 'name',
 'output_ranges',
 'set_clamp_high_value',
 'set_clamp_low_value',
 'set_current',
 'set_name',
 'set_voltage',
 'voltage']
```

3.5.1 Current Range

As an API user, you have the option of manually setting different current ranges.
It often makes sense to select a current range that is as close as possible to the current flowing through the DUT.
For small currents, the smallest possible matching current range should be selected.
This increases the accuracy when measuring a current. The two possible device types of an idSMU (SMU and DPS) each have different current ranges.

Setting and querying the current ranges

The enum class for the current ranges only contains the respective valid members at channel level for SMU or DPS.
The enum at board level is a union of both.
In the latter case, the user is responsible for not selecting the wrong value.

```
print(list(SmuCurrentRange.__members__))
print(list(DpsCurrentRange.__members__))
print(list(CurrentRange.__members__))
```

```
['Range_5uA', 'Range_20uA', 'Range_200uA', 'Range_2mA', 'Range_70mA']
['Range_25uA', 'Range_250uA', 'Range_2500uA', 'Range_25mA', 'Range_500mA', 'Range_
↪1200mA']
['Range_5uA', 'Range_20uA_SMU', 'Range_200uA_SMU', 'Range_2mA_SMU', 'Range_70mA_SMU
↪', 'Range_25uA_DPS', 'Range_250uA_DPS', 'Range_2500uA_DPS', 'Range_25mA_DPS',
↪'Range_500mA_DPS', 'Range_1200mA_DPS']
```

The default value (after initialization) of a SMU based device is 70mA (and 500mA for a DPS device). The `current_range` property of a channel or the `get_current_range()` method of a board can be used to query the current range:

```
print(smu_channel.current_range)
print(mbX1.get_current_range("ch1"))
```

```
SmuCurrentRange.Range_70mA
CurrentRange.Range_70mA_SMU
```

The corresponding setter and board method are `current_range` and `set_current_ranges()`

```
smu_channel.current_range = SmuCurrentRange.Range_2mA
print(mbX1.get_current_range("ch1"))
mbX1.set_current_ranges(CurrentRange.Range_200uA_SMU, ["ch1"])
print(smu_channel.current_range)
```

```
CurrentRange.Range_2mA_SMU
SmuCurrentRange.Range_200uA
```

More about current ranges and autoranging in the next tutorial.

3.5.2 Current and voltage clamps

An SMU or DPS channel has clamps. If a voltage is forced, the clamp is a current clamp.
If a current is forced (SMU types only), a voltage is clamped.
The clamp is active by default. We can check this with the 'clamp_enabled' property:

```
print(f'Is clamp enabled? {smu_channel.clamp_enabled}')
```

```
Is clamp enabled? True
```

In voltage force mode, the default clamp value is 70mA

```
print(f'The default upper clamp value is {smu_channel.clamp_high_value}A')
```

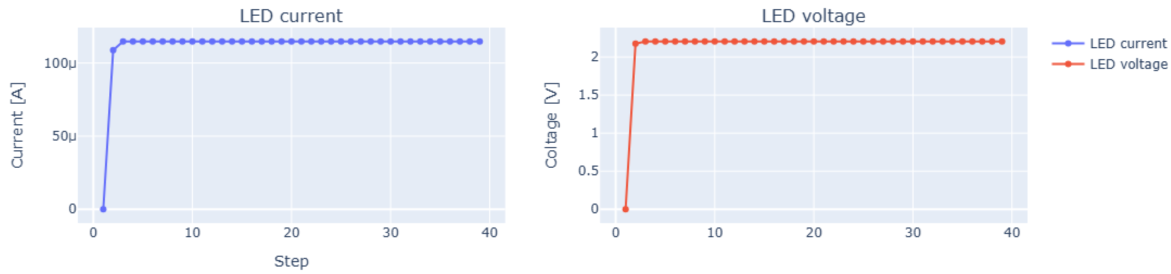
```
The default upper clamp value is 0.07A
```

To see the clamp in action, the voltage is swept across an LED and the current is measured (more on IV sweeps in the next chapters).
The clamp is first set to 100uA and the current range is adjusted to this current range:

```
currents = np.zeros(40);voltages = np.zeros(40)
smu_channel.clamp_enabled = True

smu_channel.clamp_high_value=0.0001
# an alternative to set lower and upper clamp together is to use the board method:
# mbX1.set_clamps_low_and_high_values(-0.001, 0.0001, ["ch1"])
smu_channel.current_range = SmuCurrentRange.Range_200uA
for i in range(1,40):
    smu_channel.voltage = 2.0+0.2*i
    currents[i] = smu_channel.current
    voltages[i] = smu_channel.voltage
```

As can be seen in the plot, the clamp becomes active slightly above the set maximum.

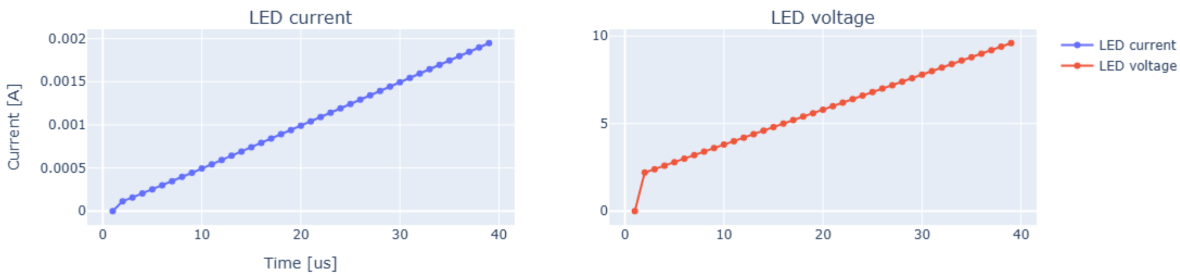
```
fig = make_subplots(rows=1, cols=2,  subplot_titles=("LED current", "LED voltage"))
fig.add_trace(go.Scatter(x=np.arange(1,40), y=currents,
                    mode='lines+markers+text',
                    textposition="top center",
                    name='LED current'),  row=1, col=1)
fig.add_trace(go.Scatter(x=np.arange(1,40), y=voltages,
                    mode='lines+markers+text',
                    textposition="top center",
                    name='LED voltage'),  row=1, col=2)
fig.update_layout(height=600)
fig.update_xaxes(title_text="Step", row=1, col=1);fig.update_yaxes(title_text=
↪"Current [A]", row=1, col=1)
fig.update_xaxes(title_text="Step", row=1, col=1);fig.update_yaxes(title_text=
↪"Coltage [V]", row=1, col=2)
fig
#uncomment in pure python script:
#fig.show()
```



The clamp will now be disabled and the current measurement will be repeated:

```
currents = np.zeros(40);voltages = np.zeros(40)

smu_channel.clamp_high_value=0.0001
smu_channel.clamp_enabled = False
smu_channel.current_range = SmuCurrentRange._2mA
for i in range(1,40):
    smu_channel.voltage = 2.0+0.2*i
    currents[i] = smu_channel.current
    voltages[i] = smu_channel.voltage
```

```
fig = make_subplots(rows=1, cols=2,  subplot_titles=("LED current", "LED voltage"))
fig.add_trace(go.Scatter(x=np.arange(1,40), y=currents,
                    mode='lines+markers+text',
                    textposition="top center",
                    name='LED current'),  row=1, col=1)
fig.add_trace(go.Scatter(x=np.arange(1,40), y=voltages,
                    mode='lines+markers+text',
                    textposition="top center",
                    name='LED voltage'),  row=1, col=2)
fig.update_layout(height=600)
fig.update_xaxes(title_text="Step", row=1, col=1);fig.update_yaxes(title_text=
↪"Current [A]", row=1, col=1)
fig.update_xaxes(title_text="Step", row=1, col=1);fig.update_yaxes(title_text=
↪"Coltage [V]", row=1, col=2)
fig
#uncomment in pure python script:
#fig.show()
```



