MODULAB

Release 0.1.1

Silas Hörz

Nov 18, 2025

CONTENTS:

CHAPTER

ONE

LOG MANAGER

class modules.log.LogManager.**LogManager**

Bases: QObject

Verwaltet das Logging für die gesamte Anwendung.

Diese Klasse ist als zentraler Dienst konzipiert. Sie erfüllt zwei Hauptaufgaben:

1. **Datei-Logging:** Schreibt alle Logs (DEBUG, INFO, WARNING, ERROR) in eine zeitgestempelte .log-Datei im Benutzerverzeichnis (*~/Modulab/Logs*) unter Verwendung des standard *logging*-Moduls.
2. **GUI-Benachrichtigung:** Speichert Logs im Speicher (in-memory) und löst das *message_logged*-Signal für jeden neuen Eintrag aus, um UIs (wie ein Log-Widget) in Echtzeit zu aktualisieren.

Sie wird typischerweise einmal erstellt und an alle anderen Manager übergeben.

Signale:

message_logged (dict):
Wird für jeden Log-Eintrag (Info, Error, etc.) ausgelöst. Das übergebene Wörterbuch (dict) hat die Struktur: *{'timestamp': datetime.datetime, 'type': str, 'message': str}*

INFO = 'INFO'

Konstante für den 'INFO'-Log-Level.

Type
str

WARNING = 'WARNING'

Konstante für den 'WARNING'-Log-Level.

Type
str

ERROR = 'ERROR'

Konstante für den 'ERROR'-Log-Level.

Type
str

DEBUG = 'DEBUG'

Konstante für den 'DEBUG'-Log-Level.

Type
str

__init__()

Initialisiert den LogManager.

Erstellt das Log-Verzeichnis (*~/Modulab/Logs*), falls es nicht existiert, und konfiguriert den Python *logging*-Handler für das zeitgestempelte Sitzungs-Logfile.

Raises
RuntimeError – Wenn das Log-Verzeichnis nicht erstellt oder beschrieben werden kann.

info(*message*)

Loggt eine Info-Nachricht.

Parameters
message (str) – Die zu loggende Nachricht.

Examples

Eine einfache Statusmeldung loggen:

```
# Annahme: 'log_mgr' ist eine Instanz von LogManager
log_mgr.info("Verbindung zum Gerät erfolgreich hergestellt.")
```

warning(*message*)

Loggt eine Warnung.

Parameters
message (str) – Die zu loggende Warnung.

Examples

Eine Warnung für einen ungültigen Wert loggen:

```
log_mgr.warning("Integrationszeit auf 0 gesetzt. Ignoriere Befehl.")
```

error(*message*, *exc_info=True*)

Loggt einen Fehler.

Standardmäßig wird die Ausnahme-Info (Stack Trace) mitgeloggt, wenn diese Funktion innerhalb eines *except*-Blocks aufgerufen wird.

Parameters

- **message** (str) – Die zu loggende Fehlermeldung.
- **exc_info** (bool, *optional*) – Wenn True (Standard), wird der Stack Trace automatisch mitgeloggt. Setzen Sie auf False, um dies zu unterdrücken.

Examples

Einen einfachen Fehler loggen (ohne Stack Trace):

```
if not manager.is_connected():
    log_mgr.error("Verbindung fehlgeschlagen. Kein Stack Trace.", exc_
↪info=False)
```

Einen Fehler innerhalb einer Ausnahmebehandlung loggen (mit Stack Trace):

```
try:
    # Code, der fehlschlagen könnte
    result = 10 / 0
except ZeroDivisionError as e:
    # 'exc_info=True' ist Standard, aber hier zur Verdeutlichung.
    # 'e' wird automatisch vom Logger erfasst.
    log_mgr.error(f"Schwerer Rechenfehler: {e}", exc_info=True)
```

debug(*message*)

Loggt eine Debug-Nachricht.

Diese Nachrichten sind oft sehr detailliert und nur für die Fehlersuche gedacht.

Parameters

message (str) – Die zu loggende Debug-Nachricht.

Examples

Messwerte oder Zwischenschritte loggen:

```
log_mgr.debug(f"Spektrum aufgenommen. {len(intensities)} Datenpunkte.")
```

get_all_messages()

Gibt die komplette Liste aller Log-Einträge der aktuellen Sitzung zurück.

Nützlich, um ein Log-Fenster beim Öffnen zu initialisieren.

Returns

Eine Liste von Log-Einträgen. Jedes dict hat die
Struktur: *{'timestamp': datetime, 'type': str, 'message': str}*

Return type
list[dict]

Examples

Alle bisherigen Logs beim Start eines Widgets laden:

```
alle_logs = log_gmr.get_all_messages()
for eintrag in alle_logs:
    # z.B. in eine QListWidget einfügen
    print(f"[{eintrag['type']}] {eintrag['message']}")
```

get_latest_message()

Gibt nur den letzten Log-Eintrag zurück. (Für bspw. Status Label)

Returns

Der letzte Log-Eintrag als dict mit der Struktur
{'timestamp': ..., 'type': ..., 'message': ...} oder *None*, wenn noch keine Logs vorhanden sind.

Return type
dict | None

Examples

Den Text für eine Statusleiste setzen:

```
letzte_meldung = log_mgr.get_latest_message()
if letzte_meldung:
    status_bar.showMessage(letzte_meldung['message'])
```

CHAPTER

TWO

PROFILE MANAGER

class modules.profile.ProfileManager.**ProfileManager**(*log_manager*)

Bases: QObject

Erstellt, speichert und verwaltet Benutzerprofile (Key-Value-Speicher).

