ionpy Pro OS — Tiefgreifende Technische Analyse & Roadmap

Inhaltsverzeichnis

---

1. Executive Summary

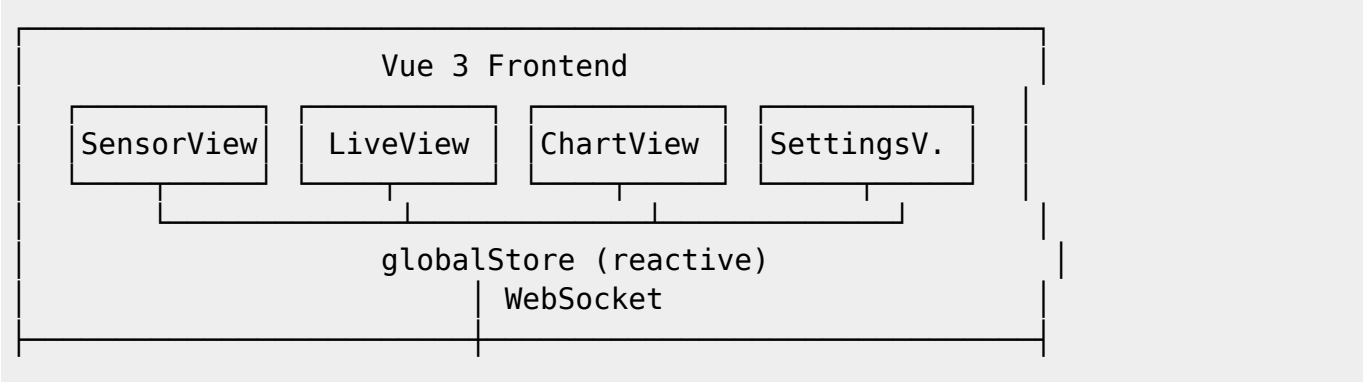
ionpy ist ein beeindruckend durchdachtes Hardware-Abstraktionssystem mit einer klaren Schichtenarchitektur: **Transport → Framing → Device → EventBus → WebSocket → Vue-Frontend**. Die deklarative Entity-Definition über Python-Klassenattribute ist ein cleverer ORM-Ansatz, der an Django-Models erinnert. Das Frontend mit GridStack-basiertem Window-Management und uPlot-Charts ist für ein Ein-Personen-Projekt außergewöhnlich professionell.

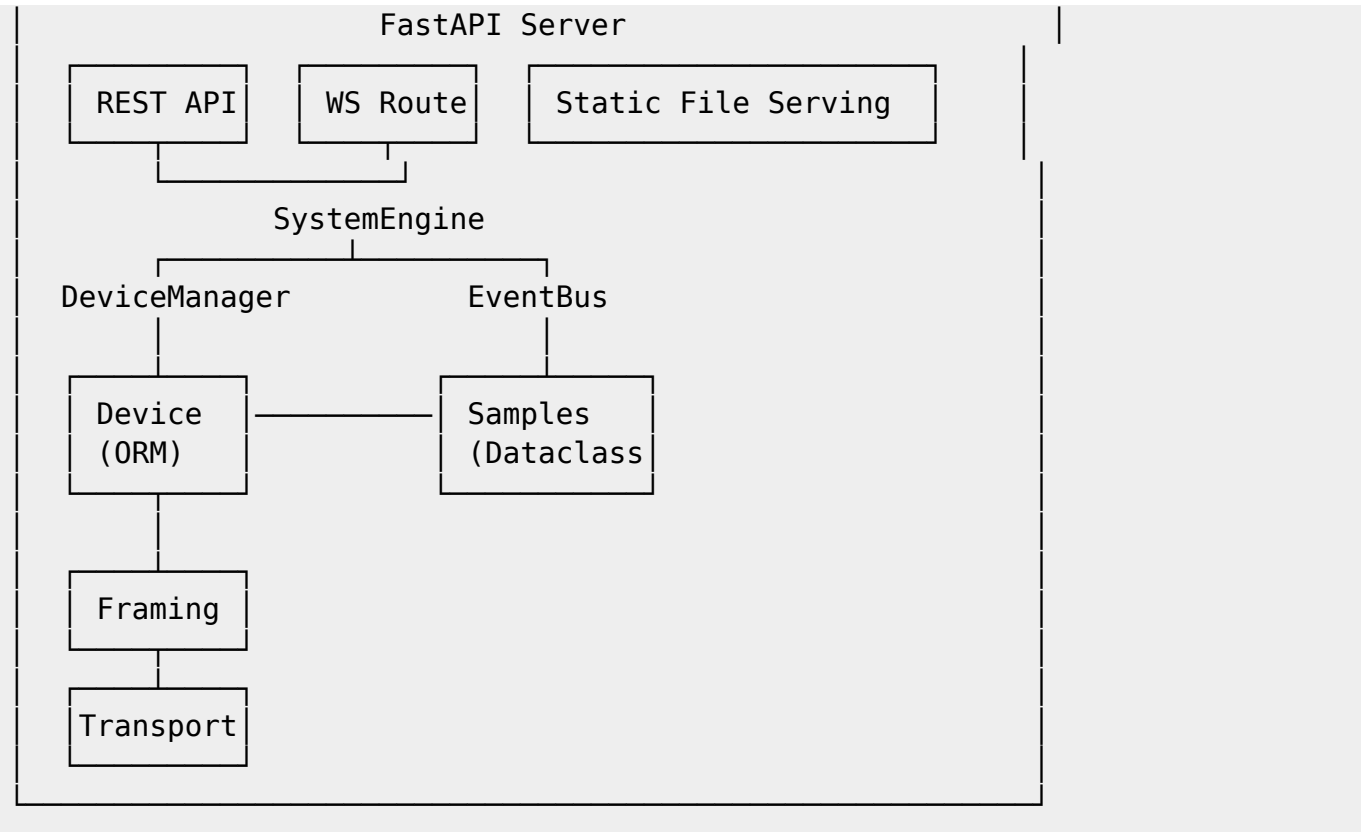
Stärken: Klare Schichtentrennung, elegantes Entity-ORM, solides Reconnect-Handling, durchdachtes UI.