```
srunner.shutdown()
```

```
measresults = mbX1.measure_channels(wait_for_result=True, sample_count=1,␣
↪repetitions=2, channel_names=["channel1", "channel2"])
```

Important Note:

The user is responsible to provide a valid list of channel names/identifiers. In addition to the valid names, care must also be taken to ensure that the channel names in the list are unique without duplicate entries. Otherwise errors will occur!

Measurement Results

The result of a measurement via a board is a vector (list) of measurement results.
Each of these measurement results relates to an idSmu module/device.
As the `measure_channels()` method can be used to measure several channels on several modules quasi-parallel, this result list can contain more than one entry. As we only measured on one module on two channels, the list only has one entry:

```
print(len(measresults))
measresult0 = measresults[0]
type(measresult0)
```

```
1
```

```
aspectdeviceengine.enginecore.ReadAdcCommandIdSmuResult
```

The elements of the result are of type `ReadAdcCommandIdSmuResult`. Useful properties are `device_id` , `channel_ids`, `channel_names`, `execution_time`:

```
print(f'The results come from the measurement on the device with the id {measresult0.
↪device_id}')
print(f'The total execution time (including data transfer via usb etc) was
↪{measresult0.execution_time} microseconds.')
```

```
The results come from the measurement on the device with the id M1.S1
The total execution time (including data transfer via usb etc) was 669␣
↪microseconds.
```

To further simplify the assignment of the results to the measured resources, we can query the IDs of the channels or their names (the names must have been set before execution, see above)

```
print(measresult0.channel_ids, measresult0.channel_names)
```

```
StringList[M1.S1.C1, M1.S1.C2] StringList[channel1, channel2]
```

There are now several ways to obtain the result for a specific channel (as numpy array), which are all equivalent:

```
print(measresult0.get_float_values("M1.S1.C1"))
print(measresult0["M1.S1.C1"])
print(measresult0["channel1"])
print(measresult0[measresult0.channel_names[0]])
```

(continues on next page)

(continued from previous page)

```
# example of averaging the results for each channel:
for channel_name in measresult0.channel_names:
    print(f'{channel_name} with an average value of {np.mean(measresult0[channel_
↪name]):.4f} V')
```

```
[2.09875488 2.09942627]
[2.09875488 2.09942627]
[2.09875488 2.09942627]
[2.09875488 2.09942627]
channel1 with an average value of 2.0991 V
channel2 with an average value of 3.1399 V
```

With asynchronous measurements, there is no waiting for a measurement. The commanded measurements are started in the background in high-performance C++ threads. The measurement results can be retrieved in python at any time. The result of the non-waiting `measure_channels()` method is therefore an empty array as shown below:

```
measresults = mbX1.measure_channels(wait_for_result=False, sample_count=1,␣
↪repetitions=2, channel_names=["channel1", "channel2"])
print(measresults)
```

```
[]
```

The `get_measurement_results_for_channel()` method returns the result of at least the specified channel. If several channels are measured simultaneously on one device, as in this example, all results are returned (a device cannot return the results separately for each channel). Since *chanel1* and *channel2* channels are on the same device, an object is returned that bundles the results for these two channels. It is of the same type as the element already from the array after the synchronous call of the measurement. To recognize it, we call it "measresult0" again

```
measresult0 = mbX1.get_measurement_results_for_channel("channel1")
```

```
print(measresult0["channel1"])
print(measresult0.timecode)
```

```
[2.10043335 2.10043335]
[0 0]
```

Timecode generation during measurements

In addition to the actual measurement data, a time code can be generated that tracks the exact time at which a measurement was started. The following method is used to activate this (after restarting a device, the default is *disabled*):

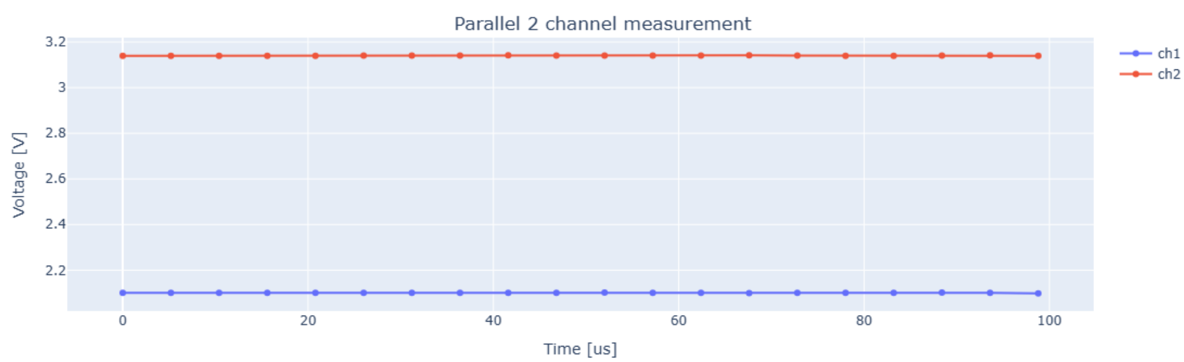
```
mbX1.enable_timecode("M1.S1")
```

The timecode is counted in multiples od 10ns (100Mhz clock). It is generated by counter that always runs when enabled, not only when commands are sent. We substract the first value from the array for an offset of zero and devide by 100 to get the time in units of microseconds.

```
measresults = mbX1.measure_channels(wait_for_result=False, sample_count=1,␣
↪repetitions=20, channel_names=["channel1", "channel2"])
measresult0 = mbX1.get_measurement_results_for_channel("channel1")

timecode = (measresult0.timecode-measresult0.timecode[0])/100
```

Now we can display the measurement results in a plot over an axis that displays the normalized time vs the measurement results

```
x_ = timecode
# Create traces
fig = go.Figure()
fig.add_trace(go.Scatter(x=x_, y=measresult0["channel1"],
                    mode='lines+markers',
                    name='ch1'))
fig.add_trace(go.Scatter(x=x_, y=measresult0["channel2"],
                    mode='lines+markers',
                    name='ch2'))
fig.update_layout(  title={'text': "Parallel 2 channel measurement",  'y':0.9,  'x':0.
↪5, 'xanchor': 'center',  'yanchor': 'top'},
                   xaxis_title='Time [us]',  yaxis_title='Voltage [V]',␣
↪margin=dict(l=20, r=20, t=55, b=20))
fig
#uncomment in pure python script:
#fig.show()
```



Do not forget to shut down the services before proceeding:

```
srunner.shutdown()
```

3.6 Part 5: IV Measurements

This is the fith introductory overview of programming the Aspect Device Engine Python API.
This document is available as pdf and interactive jupyter notebook. The introduction includes the following objectives:

- Performing IV sweeps
- Working with autoranging in IV-sweeps

Please note: This chapter shows how to perform simple IV sweeps in software.
For more advanced IV sweeps, see the next chapter 'List Sweeps'.