Diese Klasse verwaltet app-weite Einstellungen, indem sie diese in *.json*-Dateien im Benutzerverzeichnis (*~/Modulab/Profiles*) speichert. Jede *.json*-Datei repräsentiert ein "Profil" (z.B. "Experiment_A", "Default").

Sie dient als zentraler Key-Value-Speicher für alle anderen Manager (z.B. zum Speichern der letzten Integrationszeit oder des letzten verbundenen Geräts).

Parameters
log_manager (LogManager) – Eine Instanz des LogManagers für das Logging.

Signale:

profile_loaded (str):
Wird ausgelöst, nachdem *load_profile* erfolgreich war. Args: (str: Der Name des geladenen Profils).

CONFIG_FILE_NAME = 'config.json'

Interner Dateiname für die Manager-Konfiguration (z.B. letztes Profil).

Type
str

KEY_LAST_PROFILE = 'last_profile_name'

Interner Schlüssel zum Speichern des Namens des zuletzt geladenen Profils.

Type
str

__init__(*log_manager*)

Initialisiert den ProfileManager.

Erstellt das Profil-Verzeichnis (*~/Modulab/Profiles*), falls es nicht existiert.

Parameters
log_manager (LogManager) – Die LogManager-Instanz.

create_profile(*profile_name*, *data=None*)

Erstellt eine neue, leere Profildatei (.json).

Wenn *data* angegeben wird, wird das Profil mit diesen Anfangswerten erstellt.

Parameters

- **profile_name** (str) – Der Name für das neue Profil (ohne .json).

- **data** (dict, *optional*) – Ein Wörterbuch mit Anfangsdaten für das Profil.

Returns

True bei Erfolg, False, wenn das Profil bereits existiert
oder ein Fehler auftritt.

Return type
bool

Examples

Ein neues, leeres Profil "Experiment_A" erstellen:

```
profile_mgr.create_profile("Experiment_A")
```

Ein Profil mit Standard-Einstellungen für ein Spektrometer erstellen:

```
default_settings = {
    "Spec_integration_time_us": 10000,
    "Spec_correct_dark_counts": False
}
profile_mgr.create_profile("Default_Spektrometer", data=default_settings)
```

load_profile(*profile_name*)

Lädt ein Profil in den Speicher und macht es zum "aktuellen" Profil.

Alle zukünftigen *read()*- und *write()*-Operationen beziehen sich auf dieses Profil. Setzt auch dieses Profil als "zuletzt verwendet".

Parameters
profile_name (str) – Der Name des zu ladenden Profils.

Returns
True bei Erfolg, False, wenn das Profil nicht gefunden wurde.

Return type
bool

Examples

Das "Default" Profil beim Start laden:

```
if not profile_mgr.load_profile("Default"):
    profile_mgr.create_profile("Default")
    profile_mgr.load_profile("Default")
```

delete_profile(*profile_name*)

Löscht eine Profildatei (.json) vom Datenträger.

Parameters
profile_name (str) – Der Name des zu löschenden Profils.

Returns

True bei Erfolg, False, wenn die Datei nicht existiert
oder ein Fehler auftritt.

Return type
bool

`list_profiles()`

Listet alle verfügbaren Profile (alle .json-Dateien) im Profilordner auf.

Returns
Eine alphabetisch sortierte Liste aller Profilnamen.

Return type
list[str]

Examples

Eine QComboBox mit allen Profilen füllen:

```
profil_liste = profile_mgr.list_profiles()
ui.profile_combobox.clear()
ui.profile_combobox.addItems(profil_liste)
```

`write(key, value)`

Schreibt ein Key-Value-Paar in das *aktuell geladene* Profil.

Dies ist die Hauptmethode für andere Manager, um ihre Einstellungen zu speichern.

Parameters

- **key** (str) – Der Einstellungs-Schlüssel (z.B. “Spec_integration_time_us”).
- **value** (any) – Der zugehörige Wert (muss JSON-serialisierbar sein).

Returns
True bei Erfolg, False, wenn kein Profil geladen ist.

Return type
bool

Examples

Einstellungen von anderen Managern speichern:

```
# Im SpectrometerManager (nach Änderung der Integrationszeit)
time_us = 100000
profile_mgr.write("Spec_integration_time_us", time_us)

# Im SmuManager (nach Verbindung)
port = "COM3"
profile_mgr.write("Smu_LastDevice", port)
```

`read(key)`

Liest einen Wert anhand des ‘key’ aus dem *aktuell geladenen* Profil.

Dies ist die Hauptmethode für andere Manager, um ihre Einstellungen zu laden.

Parameters
key (str) – Der Einstellungs-Schlüssel (z.B. “Spec_integration_time_us”).

Returns

Der gespeicherte Wert, oder *None*, wenn der Schlüssel
nicht existiert oder kein Profil geladen ist.

Return type
any | None

Examples

Einstellungen in anderen Managern laden:

```
# Im SpectrometerManager (während __init__)
# Lade gespeicherte Zeit, oder nutze 100ms als Standard
default_time = 100 * 1000
time_us = profile_mgr.read("Spec_integration_time_us")

if time_us is None:
    time_us = default_time

self.set_integrationtime(time_us)
```

`get_current_profile_name()`

Gibt den Namen des aktuell geladenen Profils zurück.

Returns
Der Name des Profils, oder *None*.

Return type
str | None

`get_last_profile_name()`

Liest aus der *config.json*, welches Profil zuletzt geladen wurde.