Größte Lücken: Kein Persistenz-Layer (Datenbank), kein Automation/Scripting-Engine, fehlendes Alarm-Routing, keine Unit-Tests, einige Race-Conditions im State-Management.

---

2. Architektur-Übersicht





---

3. Was ist gut gelöst?

3.1 Deklaratives Entity-ORM

Die Idee, Entities als Klassenattribute zu definieren und per `_build_entities_from_class()` via `copy.deepcopy()` zu instanziieren, ist elegant und spart enorm viel Boilerplate:

```
# So sieht ein kompletter Treiber in seiner Minimalform aus:
class MyPSU(AbstractDevice):
    voltage = NumericEntity(name="Spannung", unit="V", mode=AccessMode.RO)
    current = NumericEntity(name="Strom", unit="A", mode=AccessMode.RO)
    output  = ToggleEntity(name="Ausgang", mode=AccessMode.RW)
```

Bewertung: ★★★★★ — Industriequalität. Vergleichbar mit SQLAlchemy/Django ORM Patterns.

3.2 Transport/Framing Stack mit Factory & Registry

Die `TransportFactory` mit ihrer Registry ist sauber und erweiterbar. Der Ansatz, Transport und Framing getrennt zu halten, ermöglicht beliebige Kombinationen (Serial+Modbus, TCP+Delimiter, TCP+StartStop etc.):

```
TRANSPORT_REGISTRY = {
    "serial": "devices.transport.serial.Serial",
    "tcp":    "devices.transport.tcp_client.TCPClient",
```

```
}
FRAMING_REGISTRY = {
    "delimiter":  "devices.framing.delimiter.Delimiter",
    "modbus_rtu": "devices.framing.modbus_rtu.ModbusRTUFramer",
}
```

Bewertung: ★★★★☆ — Sehr gut, aber es fehlen noch einige Transporte (siehe Roadmap).

3.3 EventBus mit Queue-basiertem Publish/Subscribe

Das Queue-basierte Pub/Sub-System ist elegant. Jeder Subscriber bekommt seine eigene `asyncio.Queue`, was Race-Conditions vermeidet. Der neue QueueFull-Fix mit FIFO-Drop ist pragmatisch und verhindert Memory-Leaks:

```
except asyncio.QueueFull:
    try:
        q.get_nowait()  # Ältestes verwerfen
    except asyncio.QueueEmpty:
        pass
    q.put_nowait(sample)
```

Bewertung: ★★★★☆ — Gut für den aktuellen Umfang, braucht aber Topic-Filtering (siehe 4.2).

3.4 Frontend: Window-Manager mit GridStack

Das GridStack-basierte Window-Management mit Minimize/Maximize/Popout/Restore ist für eine Web-Applikation beeindruckend. Die Workspace-Serialisierung (inkl. `exportState()` pro Widget) ist durchdacht:

```
const exportState = () => ({
    viewMode: viewMode.value,
    activeTab: activeTab.value,
    showReadOnly: showReadOnly.value
});
```

Bewertung: ★★★★★ — Auf dem Niveau kommerzieller SCADA-Systeme.

3.5 Reconnect-Handling

Die zweistufige Reconnect-Logik (passiver `is_alive()` Check in der Loop + aktiver `check_hardware_exists()` bei Recovery) ist industrietauglich und vermeidet unnötige Port-Scans.

Bewertung: ★★★★☆ — Solide, aber Exponential-Backoff fehlt (siehe 4.5).