```
from aspectdeviceengine.enginecore import IdSmuService, IdSmuServiceRunner,␣
↪IdSmuBoardModel, MeasurementMode, FunctionGeneratorType, generate_function_
↪generator_data, CurrentRange
from aspectdeviceengine.enginecore import IdSmuBoardModel,␣
```

(continues on next page)

(continued from previous page)

```
↪ListSweepChannelConfiguration, ListSweep
import plotly.graph_objects as go
import numpy as np
srunner = IdSmuServiceRunner()
```

A few preparatory measures, nothing new here:

```
# Inintialization and preparation:
mbX1 : IdSmuBoardModel = srunner.get_idsmu_service().get_first_board()
print(mbX1.is_board_initialized())
device_id = "M1.S1"
channel1_id = f'{device_id}.C1'
channel2_id = f'{device_id}.C3'
channel_list = [channel1_id, channel2_id]

mbX1.set_voltages(1, channel_list)
mbX1.set_enable_channels(True,channel_list )
mbX1.set_measurement_modes(MeasurementMode.isense, channel_list)
mbX1.set_current_ranges(CurrentRange.Range_2mA_SMU, channel_list)
```

```
True
```

We define a simple helper function that can perform a software sweep.
It returns the force values, the measured values and the current range for each step:

```
def sw_sweep(start, stop, step):
    force_values = np.arange(start,stop+step, step)
    ch1_results = np.zeros(len(force_values))
    ch2_results = np.zeros(len(force_values))
    ch1_ranges = []
    ch2_ranges = []
    for i, voltage in enumerate(force_values):
        mbX1.set_voltages(voltage, channel_list)
        measresults = mbX1.measure_channels(wait_for_result=False, sample_count=1,␣
↪repetitions=1, channel_names=channel_list)
        measresult0 = mbX1.get_measurement_results_for_channel(channel1_id)
        ch1_ranges.append(mbX1.idSmu2Modules['M1.S1'].smu.channels[channel1_id].
↪current_range.name)
        ch2_ranges.append(mbX1.idSmu2Modules['M1.S1'].smu.channels[channel2_id].
↪current_range.name)
        ch1_results[i] = measresult0[channel1_id][0]
        ch2_results[i] = measresult0[channel2_id][0]
    return (force_values, ch1_results, ch2_results, ch1_ranges,ch2_ranges)
```

3.6.1 Autoranging in manual IV sweeps

The first step is to perform the sweep with autoranging deactivated. The respective current range is then output for each step.

```
mbX1.set_current_ranges(CurrentRange.Range_200uA_SMU, channel_list)
mbX1.enable_autorange(False, channel_list)
mbX1.set_voltages(1, channel_list)
sweep_without_autorange = sw_sweep(1,4, 0.2)
print(sweep_without_autorange[3])
```

```
['Range_200uA', 'Range_200uA', 'Range_200uA', 'Range_200uA', 'Range_200uA', 'Range_
↪200uA', 'Range_200uA', 'Range_200uA', 'Range_200uA', 'Range_200uA', 'Range_200uA
↪', 'Range_200uA', 'Range_200uA', 'Range_200uA', 'Range_200uA', 'Range_200uA']
```

Autoranging is performed for both channels using the `enable_autorange()` method:

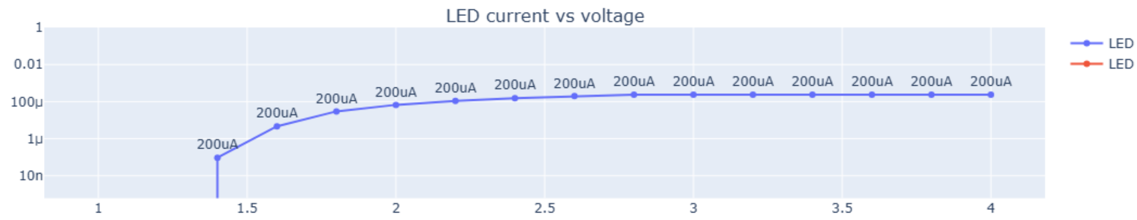
```
mbX1.enable_autorange(True, channel_list)
#Alternatively, autoranging can also be queried or set via the properties/setter of a␣
↪channel
print(mbX1.idSmu2Modules['M1.S1'].smu.channels[channel1_id].autorange)
sweep_with_autorange = sw_sweep(1,4, 0.2)
print(sweep_with_autorange[4])
```

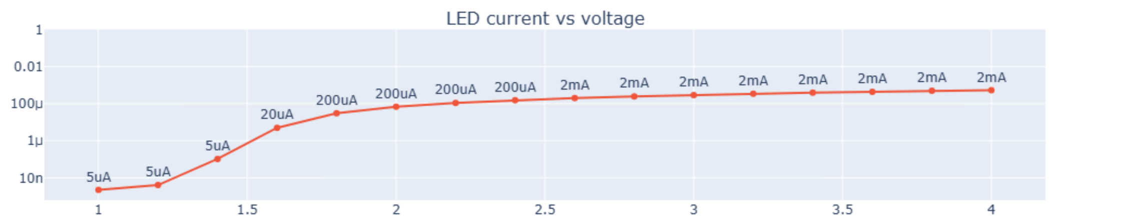
```
True
['Range_5uA', 'Range_5uA', 'Range_5uA', 'Range_5uA', 'Range_5uA', 'Range_5uA',
↪'Range_5uA', 'Range_20uA', 'Range_200uA', 'Range_200uA', 'Range_200uA', 'Range_
↪200uA', 'Range_2mA', 'Range_2mA', 'Range_2mA', 'Range_2mA']
```

3.6.2 Plotting the results

The following plots show the difference between the sweep with and without autorange. With a fixed value of 200uA, the current curve saturates at approximately this value.
With an autorange set, the better resolution at low currents becomes clear. The curve does not saturate and reaches around 500uA. The plots are annotated with the current ranges:
(Note: the enum prefixes 'Range_' are removed for better readability)

```
from plotly.subplots import make_subplots
def create_fig():
    fig = make_subplots(rows=2, cols=1,  subplot_titles=("LED current vs voltage",
↪"LED current vs voltage"))
    fig.add_trace(go.Scatter(x=sweep_without_outorange[0], y=sweep_without_
↪outorange[1],
                        mode='lines+markers+text',
                        text = [s.replace("Range_", "") for s in sweep_without_
↪autorange[3]],
                        textposition="top center",
                        name='LED'),  row=1, col=1)
    fig.add_trace(go.Scatter(x=sweep_with_outorange[0], y=sweep_with_outorange[1],
                        mode='lines+markers+text',
                        text = [s.replace("Range_", "") for s in sweep_with_
↪autorange[3]],
                        textposition="top center",
                        name='LED'),  row=2, col=1)
    fig.update_yaxes(type="log", range=[np.log(0.0001),np.log(1)], row=1, col=1)
    fig.update_yaxes(type="log", range=[np.log(0.0001),np.log(1)], row=2, col=1)
    fig.update_layout(height=600)
    return fig
fig = create_fig()
fig
#uncomment in pure python script:
#fig.show()
```






```
srunner.shutdown()
```

3.7 Part 6: List sweeps

This is the sixth introductory overview of programming the Aspect Device Engine Python API.
This document is available as pdf and interactive jupyter notebook. The introduction includes the following objectives:

- Learn how to use the ListSweep class to perform multi-channel sweeps in sub-milliseconds resolution

The *function generator* methods already discussed can be used to generate simple sweeps in real time. With the so-called *list sweeps*, more complex measurements can be carried out simultaneously on several channels of a module.

> Please note: In this example, an idSMU board equipped with LEDs wit series resistors at the outputs was used. If a differently configured board is used, the measurement curves may look different. In the case of open outputs, the sweep can also be performed via the voltage instead of the current. The voltage is measured back correctly, even with open outputs!