Prüft gleichzeitig, ob dieses Profil noch existiert.

Returns
Der Name des letzten Profils, oder *None*.

Return type
str | None

Examples

Beim App-Start das letzte Profil automatisch laden:

```
last_profile = profile_mgr.get_last_profile_name()
if last_profile:
    profile_mgr.load_profile(last_profile)
else:
    # Lade Fallback oder zeige Profil-Auswahl
    profile_mgr.load_profile("Default")
```

CHAPTER

THREE

DEVICE MANAGER

class modules.device.DeviceManager.**Device**(*name*, *geometry*, *tags=None*, ***dimensions*)

Bases: `object`

Repräsentiert ein einzelnes physisches Device (z.B. Pixel, Teststruktur).

Dies ist eine Daten-Container-Klasse, die alle geometrischen und metadatenbezogenen Eigenschaften eines Devices speichert.

__init__(*name*, *geometry*, *tags=None*, ***dimensions*)

Initialisiert ein neues Device-Objekt.

Parameters

- **name** (`str`) – Der eindeutige Name des Devices (z.B. “Pixel_R1C1”).
- **geometry** (`str`) – Der Geometrie-Typ. Muss ‘rectangle’ oder ‘circle’ sein.
- **tags** (`list`, *optional*) – Eine Liste von String-Tags zur Filterung.
- ****dimensions** (`float`) – Dynamische Schlüsselwortargumente für die Maße des Devices in Metern [m].
  - Für ‘rectangle’: *length*, *width*
  - Für ‘circle’: *radius*
  - Optionale Cutouts: *cutout_length*, *cutout_width*, *cutout_radius*.

Examples

Ein rechteckiges Device (1mm x 0.5mm) erstellen:

```
dev1 = Device(
    name="Pixel_1",
    geometry="rectangle",
    tags=["OLED", "Red"],
    length=1e-3,
    width=0.5e-3
)
```

Ein kreisförmiges Device (Radius 80µm) mit einem kreisförmigen Ausschnitt (Radius 10µm) erstellen:

```
dev2 = Device(
    name="Ring_Struktur",
    geometry="circle",
    radius=80e-6,
    cutout_radius=10e-6
)
```

```
```

get_area()

Berechnet die aktive Fläche [m^2] des Devices.

Die Fläche wird als *Hauptfläche - Ausschnittfläche* berechnet, basierend auf der Geometrie und den *dimensions*.

Hinweis:

- Wenn die Maße ungültig (nicht numerisch) sind, wird 0.0 zurückgegeben.
- Wenn die Ausschnittfläche größer als die Hauptfläche ist, wird 0.0 zurückgegeben.

Returns
Die berechnete aktive Fläche in Quadratmetern [m^2].

Return type
float

to_dict()

Konvertiert das Device-Objekt in ein serialisierbares Diktat.

Wird verwendet, um das Device im ProfileManager zu speichern.

Returns
Eine Diktat-Repräsentation des Devices.

Return type
dict

classmethod from_dict(*data*)

Erstellt eine Device-Instanz aus einem Diktat.

Wird verwendet, um das Device aus dem ProfileManager zu laden.

Parameters
data (dict) – Das Diktat, das *to_dict()* erzeugt hat.

Returns
Eine neue Instanz der Device-Klasse.

Return type
Device

__repr__()

Stellt eine formale String-Repräsentation des Objekts bereit.

class modules.device.DeviceManager.**DeviceManager**(*log_manager=None*, *profile_manager=None*, *parent=None*)

Bases: QObject

Erstellt, bearbeitet, löscht und verwaltet *Device*-Objekte.

Diese Klasse dient als zentraler Service für die Verwaltung einer Liste von Devices. Sie ist verantwortlich für: - Das Erstellen, Bearbeiten und Löschen von *Device*-Objekten. - Das Speichern und Laden der Device-Liste im/aus dem *ProfileManager*. - Die Verwaltung, welches Device das "aktive" ist.

Parameters

- **log_manager** (LogManager) – Eine Instanz des LogManagers.

- **profile_manager** (ProfileManager) – Eine Instanz des ProfileManagers.
- **parent** (QObject, *optional*) – Ein übergeordnetes QObject für das Speichermanagement.

Signale:

device_loaded (str):
Wird ausgelöst, wenn ein Device mit *set_active_device* als aktiv ausgewählt wurde. Args: (str: Der Name des geladenen Devices).

`KEY_DEVICE_LIST = 'devices'`

Der Profil-Schlüssel zum Speichern der Liste aller Device-Diktate.

Type
str

`KEY_ACTIVE_DEVICE = 'active_device_name'`

Der Profil-Schlüssel zum Speichern des Namens des aktiven Devices.

Type
str

`__init__`(*log_manager=None*, *profile_manager=None*, *parent=None*)

Initialisiert den DeviceManager.

Parameters

- **log_manager** (LogManager, *optional*) – Die LogManager-Instanz.
- **profile_manager** (ProfileManager, *optional*) – Die ProfileManager-Instanz.
- **parent** (QObject, *optional*) – Ein übergeordnetes QObject.

`load_from_profile`()

Lädt die Device-Liste und das aktive Device aus dem geladenen Profil.

Diese Funktion wird typischerweise aufgerufen, nachdem der ProfileManager das *profile_loaded*-Signal gesendet hat.

`create_device`(*name*, *geometry*, *tags=None*, ***dimensions*)

Erstellt ein neues Device, fügt es zur Liste hinzu und speichert es im Profil.