3.6 Chart-System (uPlot + Plugins)

Die Plugin-Architektur für uPlot (Navigation, Measure, Limits, Export) ist sauber modularisiert. Besonders das Measure-Plugin mit Snap-to-Data und Delta-Berechnung ist Labor-tauglich.

Bewertung: ★★★★☆ — Sehr gut, braucht aber historische Daten (DB-Backend).

---

4. Schwachstellen & Handlungsbedarf

4.1 KRITISCH: Kein Persistenz-Layer

Problem: Alle Messdaten existieren nur im RAM. Ein Server-Neustart = totaler Datenverlust. Der `state_cache` in der Engine ist nur ein flüchtiges Dictionary.

Lösung: Ein `StorageBackend` mit austauschbarer Strategie:

```
# core/storage/base.py
from abc import ABC, abstractmethod
from typing import List, Optional
from datetime import datetime

class StorageBackend(ABC):
    """Abstrakte Basis für alle Datenspeicher."""

    @abstractmethod
    async def store_sample(self, sample: dict):
        """Speichert einen einzelnen Messwert."""
        pass

    @abstractmethod
    async def query(self, device_id: str, entity_id: str,
                    start: datetime, end: datetime,
                    downsample: Optional[int] = None) -> List[dict]:
        """Gibt historische Daten zurück, optional heruntergesampelt."""
        pass

    @abstractmethod
    async def get_latest(self, device_id: str, entity_id: str) ->
Optional[dict]:
        """Gibt den zuletzt gespeicherten Wert zurück."""
        pass
```

```
# core/storage/sqlite_backend.py
import aiosqlite
from core.storage.base import StorageBackend

class SQLiteBackend(StorageBackend):
```

```
    def __init__(self, db_path: str = "data/ionpy.db"):
        self.db_path = db_path

    async def initialize(self):
        async with aiosqlite.connect(self.db_path) as db:
            await db.execute("""
                CREATE TABLE IF NOT EXISTS samples (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    timestamp REAL NOT NULL,
                    device_id TEXT NOT NULL,
                    entity_id TEXT NOT NULL,
                    value REAL,
                    value_text TEXT,
                    alarm_state TEXT DEFAULT 'NORMAL',
                    session_id TEXT
                )
            """)
            await db.execute("""
                CREATE INDEX IF NOT EXISTS idx_samples_lookup
                ON samples(device_id, entity_id, timestamp)
            """)
            await db.commit()

    async def store_sample(self, sample: dict):
        val = sample.get("value")
        async with aiosqlite.connect(self.db_path) as db:
            await db.execute(
                "INSERT INTO samples (timestamp, device_id, entity_id,
value, value_text, alarm_state, session_id) VALUES (?, ?, ?, ?, ?, ?, ?)",
                (
                    sample["timestamp"],
                    sample["device_id"],
                    sample["entity_id"],
                    val if isinstance(val, (int, float)) else None,
                    str(val) if not isinstance(val, (int, float)) else None,
                    sample.get("alarm_state", "NORMAL"),
                    sample.get("session_id")
                )
            )
            await db.commit()

    async def query(self, device_id, entity_id, start, end,
downsample=None):
        async with aiosqlite.connect(self.db_path) as db:
            db.row_factory = aiosqlite.Row
            rows = await db.execute_fetchall(
                "SELECT timestamp, value, value_text, alarm_state FROM
samples WHERE device_id=? AND entity_id=? AND timestamp BETWEEN ? AND ?
ORDER BY timestamp",
                (device_id, entity_id, start.timestamp(), end.timestamp())
            )
```

```
            results = [dict(r) for r in rows]

            if downsample and len(results) > downsample:
                step = len(results) // downsample
                results = results[::step]

            return results
```

Hinweis für KI-Umsetzung: Der `_cache_worker` in der Engine muss um einen parallelen `_storage_worker` ergänzt werden. Dabei sollte ein Batch-Write (z.B. alle 500ms sammeln, dann bulk-insert) verwendet werden, um die Datenbank nicht mit Einzel-Inserts zu erschlagen.

4.2 HOCH: EventBus hat kein Topic-Filtering

Problem: Jeder Subscriber bekommt ALLE Samples — auch die, die er nicht braucht. Bei 10 Geräten mit je 20 Sensoren bei 10Hz sind das 2000 Msg/s in JEDER Queue.