```
from aspectdeviceengine.enginecore import IdSmuService, IdSmuServiceRunner,␣
↪IdSmuBoardModel, MeasurementMode, FunctionGeneratorType, generate_function_
↪generator_data, CurrentRange
from aspectdeviceengine.enginecore import IdSmuBoardModel,␣
↪ListSweepChannelConfiguration, ListSweep
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import numpy as np
srunner = IdSmuServiceRunner()
mbX1 : IdSmuBoardModel = srunner.get_idsmu_service().get_first_board()
```

```
# It is always a good idea to check if the modules on the board are initialized
print(mbX1.is_board_initialized())
print(mbX1.device_information)
```

```
True
+----------+------------+-------+----------+-------------+
| DeviceId | IdSmu-Type | Name  | Firmware | Initialized |
+----------+------------+-------+----------+-------------+
| M1.S1    | IdSmu2     | M1.S1 | 0x08191f | true        |
+----------+------------+-------+----------+-------------+
```

3.7.1 Preparation

We prepare some variables and enable 2 channels of one idSMU module for the sweep with the multi-side methods. Nothing new here:

```
device_id = "M1.S1"
channel1_id = f'{device_id}.C1'
channel2_id = f'{device_id}.C3'

# assign meaningful alias names for the channels
mbX1.set_channel_name(channel1_id, "LED1")
mbX1.set_channel_name(channel2_id, "LED2")
mbX1.set_device_name(device_id, "My_LED_Module")
channel_list = ['LED1', 'LED2']

# enable the two channels together
mbX1.set_enable_channels(True,channel_list)
idSmu2 = mbX1.idSmu2Modules['M1.S1']

# print some channel information
for i, channel in enumerate(idSmu2.smu.channels.as_list()):
    print(f"Channel {i+1} with name {channel.name}" +
          " is enabled: " + ("YES" if channel.enabled else "NO"))
```

```
Channel 1 with name LED1 is enabled: YES
Channel 2 with name M1.S1.C2 is enabled: NO
Channel 3 with name LED2 is enabled: YES
Channel 4 with name M1.S1.C4 is enabled: NO
```

3.7.2 Preparing the parameters before a sweep

It is **important to understand** that the sweep works with the previously set configurations for the **force type** (voltage for DSP and SMU or current for SMU types) and the **measurement type** (voltage or current). These configurations are *not* part of the parameterisation of the sweep.
With a SMU type there is the option of forcing voltage or current, with a DPS type only voltage. The `set_voltages()` method encapsulates switching to the force type voltage and setting the output value. Although there are also dedicated methods for switching the force type, this is the simplest (for the SMU there is a corresponding `set_currents()` method). To measure a current in the sweep, we set the measurement type to current *(isense)*.
We then adjust the current range to the maximum expected current. An autorange does not currently exist for the list sweeps.

```
mbX1.set_voltages(1, channel_list)
mbX1.set_measurement_modes(MeasurementMode.isense, channel_list)
mbX1.set_current_ranges(CurrentRange.Range_2mA_SMU, channel_list) # Note: the ranges␣
↪for DPS-Modules differ from SMU modules!
mbX1.get_output_force_value(channel_list[0])
```

```
1.0
```

3.7.3 Configuring the sweep

The `ListSweepChannelConfiguration` and `ListSweep` classes are involved in the parameterisation of a sweep.
`ListSweepChannelConfiguration` is used to configure the form of the sweep for a channel.
`ListSweep` executes the sweep and saves the measurement results for each sweep.
In addition, parameters common to all sweeps are set here, such as the measurement delay.

3.7.4 Single channel sweep

Configuration

We configure a simple linear sweep from 1V to 3V with 20 steps for a single channel. The start and end values are always included in the sweep.
This is why you can expect *number of steps + 1* measurement results.
Important note:

> The number of sweep steps is currently limited to 52 steps for a single channel sweep as the memory from which the sweep is executed is limited (see advanced topics)

With 4 lines of code you can configure an executable sweep. In addition, the measurement delay is adjusted in this example:

```
# Instantiation of a channel configuration
config_ch1 : ListSweepChannelConfiguration = ListSweepChannelConfiguration()

# Configuration of a 20-steps linear sweep from 1 to 3 including 1 and 3, in units of␣
↪volt
# (because the output force typ was set to voltage before)
config_ch1.set_linear_sweep(1, 3, 20)

# Instantiation of a ListSweep object
sweep_1ch : ListSweep = ListSweep("My_LED_Module", mbX1)

# By adding at least one channel configuration to the list sweep, the sweep is ready␣
↪to start.
sweep_1ch.add_channel_configuration("LED1", config_ch1)

# The default measurement delay of 0 (plus some overhead in the lower us-range)
# is often to short. To allow the output voltage to settle before the measurement
# this time is increased to 100 microseconds
sweep_1ch.set_measurement_delay(100)

# let's examine the sweep array before execution:
print(f'Size of sweep array: {len(config_ch1.force_values)}')
```

(continues on next page)

(continued from previous page)

```
print(f'Type of sweep array: {type(config_ch1.force_values)}')
print(f'Values: {config_ch1.force_values}')
```

```
Size of sweep array: 21
Type of sweep array: <class 'numpy.ndarray'>
Values: [1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.  2.1 2.2 2.3 2.4 2.5 2.6 2.7
 2.8 2.9 3. ]
```

Execution and results

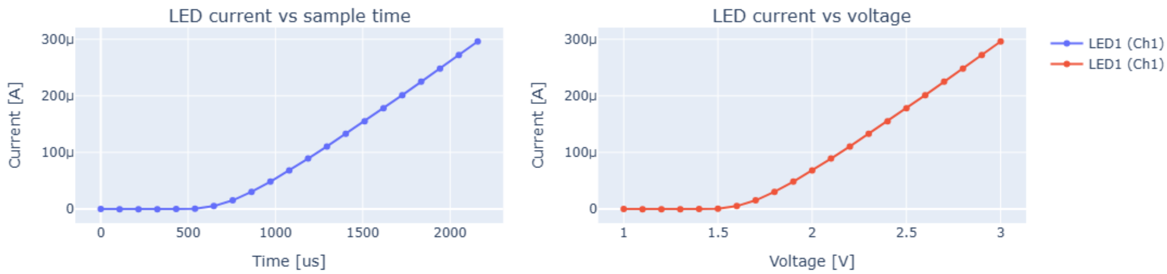
The sweep is executed by calling the `run()` method.

```
# The run() method starts the sweep and returns after the measurement sweep is␣
↪finished
sweep_1ch.run()
# The get_measurement_result() method returns the measured values
currents_LED1 = sweep_1ch.get_measurement_result("LED1")

# The get_force_values() of the configuration object returns the forced values
voltages_LED1 = config_ch1.force_values

# The measurement times (=sample times in case of the sample count = 1 which is␣
↪default)
# can be obtained via the sweep object
sample_times = sweep_1ch.timecode
```

Plotting the results

The results are visualised as plots below:

```
fig = make_subplots(rows=1, cols=2,  subplot_titles=("LED current vs sample time",
↪"LED current vs voltage"))
fig.add_trace(go.Scatter(x=sample_times, y=currents_LED1,
                    mode='lines+markers',
                    name='LED1 (Ch1)'),  row=1, col=1)
fig.add_trace(go.Scatter(x=voltages_LED1, y=currents_LED1,
                    mode='lines+markers',
                    name='LED1 (Ch1)'),  row=1, col=2)
fig.update_xaxes(title_text="Time [us]", row=1, col=1)
fig.update_yaxes(title_text="Current [A]", row=1, col=1)
fig.update_xaxes(title_text="Voltage [V]", row=1, col=2)
fig.update_yaxes(title_text="Current [A]", row=1, col=2)
fig
#uncomment in pure python script:
#fig.show()
```



3.7.5 Multi channel sweep

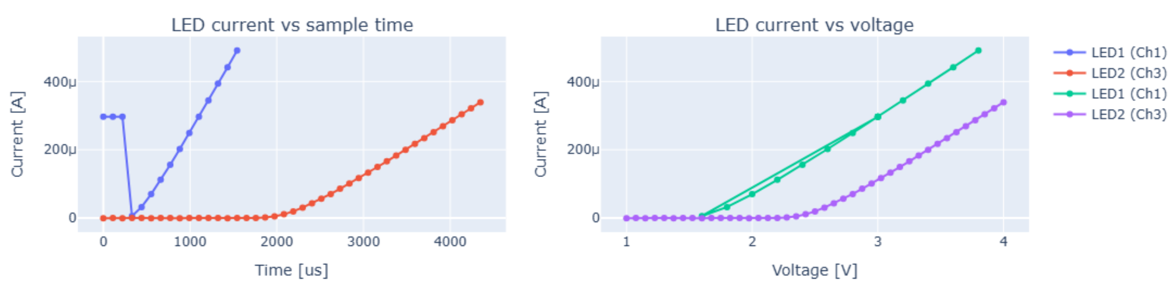
With a few simple modifications for a second channel, a multi-channel sweep is parameterised. The sweeps can be of different lengths. Here we parameterise both channels differently to show the possibility of flexibly setting the sweep parameters for each channel. Instead of the built-in method for a linear sweep,
this time we use an array with its own force values on the first channel
(even if the numpy arange() function again generates linear values with equidistant values)