Parameters

- **name** (str) – Der eindeutige Name des Devices (z.B. “Pixel_R1C1”).
- **geometry** (str) – Der Geometrie-Typ (‘rectangle’ oder ‘circle’).
- **tags** (list, *optional*) – Eine Liste von String-Tags.
- ****dimensions** (float) – Dynamische Schlüsselwortargumente für die Maße des Devices in Metern [m] (z.B. *length=1e-3*).

Returns
True bei Erfolg, False, wenn der Name bereits existiert.

Return type
bool

Examples

Ein rechteckiges Device (1mm x 0.5mm) erstellen:

```
# Annahme: 'device_mgr' ist eine Instanz von DeviceManager
device_mgr.create_device(
    "OLED_Pixel_1",
    "rectangle",
    length=1e-3,
    width=0.5e-3
)
```

Ein kreisförmiges Device (Radius 80µm) mit Tags erstellen:

```
device_mgr.create_device(
    "Pin80",
    "circle",
    tags=["Test_Struktur", "Rund"],
    radius=80e-6
)
```

delete_device(*name*)

Löscht ein Device anhand seines Namens aus der Liste.

Parameters

name (str) – Der Name des zu löschenden Devices.

Returns

True bei Erfolg, False, wenn das Device nicht gefunden wurde.

Return type

bool

edit_device(*name*, *new_geometry=None*, *new_tags=None*, *new_dimensions=None*)

Bearbeitet ein existierendes Device.

Der Name kann nicht geändert werden. Nur die übergebenen (nicht-None) Parameter werden aktualisiert.

Parameters

- **name** (str) – Der Name des zu bearbeitenden Devices.
- **new_geometry** (str, *optional*) – Die neue Geometrie ('rectangle'/'circle').
- **new_tags** (list, *optional*) – Die *komplett neue* Liste von Tags.
- **new_dimensions** (dict, *optional*) – Das *komplett neue* Diktat für Dimensionen.

Returns

True bei Erfolg, False, wenn das Device nicht gefunden wurde.

Return type

bool

Examples

Die Dimensionen eines Devices ändern:

```
neue_maße = {'length': 1.1e-3, 'width': 0.6e-3}
device_mgr.edit_device("OLED_Pixel_1", new_dimensions=neue_maße)
```

Nur die Tags eines Devices ändern:

```
device_mgr.edit_device("Pin80", new_tags=["Test_Struktur", "Wichtig"])
```

get_device_by_name(*name*)

Sucht und retourniert das Device-Objekt anhand des Namens.

Parameters
name (str) – Der Name des gesuchten Devices.

Returns
Das *Device*-Objekt oder *None*, wenn nicht gefunden.

Return type
Device | None

list_device_names()

Gibt eine Liste aller Device-Namen zurück.

Nützlich für die Anzeige in einer Combobox oder Liste in der GUI.

Returns
Eine Liste der Namen aller geladenen Devices.

Return type
list[str]

Examples

Eine QComboBox mit allen Device-Namen füllen:

```
namen = device_mgr.list_device_names()
ui.device_combobox.clear()
ui.device_combobox.addItems(namen)
```

set_active_device(*name*)

Setzt das aktive Device für die Anwendung.

Löst das *device_loaded*-Signal aus.

Parameters
name (str) – Der Name des Devices, das aktiv werden soll.

Returns
True bei Erfolg, False, wenn das Device nicht gefunden wurde.

Return type
bool

get_active_device()

Gibt das *gesamte Objekt* des aktiven Devices zurück.

Die UI kann dann *device.get_area()* oder *device.dimensions* selbst aufrufen.

Returns
Das aktuell aktive *Device*-Objekt oder *None*.

Return type
Device | None

Examples

Informationen des aktiven Devices abrufen:

```
aktives_gerät = device_mgr.get_active_device()
if aktives_gerät:
    print(f"Aktuell: {aktives_gerät.name}")
    print(f"Fläche: {aktives_gerät.get_area()} m²")
else:
    print("Kein Device aktiv.")
```

`get_active_device_area()`

Bequemlichkeitsfunktion: Gibt die Fläche [m^2] des aktiven Devices zurück.

Returns
Die Fläche in m^2 oder *None*, wenn kein Device aktiv ist.

Return type
float | None

`get_active_device_dimensions()`

Bequemlichkeitsfunktion: Gibt die Maße des aktiven Devices zurück.

Returns

Das Dimensions-Wörterbuch (z.B. *{'length': 0.1, ...}*)
oder ein leeres dict.

Return type
dict

CHAPTER

FOUR

SMU MANAGER

class modules.smu.SmuManager.**SmuManager**(*log_manager*, *profile_manager*)

Bases: QObject

Manager zur Steuerung und Verwaltung von SMU-Geräten (Source Measure Units).

Diese Klasse kapselt die Gerätetreiber (z.B. Keithley2602), verwaltet die serielle Verbindung, aktualisiert die Geräteliste und stellt High-Level-Methoden für die Konfiguration (Source, Sense, Limits) und Messung (IV) bereit. Sie nutzt PySide6-Signale, um die GUI über Statusänderungen zu informieren.

Parameters

- **log_manager** (LogManager) – Eine Instanz eines Log-Managers (erwartet .info, .error, etc.).
- **profile_manager** (ProfileManager) – Eine Instanz zur Verwaltung von App-Einstellungen (Lesen/Schreiben).

Signale:

connection_status_changed (bool, str):
Wird ausgelöst, wenn sich der Verbindungsstatus ändert. Args: (bool: verbunden, str: Gerätename/IDN).

device_list_updated (list):
Wird ausgelöst, nachdem die Liste der seriellen Ports aktualisiert wurde. Args: (list: Liste von Port-Namen [str]).

new_measurement_acquired (str, float, float):
Wird ausgelöst, wenn eine neue Messung verfügbar ist. Args: (str: Kanal, float: Strom, float: Spannung).

get_deviceList()

Scannt nach verfügbaren seriellen Ports und aktualisiert die interne Liste.

Sucht nach allen COM-Ports und fügt zusätzlich einen "DUMMY"-Port für Testzwecke hinzu. Löst das *device_list_updated*-Signal aus.

Returns
Eine Liste der gefundenen Port-Namen (z.B. ['COM1', 'COM3', 'DUMMY']).

Return type
list

connect(*port_name*)

Verbindet eine SMU an einem bestimmten COM-Port. ...

Parameters
port_name (str) – Der Name des Ports (z.B. “COM1” oder “DUMMY”).

Returns
True bei erfolgreicher Verbindung, sonst False.

Return type
bool

Examples

Mit einem echten COM-Port verbinden:

```
# 'COM3' ist nur ein Beispiel
success = manager.connect('COM3')
if success:
    print("Verbunden!")
```

Mit dem DUMMY-Gerät für Tests verbinden:

```
manager.connect('DUMMY')
```

`connect_LastDevice()`

Versucht, die Verbindung mit dem zuletzt genutzten Gerät wiederherzustellen.

Aktualisiert zuerst die Geräteliste und prüft, ob der gespeicherte Port (*self.LastDevice*) verfügbar ist.

Returns

True bei Erfolg, False, wenn kein Gerät gespeichert war
oder die Verbindung fehlschlägt.

Return type
bool

`disconnect()`

Trennt die aktive Verbindung zum SMU-Gerät.

Setzt den internen Zustand zurück und löst *connection_status_changed* aus.

`get_activeDeviceName()`

Gibt einen formatierten Namen des aktuell verbundenen Geräts zurück.

Parst die IDN-Nachricht, um Modell und Seriennummer zu extrahieren.

Returns

Der formatierte Gerätename
(z.B. “MODEL 2602 (SN: 12345) @ COM1”) oder “DUMMY” oder “” (leer, wenn nicht verbunden).

Return type
str

`is_connected()`

Prüft, ob eine aktive und offene Verbindung zur SMU besteht.

Returns
True, wenn verbunden und der Treiber als ‘offen’ gemeldet ist, sonst False.

Return type
bool

`reset_channel`(*channel*)

Setzt einen SMU-Kanal auf Werkseinstellungen zurück.

Parameters
channel (str) – Der Kanal, der zurückgesetzt wird (z.B. ‘a’ or ‘b’).

`set_source_voltage`(*channel*)

Konfiguriert den Kanal als SPANNUNGSQUELLE (V-Source).

Aktualisiert auch den internen Zustand, damit *set_source_limit* und *set_source_level* korrekt funktionieren.

Parameters
channel (str) – Der zu konfigurierende Kanal (z.B. ‘a’).

`set_source_current`(*channel*)

Konfiguriert den Kanal als STROMQUELLE (I-Source).

Aktualisiert auch den internen Zustand, damit *set_source_limit* und *set_source_level* korrekt funktionieren.

Parameters
channel (str) – Der zu konfigurierende Kanal (z.B. ‘a’).

`set_sense_local`(*channel*)

Stellt den Sense-Modus auf LOKAL (2-Draht-Messung).

Parameters
channel (str) – Der zu konfigurierende Kanal (z.B. ‘a’).

`set_sense_remote`(*channel*)

Stellt den Sense-Modus auf REMOTE (4-Draht-Messung).

Parameters
channel (str) – Der zu konfigurierende Kanal (z.B. ‘a’).

`set_source_level`(*channel*, *level*)

Setzt das Source-Level (V oder A).

Die Einheit (V oder A) hängt von der zuvor mit *set_source_voltage* oder *set_source_current* konfigurierten Source-Funktion ab.

Parameters

- **channel** (str) – Der zu konfigurierende Kanal (z.B. ‘a’).
- **level** (float) – Das zu setzende Level (in Volt oder Ampere).

`set_source_limit`(*channel*, *limit*)

Setzt das Source-Limit (A oder V).

Die Einheit ist *entgegengesetzt* zur Source-Funktion: - Wenn Source = Spannung (V), ist dies das Strom-Limit (A). - Wenn Source = Strom (I), ist dies das Spannungs-Limit (V).

Parameters

- **channel** (str) – Der zu konfigurierende Kanal (z.B. ‘a’).
- **limit** (float) – Das zu setzende Limit (in Ampere oder Volt).

Examples

Ein Strom-Limit (100mA) für eine Spannungsquelle setzen:

```
# 1. Zuerst Kanal 'a' als SPANNUNGSQUELLE definieren
    manager.set_source_voltage('a')

# 2. Jetzt das Limit setzen (0.1 = 100mA STROM-Limit)
manager.set_source_limit('a', 0.1)
```

Ein Spannungs-Limit (20V) für eine Stromquelle setzen:

```
# 1. Zuerst Kanal 'b' als STROMQUELLE definieren
manager.set_source_current('b')

# 2. Jetzt das Limit setzen (20.0 = 20V SPANNUNGS-Limit)
manager.set_source_limit('b', 20.0)
```

set_output_state(*channel*, *enable*)

Schaltet den Ausgang eines Kanals EIN oder AUS.