```
mbX1.set_current_ranges(CurrentRange.Range_2mA_SMU, channel_list)
# first channel configruation
config_ch1 : ListSweepChannelConfiguration = ListSweepChannelConfiguration()
# custom force values
force_values = np.arange(1, 4, 0.20)
# We modify the numpy array a little to distinguish the sweep curve from the linear␣
↪case
force_values[0:3] = 3
config_ch1.force_values = force_values

# second channel configuration with linear sweep
config_ch2 : ListSweepChannelConfiguration = ListSweepChannelConfiguration()
config_ch2.set_linear_sweep(1, 4, 40)

sweep : ListSweep = ListSweep("My_LED_Module", mbX1)
sweep.add_channel_configuration("LED1", config_ch1)
sweep.add_channel_configuration("LED2", config_ch2)
sweep.set_measurement_delay(100)
sweep.run()

currents_LED1 = sweep.get_measurement_result("LED1")
currents_LED2 = sweep.get_measurement_result("LED2")
sample_times = sweep.timecode
```

Plotting the results

The results are visualised as plots below:

```
fig = make_subplots(rows=1, cols=2,  subplot_titles=("LED current vs sample time",
↪"LED current vs voltage"))
fig.add_trace(go.Scatter(x=sample_times, y=currents_LED1,
                    mode='lines+markers',
                    name='LED1 (Ch1)'), row=1, col=1)
fig.add_trace(go.Scatter(x=sample_times, y=currents_LED2,
                    mode='lines+markers',
                    name='LED2 (Ch3)'), row=1, col=1)
fig.add_trace(go.Scatter(x=config_ch1.force_values, y=currents_LED1,
                    mode='lines+markers',
                    name='LED1 (Ch1)'), row=1, col=2)
fig.add_trace(go.Scatter(x=config_ch2.force_values, y=currents_LED2,
                    mode='lines+markers',
                    name='LED2 (Ch3)'), row=1, col=2)
fig.update_xaxes(title_text="Time [us]", row=1, col=1)
fig.update_yaxes(title_text="Current [A]", row=1, col=1)
fig.update_xaxes(title_text="Voltage [V]", row=1, col=2)
fig.update_yaxes(title_text="Current [A]", row=1, col=2)
fig
#uncomment in pure python script:
#fig.show()
```



3.7.6 Advanced topics

Changing the current range during a sweep

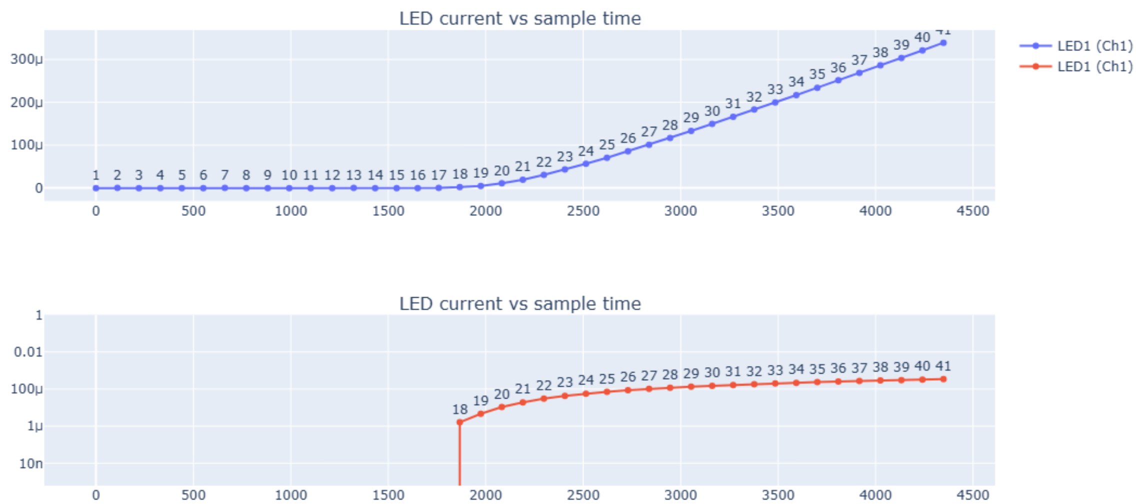
The list sweep does not support autoranging. However, it is possible to change the range at user-defined points in the sweep.
The last sweep ran with a range of 2mA. We plot the last sweep again and annotate the step indices to the plot.
The switch-on behaviour becomes visible at steps 20-21 or 10-20 microamperes:

```
def create_fig():
    fig = make_subplots(rows=2, cols=1,  subplot_titles=("LED current vs sample time",
↪ "LED current vs sample time"))
    fig.add_trace(go.Scatter(x=sweep.timecode, y=sweep.get_measurement_result("LED2"),
                        mode='lines+markers+text',
                        text = [str(i) for i in range(1, len(sample_times) + 1)],
```

(continues on next page)

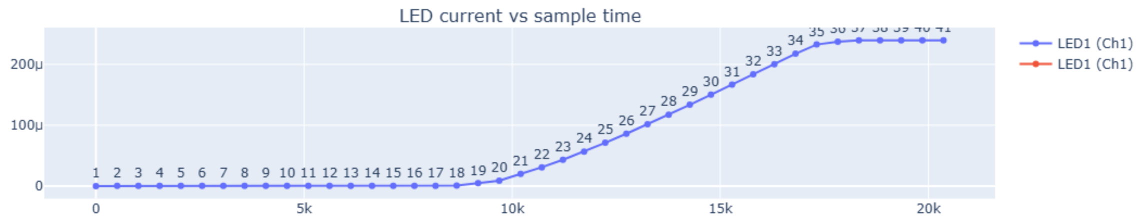
(continued from previous page)

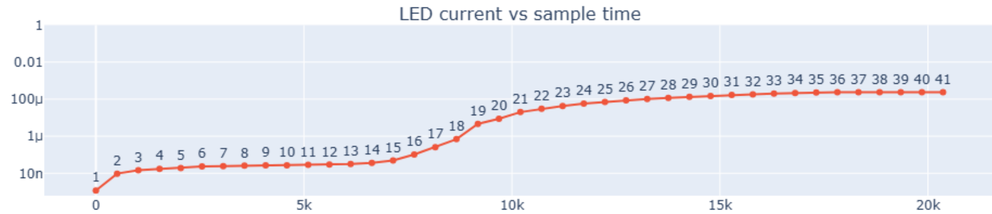
```
                            textposition="top center",
                            name='LED1 (Ch1)'),  row=1, col=1)
    fig.add_trace(go.Scatter(x=sweep.timecode, y=sweep.get_measurement_result("LED2"),
                            mode='lines+markers+text',
                            text = [str(i) for i in range(1, len(sample_times) + 1)],
                            textposition="top center",
                            name='LED1 (Ch1)'),  row=2, col=1)
    fig.update_yaxes(type="log", range=[np.log(0.0001),np.log(1)], row=2, col=1)
    fig.update_layout(height=600)
    return fig
fig = create_fig()
fig
#uncomment in pure python script:
#fig.show()
```



Once you have identified useful points for switching,
you can use the `change_current_range_at()` method to switch the range at these points.
First, we remove any current range switching from the configuration by calling `clear_current_ranges()`.
In this example the sweep starts with a range u 5uA, switches to 20uA at index 18uA and to 200uA at index 20.
As can be seen in the plots below, the range below 20 microamperes is now much better resolved than in the last sweep.