Parameters

- **channel** (str) – Der zu schaltende Kanal (z.B. 'a').
- **enable** (bool) – True, um den Ausgang einzuschalten, False, um ihn auszuschalten.

measure_iv(*channel*)

Führt eine einzelne I/V-Messung auf dem Kanal durch. … (andere Sektionen) …

Parameters

channel (str) – Der zu messende Kanal (z.B. 'a').

Returns

Ein Tupel aus (Strom, Spannung) bei Erfolg.
None bei einem Messfehler.

Return type

tuple[float, float] | None

Examples

Das Auslesen der Messung und Speichern in Variablen:

```
# Annahme: 'manager' ist eine Instanz von SmuManager
result = manager.measure_iv('a')

if result:
    current, voltage = result
    print(f"Messung OK: {current} A, {voltage} V")
else:
    print("Messung fehlgeschlagen oder keine Verbindung.")
```

CHAPTER

FIVE

SPECTROMETER MANAGER

class modules.spectrometer.SpectrometerManager.**SpectrometerManager**(*log_manager*, *profile_manager*)

Bases: QObject

Manager zur Steuerung und Verwaltung von Ocean Optics Spektrometern.

Diese Klasse kapselt die *python-seabreeze*-Bibliothek, um eine stabile Schnittstelle für die Geräteverbindung, Konfiguration (Integrationszeit, Korrekturen) und Datenaufnahme (Spektren) bereitzustellen. Sie nutzt PySide6-Signale, um die GUI über Statusänderungen zu informieren.

Parameters

- **log_manager** (LogManager) – Eine Instanz eines Log-Managers (erwartet .info, .error, etc.).
- **profile_manager** (ProfileManager) – Eine Instanz zur Verwaltung von App-Einstellungen (Lesen/Schreiben).

Signale:

connection_status_changed (bool, str):
Wird ausgelöst, wenn sich der Verbindungsstatus ändert. Args: (bool: verbunden, str: Gerätename/IDN).

device_list_updated (list):
Wird ausgelöst, nachdem die Geräteliste aktualisiert wurde. Args: (list: Liste von Gerätenamen [str], z.B. [“FLAME (Q…)”, …]).

new_spectrum_acquired (numpy.ndarray, numpy.ndarray):
Wird ausgelöst, wenn ein neues Spektrum verfügbar ist. Args: (numpy.ndarray: Wellenlängen, numpy.ndarray: Intensitäten).

__init__(*log_manager*, *profile_manager*)

Initialisiert den SpectrometerManager.

Lädt die zuletzt verwendete Konfiguration (Integrationszeit, Korrekturen) aus dem ProfileManager und versucht automatisch, eine Verbindung zum zuletzt verwendeten Gerät herzustellen.

get_deviceList()

Scannt nach verfügbaren Spektrometern und aktualisiert die interne Liste.

Verwendet *seabreeze.list_devices()* und erstellt eine Zuordnung (Map) von formatierten Gerätenamen zu Geräteinstanzen. Löst das *device_list_updated*-Signal aus.

Returns
Eine Liste formatierter Gerätenamen (z.B. [“Oceanoptics (O…)”, “USB2000 (…)”]).

Return type
list

Examples

Eine Geräteliste abrufen und in einer Combobox anzeigen:

```
# Annahme: 'spectrometer_mgr' ist eine Instanz von SpectrometerManager
# und 'ui.combo_devices' ist eine QComboBox.

# Zuerst das Signal verbinden (z.B. in __init__ der GUI)
spectrometer_mgr.device_list_updated.connect(
    lambda devices: ui.combo_devices.addItems(devices)
)

# Manuell eine Aktualisierung auslösen
ui.combo_devices.clear()
spectrometer_mgr.get_deviceList()
```

`connect`(*device_name_or_serial*)

Verbindet ein Spektrometer über seinen Namen oder seine Seriennummer.

Trennt zuerst eine eventuell bestehende Verbindung. Speichert die Seriennummer bei Erfolg für die Wiederverbindung.

Parameters
device_name_or_serial (str) – Kann entweder der formatierte Name aus *get_deviceList()* (z.B. "FLAME (Q…)") oder die reine Seriennummer (z.B. "Q…") sein.

Returns
True bei erfolgreicher Verbindung, sonst False.

Return type
bool

Examples

Verbindung über den formatierten Namen (z.B. aus einer Combobox):

```
name = "FLAME (QEP20488)"
success = spectrometer_mgr.connect(name)
if success:
    print("Verbunden mit", name)
```

Verbindung direkt über die Seriennummer:

```
serial = "QEP20488"
spectrometer_mgr.connect(serial)
```

`connect_LastDevice`()

Versucht, die Verbindung mit dem zuletzt genutzten Gerät wiederherzustellen.

Verwendet die im Profil gespeicherte Seriennummer (*self.LastDevice*).

Returns

True bei Erfolg, False, wenn kein Gerät gespeichert war
oder die Verbindung fehlschlägt.

Return type
bool

`disconnect()`

Trennt die aktive Verbindung zum Spektrometer.

Schließt das Gerät über *spectrometer.close()* und löst *connection_status_changed* aus.

`get_activeDeviceName()`

Gibt den formatierten Namen des aktuell verbundenen Geräts zurück.

Returns

Der formatierte Gerätename (z.B. "FLAME (Q…)")
oder "" (leer, wenn nicht verbunden).

Return type
str

`is_connected()`

Prüft, ob eine aktive Verbindung zum Spektrometer besteht.

Returns
True, wenn verbunden, sonst False.

Return type
bool

`set_correction_dark_count(enable)`

Aktiviert/Deaktiviert die Korrektur des Dunkelstroms (Dark Counts).

Wenn aktiviert, wird der Durchschnittswert der elektrisch verdunkelten Pixel vom Spektrum abgezogen.

Parameters
enable (bool) – True, um die Korrektur zu aktivieren, False zum Deaktivieren.

Examples

Dunkelstrom-Korrektur aktivieren:

```
spectrometer_mgr.set_correction_dark_count(True)
```

`get_correction_dark_count()`

Gibt den aktuellen Status der Dunkelstrom-Korrektur zurück.

Returns
True, wenn die Korrektur aktiv ist.

Return type
bool

`set_correction_non_linearity(enable)`

Aktiviert/Deaktiviert die Nichtlinearitäts-Korrektur.

Wenn aktiviert und vom Gerät unterstützt, werden die Messwerte anhand der im EEPROM gespeicherten Koeffizienten linearisiert.

Parameters
enable (bool) – True, um die Korrektur zu aktivieren, False zum Deaktivieren.

Examples

Nichtlinearitäts-Korrektur deaktivieren:

```
spectrometer_mgr.set_correction_non_linearity(False)
```

`get_correction_non_linearity()`

Gibt den aktuellen Status der Nichtlinearitäts-Korrektur zurück.

Returns
True, wenn die Korrektur aktiv ist.

Return type
bool

`set_integrationtime(time_us)`

Stellt die Integrationszeit des Spektrometers in Mikrosekunden (us) ein.

Die Zeit wird automatisch auf die vom Gerät unterstützten Hardware-Limits begrenzt (clamping). Die Einstellung wird auch im Profil gespeichert.

Parameters
time_us (int) – Die gewünschte Integrationszeit in Mikrosekunden.

Returns

True, wenn die Zeit erfolgreich gesetzt (oder zwischengespeichert)
wurde, False bei einem Hardware-Fehler.

Return type
bool

Examples

Integrationszeit auf 100 Millisekunden (100.000 µs) setzen:

```
spectrometer_mgr.set_integrationtime(100 * 1000)
```

Integrationszeit auf 2 Sekunden (2.000.000 µs) setzen:

```
spectrometer_mgr.set_integrationtime(2_000_000)
```

`get_integrationtime()`

Gibt die zuletzt erfolgreich gesetzte Integrationszeit in Mikrosekunden (us) zurück.

Returns
Die Integrationszeit in Mikrosekunden.

Return type
int

`get_integrationtime_limits_us()`

Gibt die Hardware-Limits (min, max) der Integrationszeit in Mikrosekunden zurück.

Returns

(min_integrationszeit_us, max_integrationszeit_us).
Gibt (0, 0) zurück, wenn nicht verbunden.

Return type
tuple[int, int]

Examples

Minimale und maximale Zeit abfragen:

```
min_t, max_t = spectrometer_mgr.get_integrationtime_limits_us()
if max_t > 0:
    print(f"Unterstützter Bereich: {min_t} µs bis {max_t} µs")
```

`get_max_intensity()`

Gibt die maximal mögliche Intensität (ADC-Sättigungswert) des Spektrometers zurück.

Dies ist typischerweise 65535.0 (für 16-bit ADC) oder 4095.0 (für 12-bit ADC).

Returns

Der maximale Sättigungswert.
Gibt 65535.0 als Fallback zurück, wenn nicht verbunden.

Return type
float

`acquire_spectrum()`

Nimmt ein einzelnes Spektrum mit den aktuell gesetzten Korrekturen auf.

Löst bei Erfolg das *new_spectrum_acquired*-Signal aus.

Returns
Ein Tupel aus (wavelengths, intensities) bei Erfolg. (None, None) bei einem Messfehler oder wenn nicht verbunden.

Return type
tuple[np.ndarray | None, np.ndarray | None]

Examples

Ein Spektrum aufnehmen und verarbeiten:

```
# Annahme: 'spectrometer_mgr' ist eine Instanz von SpectrometerManager

wavelengths, intensities = spectrometer_mgr.acquire_spectrum()

if wavelengths is not None:
    # Finde die Wellenlänge mit der maximalen Intensität
    peak_index = np.argmax(intensities)
    peak_wl = wavelengths[peak_index]
    peak_int = intensities[peak_index]

    print(f"Stärkstes Signal bei {peak_wl:.2f} nm mit {peak_int:.0f} Counts")
else:
    print("Spektrum-Aufnahme fehlgeschlagen.")
```

CHAPTER

SIX

EXPORT MANAGER

class modules.export.ExportManager.**ExportManager**(*log_manager*, *profile_manager*)

Bases: QObject

Verwaltet den Daten-Export in HDF5-Dateien und fungiert als Datenquelle für Live-Plots.

Dieser Manager abstrahiert die Komplexität von *h5py*. Er implementiert ein Zeilen-basiertes Schreibmodell: Daten werden mit *add()* gesammelt (gestaged) und mit *commit()* synchron in die HDF5-Datei geschrieben und gleichzeitig an die GUI (z.B. PlotManager) gesendet.

Funktionsweise:

1. **Setup:** Zielordner wählen (*select_directory_dialog*).
2. **Start:** Neue Datei/Gruppe erstellen (*new*).
3. **Metadaten:** Statische Infos speichern (*add_static*).
4. **Loop:**
   - Werte hinzufügen (*add('Voltage', 5.0)*).
   - Werte hinzufügen (*add('Current', 0.001)*).
   - Schreiben & Senden (*commit()*).
5. **Ende:** Datei schließen (*stop*).