```
mbX1.set_voltages(1, channel_list)
mbX1.set_current_ranges(CurrentRange.Range_5uA, channel_list)
config_ch2.clear_current_ranges()
sweep.set_sample_count(1)
sweep.set_measurement_delay(500)
config_ch2.change_current_range_at(18, CurrentRange.Range_20uA_SMU)
config_ch2.change_current_range_at(20, CurrentRange.Range_200uA_SMU)
sweep.run()
fig = create_fig()
fig
#uncomment in pure python script:
#fig.show()
```



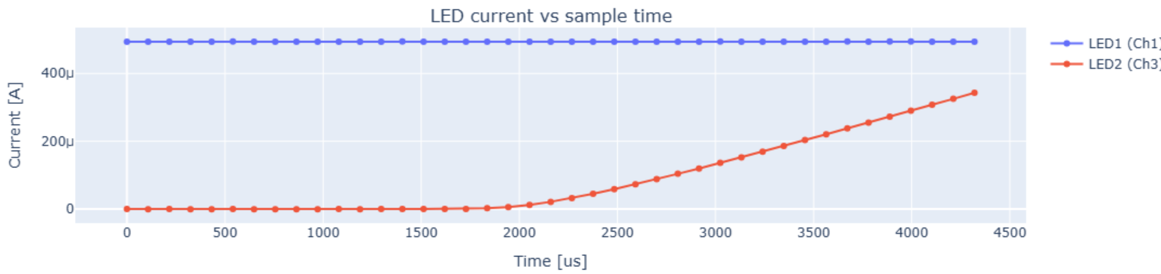


Constant force mode and sweep reusage

Sweeps can be repeated as often as required.
The parameters can remain the same or be changed between two runs.
This is illustrated by the following example where the last sweep is reused.
The first channel is set to 'constant force' mode.
In this mode, only measurements are made and the force value at the output is not varied.

```
config_ch1.set_constant_force_mode(number_of_steps=41)
sweep.run()
currents_LED1 = sweep.get_measurement_result("LED1")
currents_LED2 = sweep.get_measurement_result("LED2")
sample_times = sweep.timecode
```

```
fig = make_subplots(rows=1, cols=1,  subplot_titles=("LED current vs sample time",
↪"LED current vs voltage"))
fig.add_trace(go.Scatter(x=sample_times, y=currents_LED1,
                    mode='lines+markers',
                    name='LED1 (Ch1)'), row=1, col=1)
fig.add_trace(go.Scatter(x=sample_times, y=currents_LED2,
                    mode='lines+markers',
                    name='LED2 (Ch3)'), row=1, col=1)

fig.update_xaxes(title_text="Time [us]", row=1, col=1)
fig.update_yaxes(title_text="Current [A]", row=1, col=1)
fig
#uncomment in pure python script:
#fig.show()
```




```
srunner.shutdown()
```

3.8 Part 7: Triggering

This is the 7th introductory overview of programming the Aspect Device Engine Python API.
This document is available as pdf and interactive jupyter notebook. The introduction includes the following objectives:

TODO

```
srunner.shutdown()
```

3.9 Part 8a: Paramter Tables I

This is the 8th introductory overview of programming the Aspect Device Engine Python API.
This document is available as pdf and interactive jupyter notebook. The introduction includes the following objectives:

- How to load parameter setting tables
- Inspecting the contents of a table
- How to modify the parameters in the table
- Sending the parameters of a table to the hardware

3.9.1 Introduction

There are two basic options for programming the idSMU hardware:
Firstly, by calling object methods or setting properties of the hardware models, as described in the previous chapters.
The second method is a tabular method.
Here, tables are used to hold the values of the hardware parameters. These can be modified and applied.
The tabular method is slower than the direct method calls. However, if performance is not the highest priority,
this method can be used to export the state of the hardware and apply it again later.
Instead of dozens of method calls, just one is then sufficient to set all parameters at once.

```
from aspectdeviceengine.enginecore import IdSmuService, IdSmuSettingsService,␣
↪IdSmuServiceRunner, IdSmuBoardModel, IdSmuSettingsService
from aspectdeviceengine.enginecore import IdSmuBoardModel, IdqTable, IdqTableGroup
import plotly.graph_objects as go
```

(continues on next page)

(continued from previous page)

```
import pathlib,os
import numpy as np
srunner = IdSmuServiceRunner()
mbX1 : IdSmuBoardModel = srunner.get_idsmu_service().get_first_board()
```

The `IdSmuSettingsService` class is used to load and apply tables:

```
setting_service : IdSmuSettingsService = srunner.get_idsmu_service().get_settings_
↪service()
```

3.9.2 Import a table from csv

The parameter tables, also known as setting tables, have a special format.
If you open one of the CSV files with such a table, it looks like this, for example:

[M1_test]												
#SMU-Channel												
SettingId	Hardware	Name	Group	Type	OutputFo	ForceMod	EnableOu	CurrentRa	Measuren	EnableCla	ClampLow	ClampHigh\
0	M1.S1.C1	M1.S1.C1	All	SMU-Char	1	FV	1	70mA	HighZ	1	-0,07	0,07
0	M1.S1.C2	M1.S1.C2	All	SMU-Char	1	FV	1	70mA	HighZ	1	-0,07	0,07
0	M1.S1.C3	M1.S1.C3	All	SMU-Char	1	FV	0	70mA	HighZ	1	-0,07	0,07
0	M1.S1.C4	M1.S1.C4	All	SMU-Char	1	FV	0	70mA	HighZ	1	-0,07	0,07
[Types]												
int	string	string	string	string	float	[FV,FI,Hig	bool	[5uA,20uA	[ISense,V:	bool	float	float
#SMU-Device												
SettingId	Hardware	Name	Group	Type	INT10K	ExternalPower						
0	M1.S1	M1.S1	All	SMU-Devi	1	0						
[Types]												
int	string	string	string	string	bool	bool						

Firstly, the name of the table is given in square brackets [].
A table can consist of several groups. As the parameters for the idSMU device are different to the channel parameters, the parameters are each in their own group with their own column headers. The table name is followed by the group name (#).
This is followed by the column header of the group and then the actual data. This is followed by the entries for the data type or, in the case of enumeration types, the valid values for this parameter.
These entries are automatically appended when exporting a table - the user does not have to worry about them.
A table can be loaded with the `import_settings_from_csv()` method of the SettingService:

```
setting_service.import_settings_from_csv(os.path.join(os.path.abspath(""), "setting_
↪tables_m1.csv" ))
# List all parameter setting names
print(setting_service.get_parameter_settings_names())
# We only print 10 columns so that the output fits on the screen
print(setting_service.print_settings('M1_test', False, False, 10))
```

```
['M1_test']
Group name: SMU-Channel
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+
| SettingId | HardwareId | Name     | Group | Type        | OutputForceValue |␣
↪ForceMode | EnableOutput | CurrentRange | MeasurementMode |
+-----------+------------+----------+-------+-------------+------------------+-----
```

(continues on next page)

(continued from previous page)

```
↪------+--------------+--------------+-----------------+
| 0         | M1.S1.C1    | M1.S1.C1 | All    | SMU-Channel | 1                 | FV ␣
↪      | 1            | 70mA         | HighZ           |
| 0         | M1.S1.C2    | M1.S1.C2 | All    | SMU-Channel | 1                 | FV ␣
↪      | 1            | 70mA         | HighZ           |
| 0         | M1.S1.C3    | M1.S1.C3 | All    | SMU-Channel | 1                 | FV ␣
↪      | 0            | 70mA         | HighZ           |
| 0         | M1.S1.C4    | M1.S1.C4 | All    | SMU-Channel | 1                 | FV ␣
↪      | 0            | 70mA         | HighZ           |
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+

Group name: SMU-Device
+-----------+------------+-------+-------+------------+--------+---------------+
| SettingId | HardwareId | Name  | Group | Type       | INT10K | ExternalPower |
+-----------+------------+-------+-------+------------+--------+---------------+
| 0         | M1.S1      | M1.S1 | All   | SMU-Device | 1      | 0             |
+-----------+------------+-------+-------+------------+--------+---------------+
```

3.9.3 Modification of table entries

The entire table with its groups is managed by an actual table of the type `IdqTable`.
The `dqTableGroup` type represents a group.