Parameters

- **log_manager** (LogManager) – Instanz für das Logging.
- **profile_manager** (ProfileManager) – Instanz zum Speichern des letzten Pfades.

Signale:

data_committed (dict):
Wird bei jedem *commit()* ausgelöst. Enthält die neuen Datenpunkte für Live-Plots.

Payload-Struktur:

```
{
    'Voltage': {'value': 5.0, 'unit': 'V'},
    'Current': {'value': 1.2e-3, 'unit': 'A'},
    'Spectrum': {'value': numpy.array([...]), 'unit': 'cnt'}
}
```

export_started (str):
Wird ausgelöst, wenn eine neue Datei erstellt wurde. Args: (str: Voller Pfad zur Datei).

export_finished (str):
Wird ausgelöst, wenn der Export beendet wurde. Args: (str: Dateiname).

export_error (str):
Wird bei Schreib-/IO-Fehlern ausgelöst. Args: (str: Fehlermeldung).

`__init__`(*log_manager*, *profile_manager*)

Initialisiert den ExportManager.

`select_directory_dialog`()

Öffnet einen System-Dialog zur Auswahl des Speicherordners.

Der gewählte Pfad wird automatisch im Profil gespeichert (*Export_LastDir*).

Returns
Der ausgewählte (oder vorherige) Pfad.

Return type
str

Examples

Button-Click Handler in der GUI:

```
def on_btn_browse_clicked():
    new_path = export_mgr.select_directory_dialog()
    ui.line_edit_path.setText(new_path)
```

`set_export_directory`(*path*)

Setzt den Export-Pfad manuell und speichert ihn im Profil.

Parameters
path (str) – Der absolute Pfad zum Zielordner.

`get_export_directory`()

Liest den aktuell konfigurierten Export-Pfad aus dem Profil.

Returns
Der Pfad oder das Benutzer-Home-Verzeichnis als Fallback.

Return type
str

`new`(*filename_base*, *dataset_name='Measurement'*)

Erstellt eine neue HDF5-Datei und bereitet die Messung vor.

Der Dateiname wird automatisch mit einem Zeitstempel versehen (*Name_YYYYMMDD_HHMMSS.h5*). Schließt eine evtl. offene Datei zuvor.

Parameters

- **filename_base** (str) – Der Basisname der Datei (z.B. “Experiment_A”).
- **dataset_name** (str) – Der Name der HDF5-Gruppe für die Daten (Standard: “Measurement”).

Returns
True bei Erfolg, False bei IO-Fehlern.

Return type
bool

Examples

Eine neue Messdatei starten:

```
if export_mgr.new("OLED_IV_Curve"):
    print("Datei erstellt, bereit für Daten.")
```

add(*name*, *data*, *unit=''*)

Fügt dynamische Messdaten zum internen Puffer hinzu.

Diese Daten werden **noch nicht** auf die Festplatte geschrieben. Erst der Aufruf von *commit()* schreibt alle mit *add()* gesammelten Werte als eine Zeile in die HDF5-Datasets.

Parameters

- **name** (str) – Der Name des Datasets (Spaltenname), z.B. "Voltage".
- **data** (float | numpy.ndarray) – Der Messwert (Skalar) oder ein Array (z.B. ganzes Spektrum).
- **unit** (str, *optional*) – Die physikalische Einheit (z.B. "V", "nm"), wird als HDF5-Attribut gespeichert.

Examples

Werte für den nächsten Zeitschritt sammeln:

```
# Skalar hinzufügen
export_mgr.add("Time", 1.5, "s")
export_mgr.add("Current", 1e-6, "A")

# Array hinzufügen (z.B. Spektrum)
spectrum_data = np.array([1, 2, 3, ...])
export_mgr.add("Spectrum", spectrum_data, "counts")

# WICHTIG: Jetzt commit aufrufen!
export_mgr.commit()
```

add_static(*name*, *data*, *unit=''*)

Speichert einmalige, statische Daten (Metadaten/Konstanten).

Im Gegensatz zu *add()* wird hier **sofort** geschrieben. Das Dataset hat die Länge 1 und wird nicht erweitert.

Parameters

- **name** (str) – Name des Datasets.
- **data** – Der Wert (String, Zahl, Array).
- **unit** (str, *optional*) – Einheit.

Examples

Geräte-Infos speichern:

```
export_mgr.add_static("User", "Max Mustermann")
export_mgr.add_static("IntegrationTime", 100, "ms")
```

add_group_attribute(*key*, *value*)

Fügt Metadaten als Attribute zur HDF5-Hauptgruppe hinzu.

Parameters

- **key** (str) – Attribut-Name.
- **value** – Attribut-Wert.

commit()

Schließt den aktuellen Datenpunkt ab (Synchronisation).

Führt folgende Schritte aus: 1. Erstellt HDF5-Datasets für neue Spalten (falls nötig). 2. Erweitert alle Datasets um eine Zeile. 3. Schreibt die gepufferten Werte aus *add()* in die neue Zeile. 4. Füllt fehlende Werte (falls *add* für eine Spalte vergessen wurde) mit NaN. 5. Sendet das *data_committed*-Signal für Live-Plots. 6. Leert den Puffer für den nächsten Punkt.

Examples

Am Ende einer Messschleife aufrufen:

```
while measuring:
    val = instrument.read()
    export_mgr.add("Reading", val)
    export_mgr.commit() # Schreibt auf Disk & updated Plot
```

stop()

Beendet den Export und schließt die HDF5-Datei sauber.

Sendet das *export_finished*-Signal.

PYTHON MODULE INDEX

m