```
paratable : IdqTable = setting_service.get_parameter_setting('M1_test')
group : IdqTableGroup = paratable.get_table_group('SMU-Channel')
```

To change a table entry, we need the row index and the column name of the cell.
The helper method `get_row_index()` is used to find the row index for a known entry in the table:

```
row_idx = [group.get_row_index('HardwareId', 'M1.S1.C1'), group.get_row_index(
↪'HardwareId', 'M1.S1.C2')]
print(row_idx)
```

```
[0, 1]
```

The *set_parameter_value()* method is now used to change a value in a table group.
All values in the table are of type string:

```
group.set_parameter_value(row_index=row_idx[0], parameter_name='OutputForceValue',␣
↪parameter_value='5')
group.set_parameter_value(row_index=row_idx[1], parameter_name='OutputForceValue',␣
↪parameter_value='6')
print(setting_service.print_settings('M1_test', False, False, 10))
```

```
Group name: SMU-Channel
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+
| SettingId | HardwareId | Name     | Group | Type        | OutputForceValue |␣
↪ForceMode | EnableOutput | CurrentRange | MeasurementMode |
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+
| 0         | M1.S1.C1    | M1.S1.C1 | All    | SMU-Channel | 5                 | FV ␣
↪      | 1            | 70mA         | HighZ           |
```

(continues on next page)

(continued from previous page)

```
| 0          | M1.S1.C2   | M1.S1.C2 | All   | SMU-Channel | 6                | FV ␣
↪      | 1            | 70mA          | HighZ           |
| 0          | M1.S1.C3   | M1.S1.C3 | All   | SMU-Channel | 1                | FV ␣
↪      | 0            | 70mA          | HighZ           |
| 0          | M1.S1.C4   | M1.S1.C4 | All   | SMU-Channel | 1                | FV ␣
↪      | 0            | 70mA          | HighZ           |
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+

Group name: SMU-Device
+-----------+------------+-------+-------+------------+--------+---------------+
| SettingId | HardwareId | Name  | Group | Type       | INT10K | ExternalPower |
+-----------+------------+-------+-------+------------+--------+---------------+
| 0         | M1.S1      | M1.S1 | All   | SMU-Device | 1      | 0             |
+-----------+------------+-------+-------+------------+--------+---------------+
```

3.9.4 Writing the parameters

The `apply_parameter_setting()` method can now be used to write the parameters of a group or the entire table to the hardware.
We then examine a few channel parameters:

```
setting_service.apply_parameter_setting(setting_name='M1_test', board_address='M1',␣
↪filtered=False, table_group_name='SMU-Channel')
print(mbX1.idSmu2Modules['M1.S1'].smu.channels['M1.S1.C1'].enabled)
print(mbX1.idSmu2Modules['M1.S1'].smu.channels['M1.S1.C1'].voltage)
print(mbX1.idSmu2Modules['M1.S1'].smu.channels['M1.S1.C2'].enabled)
print(mbX1.idSmu2Modules['M1.S1'].smu.channels['M1.S1.C2'].voltage)
```

```
True
4.998809814453125
True
5.999847412109375
```

```
group.at[row_idx[0], 'OutputForceValue'] = "0"
group.at[row_idx[1], 'EnableOutput'] = "0"
print(setting_service.print_settings('M1_test', False, False, 10))
setting_service.apply_parameter_setting('M1_test', 'M1', False, 'SMU-Channel')
```

```
Group name: SMU-Channel
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+
| SettingId | HardwareId | Name     | Group | Type        | OutputForceValue |␣
↪ForceMode | EnableOutput | CurrentRange | MeasurementMode |
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+
| 0          | M1.S1.C1   | M1.S1.C1 | All   | SMU-Channel | 0                | FV ␣
↪      | 1            | 70mA          | HighZ           |
| 0          | M1.S1.C2   | M1.S1.C2 | All   | SMU-Channel | 6                | FV ␣
↪      | 0            | 70mA          | HighZ           |
| 0          | M1.S1.C3   | M1.S1.C3 | All   | SMU-Channel | 1                | FV ␣
↪      | 0            | 70mA          | HighZ           |
| 0          | M1.S1.C4   | M1.S1.C4 | All   | SMU-Channel | 1                | FV ␣
↪      | 0            | 70mA          | HighZ           |
```

(continues on next page)

(continued from previous page)

```
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+-------------+-------------+-----------------+

Group name: SMU-Device
+-----------+------------+-------+-------+------------+--------+---------------+
| SettingId | HardwareId | Name  | Group | Type       | INT10K | ExternalPower |
+-----------+------------+-------+-------+------------+--------+---------------+
| 0         | M1.S1      | M1.S1 | All   | SMU-Device | 1      | 0             |
+-----------+------------+-------+-------+------------+--------+---------------+
```

```
4
```

```
srunner.shutdown()
```

3.10 Part 8b: Paramter Tables II

This is the 8th introductory overview of programming the Aspect Device Engine Python API.
This document is available as pdf and interactive jupyter notebook. The introduction includes the following objectives:

- How to export the state of the hardware into a parameter table file
- Filtering

3.10.1 Introduction

The basics of the parameter tables were presented in the last chapter.
Further options will now be discussed.

```
from aspectdeviceengine.enginecore import IdSmuService, IdSmuSettingsService,␣
↪IdSmuServiceRunner, IdSmuBoardModel, IdSmuSettingsService
from aspectdeviceengine.enginecore import IdSmuBoardModel, IdqTable, IdqTableGroup
import plotly.graph_objects as go
import pathlib,os
import numpy as np
srunner = IdSmuServiceRunner()
mbX1 : IdSmuBoardModel = srunner.get_idsmu_service().get_first_board()
```

The `IdSmuSettingsService` class is used to load and apply tables:

```
setting_service : IdSmuSettingsService = srunner.get_idsmu_service().get_settings_
↪service()
```

3.10.2 Exporting a table

The Methode get_parameter_settings_for_board() is used to read out the current status of the hardware and generate a parameter table.
We first change a few parameters and then analyse the table:

```
mbX1.idSmu2Modules['M1.S1'].smu.channels['M1.S1.C1'].voltage = 1
mbX1.idSmu2Modules['M1.S1'].smu.channels['M1.S1.C2'].voltage = 2
mbX1.idSmu2Modules['M1.S1'].smu.channels['M1.S1.C3'].voltage = 3
mbX1.idSmu2Modules['M1.S1'].smu.channels['M1.S1.C4'].voltage = 4
mbX1_settings : IdqTable = setting_service.get_parameter_settings_for_board('M1')
print(mbX1_settings.name)
print(setting_service.print_settings(mbX1_settings.name, False, False, 10))
```

```
M1
Group name: SMU-Channel
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+
| SettingId | HardwareId | Name     | Group | Type        | OutputForceValue |␣
↪ForceMode | EnableOutput | CurrentRange | MeasurementMode |
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+
| 0         | M1.S1.C1   | M1.S1.C1 | All   | SMU-Channel | 1,000000         | FV ␣
↪       | 0            | 70mA         | HighZ           |
| 0         | M1.S1.C2   | M1.S1.C2 | All   | SMU-Channel | 2,000000         | FV ␣
↪       | 0            | 70mA         | HighZ           |
| 0         | M1.S1.C3   | M1.S1.C3 | All   | SMU-Channel | 3,000000         | FV ␣
↪       | 0            | 70mA         | HighZ           |
| 0         | M1.S1.C4   | M1.S1.C4 | All   | SMU-Channel | 4,000000         | FV ␣
↪       | 0            | 70mA         | HighZ           |
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+

Group name: SMU-Device
+-----------+------------+-------+-------+------------+--------+---------------+
| SettingId | HardwareId | Name  | Group | Type       | INT10K | ExternalPower |
+-----------+------------+-------+-------+------------+--------+---------------+
| 0         | M1.S1      | M1.S1 | All   | SMU-Device | 1      | 0             |
+-----------+------------+-------+-------+------------+--------+---------------+
```

Before saving the table, it is advisable to change the name of the table.
The default name is the board address.
The name of a table to be loaded later should therefore be different from this default value so that this table is not overwritten by the method executed above.

```
mbX1_settings.name = "M1_voltages_set"
print(mbX1_settings.name)
print(setting_service.get_parameter_settings_names())
```

```
M1_voltages_set
['M1_voltages_set']
```

The table is now exported using the `export_settings_to_csv()` method. The three parameters of the method are self-explanatory.
The reason why `setting_names` is a list is that it is also possible to write several tables to one file.
In the example here, we only write the table just created to the file:

```
setting_service.export_settings_to_csv(file_path=os.path.join(os.path.abspath(""),
                                        mbX1_settings.name + ".csv"), setting_names=[
↪'M1_voltages_set'],
                                        append_to_file=False)
```

The result looks like this:

[M1_voltages_set]												
#SMU-Channel												
SettingId	Hardware	Name	Group	Type	OutputForceValue	ForceMod	EnableOu	CurrentRa	Measuren	EnableCla	ClampLow	ClampHighValue
0	M1.S1.C1	M1.S1.C1	All	SMU-Char	1	FV	0	70mA	HighZ	1	-0,07	0,07
0	M1.S1.C2	M1.S1.C2	All	SMU-Char	2	FV	0	70mA	HighZ	1	-0,07	0,07
0	M1.S1.C3	M1.S1.C3	All	SMU-Char	3	FV	0	70mA	HighZ	1	-0,07	0,07
0	M1.S1.C4	M1.S1.C4	All	SMU-Char	4	FV	0	70mA	HighZ	1	-0,07	0,07
[Types]												
int	string	string	string	string	float	[FV,FI,Hig	bool	[5uA,20uA	[ISense,V:	bool	float	float
#SMU-Device												
SettingId	Hardware	Name	Group	Type	INT10K	ExternalPower						
0	M1.S1	M1.S1	All	SMU-Devi	1	0						
[Types]												
int	string	string	string	string	bool	bool						

3.10.3 Filtering and applying sub-tables

Previously, all entries in a table or group were always sent to the hardware.
But what if you only want to use certain entries?
There are various filter methods. You can then send only the filtered table to the hardware instead of the entire table.

Row filter

The parameters of the `filter_rows()` method are the column name, the value in this column and whether to search for whole words and not just substrings (exact_match).
To print the filtered version of a table, the second parameter in `the print_settings()` method is set to True.

```
filtered_table = mbX1_settings.filter_rows(column_name='HardwareId', filter_value='M1.
↪S1.C2', exact_match=False)
print(setting_service.print_settings('M1_voltages_set', True, False, 10))
```

```
Group name: SMU-Channel
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+
| SettingId | HardwareId | Name     | Group | Type        | OutputForceValue |␣
↪ForceMode | EnableOutput | CurrentRange | MeasurementMode |
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+
| 0         | M1.S1.C2   | M1.S1.C2 | All   | SMU-Channel | 2,000000         | FV ␣
↪      | 0            | 70mA         | HighZ           |
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+
```

The pipe character (|) can be used to realise a logical 'or':

```
filtered_table = mbX1_settings.filter_rows(column_name='HardwareId', filter_value='M1.
↪S1.C2|M1.S1.C4', exact_match=False)
print(setting_service.print_settings('M1_voltages_set', True, False, 10))
```

```
Group name: SMU-Channel
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+
| SettingId | HardwareId | Name     | Group | Type        | OutputForceValue |␣
↪ForceMode | EnableOutput | CurrentRange | MeasurementMode |
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+
| 0         | M1.S1.C2   | M1.S1.C2 | All   | SMU-Channel | 2,000000         | FV ␣
↪      | 0            | 70mA         | HighZ           |
| 0         | M1.S1.C4   | M1.S1.C4 | All   | SMU-Channel | 4,000000         | FV ␣
↪      | 0            | 70mA         | HighZ           |
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+
```

If the 'filtered' flag of the 'apply_parameter_setting()' method is set,
only the filtered version is written to the hardware instead of the entire table:

```
mbX1.idSmu2Modules['M1.S1'].smu.channels['M1.S1.C1'].voltage = 5
mbX1.idSmu2Modules['M1.S1'].smu.channels['M1.S1.C2'].voltage = 5
mbX1.idSmu2Modules['M1.S1'].smu.channels['M1.S1.C3'].voltage = 5
mbX1.idSmu2Modules['M1.S1'].smu.channels['M1.S1.C4'].voltage = 5
setting_service.apply_parameter_setting(setting_name='M1_voltages_set', board_address=
↪'M1', filtered=True, table_group_name='SMU-Channel')
```

```
2
```

We export the status of the hardware to a new table. Now only the values of the filtered table should have been written. The voltages of channels 1 and 3 were set to 5 volts. The remaining channels retain their values.

```
mbX1_settings : IdqTable = setting_service.get_parameter_settings_for_board('M1')
print(setting_service.print_settings(mbX1_settings.name, False, False, 10))
```

```
Group name: SMU-Channel
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+
| SettingId | HardwareId | Name     | Group | Type        | OutputForceValue |␣
↪ForceMode | EnableOutput | CurrentRange | MeasurementMode |
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+
| 0         | M1.S1.C1   | M1.S1.C1 | All   | SMU-Channel | 5,000000         | FV ␣
↪      | 0            | 70mA         | HighZ           |
| 0         | M1.S1.C2   | M1.S1.C2 | All   | SMU-Channel | 2,000000         | FV ␣
↪      | 0            | 70mA         | HighZ           |
| 0         | M1.S1.C3   | M1.S1.C3 | All   | SMU-Channel | 5,000000         | FV ␣
↪      | 0            | 70mA         | HighZ           |
| 0         | M1.S1.C4   | M1.S1.C4 | All   | SMU-Channel | 4,000000         | FV ␣
↪      | 0            | 70mA         | HighZ           |
+-----------+------------+----------+-------+-------------+------------------+-----
↪------+--------------+--------------+-----------------+

Group name: SMU-Device
+-----------+------------+-------+-------+------------+--------+---------------+
| SettingId | HardwareId | Name  | Group | Type       | INT10K | ExternalPower |
+-----------+------------+-------+-------+------------+--------+---------------+
| 0         | M1.S1      | M1.S1 | All   | SMU-Device | 1      | 0             |
+-----------+------------+-------+-------+------------+--------+---------------+
```

```
srunner.shutdown()
```