Lösung: Filter-basiertes Subscribe:

```
# core/event_bus.py (Erweitert)

class EventBus:
    def __init__(self, session_manager):
        self.session_manager = session_manager
        self._subscriptions: List[dict] = []

    def subscribe(self, maxsize: int = 500,
                  device_filter: str = None,
                  entity_filter: str = None,
                  type_filter: str = None) -> asyncio.Queue:
        """
        Subscribes mit optionalem Filter.
        device_filter: "DPS_01" oder "*" für alle
        entity_filter: "u_out" oder "*"
        type_filter:   "numeric" oder "*"
        """
        q = asyncio.Queue(maxsize=maxsize)
        self._subscriptions.append({
            "queue": q,
            "device": device_filter,
            "entity": entity_filter,
            "type": type_filter
        })
        return q

    async def publish(self, sample: BaseSample):
        if self.session_manager.is_recording:
            sample.session_id = self.session_manager.current_session_id

        for sub in list(self._subscriptions):
```

```
            if sub["device"] and sub["device"] != "*" and sub["device"] !=
sample.device_id:
                continue
            if sub["entity"] and sub["entity"] != "*" and sub["entity"] !=
sample.entity_id:
                continue
            if sub["type"] and sub["type"] != "*" and sub["type"] !=
sample.type:
                continue

            q = sub["queue"]
            try:
                q.put_nowait(sample)
            except asyncio.QueueFull:
                try: q.get_nowait()
                except asyncio.QueueEmpty: pass
                try: q.put_nowait(sample)
                except asyncio.QueueFull: pass
```

4.3 HOCH: Race-Condition bei device.state im Frontend

Problem: Im `store.js` wird `device.state` als Array modifiziert. Vue 3's Reactive-System trackt Array-Mutationen über `.findIndex()` + direkte Zuweisung — das funktioniert, aber es gibt eine subtile Race-Condition: Wenn zwei WebSocket-Messages für das gleiche Entity fast gleichzeitig ankommen, kann der `findIndex()` auf veralteten Daten operieren.

Lösung: State als Map statt Array:

```
// store.js (Verbesserter Ansatz)
ws.onmessage = (e) => {
    window.dispatchEvent(new CustomEvent('ws-message', { detail: e.data }));
    const sample = JSON.parse(e.data);
    const device = globalStore.devices[sample.device_id];

    if (device) {
        // NEU: State als reaktives Objekt statt Array
        if (!device.stateMap) device.stateMap = {};
        device.stateMap[sample.entity_id] = {
            sensor: sample.entity_id,
            value: sample.value,
            timestamp: sample.timestamp,
            alarm_state: sample.alarm_state
        };

        // Legacy-Kompatibilität: Array wird aus Map generiert
        device.state = Object.values(device.stateMap);
    }
};
```

4.4 MITTEL: Manifest-Schema hat Lücken

Problem: Das `get_manifest()` gibt `meta` als flaches Dict zurück, aber wichtige Felder fehlen:

- Kein `categories` (PSU, Load, BMS...)
- Kein `transports` (Serial, TCP...)
- Kein `capabilities` (Was kann dieses Gerät?)
- Kein `firmware_version` oder `serial_number`

Lösung:

```
def get_manifest(self) -> Manifest:
    meta_dict = {
        "alias": self.alias,
        "vendor": self.vendor,
        "model": self.model,
        "icon": self.icon,
        "description": self.description,
        "location": self.location,
        # NEU:
        "categories": [c.value for c in self.categories],
        "transport_type": self.framer.transport.__class__.__name__,
        "capabilities": self._get_capabilities(),
        "auto_start": self.auto_start,
        "active_polling": self.ACTIVE_POLLING,
    }
    return Manifest(...)

def _get_capabilities(self) -> list:
    """Automatische Feature-Erkennung aus registrierten Entities."""
    caps = []
    if any(isinstance(e, ToggleEntity) and e.id == "output_toggle"
           for e in self.entities.values()):
        caps.append("switchable_output")
    if "emergency_stop" in dir(self.__class__) and \
       self.__class__.emergency_stop is not AbstractDevice.emergency_stop:
        caps.append("emergency_stop")
    if any(isinstance(e, TableEntity) for e in self.entities.values()):
        caps.append("programmable_profiles")
    if any(isinstance(e, WaveformEntity) for e in self.entities.values()):
        caps.append("waveform_capture")
    return caps
```

4.5 MITTEL: Reconnect ohne Exponential Backoff

Problem: `reconnect()` wartet immer exakt `delay` Sekunden. Bei echten Hardware-Ausfällen ist das suboptimal.

Lösung:

```
async def reconnect(self, max_retries: int = 10,
                    initial_delay: float = 2.0,
                    max_delay: float = 60.0,
                    backoff_factor: float = 1.5) -> bool:
    delay = initial_delay
    for attempt in range(1, max_retries + 1):
        try:
            await self.disconnect()
            if self.check_hardware_exists():
                if await self.connect():
                    self.logger.info(f"Wiederverbunden (Versuch {attempt})")
                    return True
        except Exception as e:
            self.logger.debug(f"Reconnect-Fehler #{attempt}: {e}")

        self.logger.info(f"Versuch {attempt} fehlgeschlagen. Nächster in
{delay:.1f}s")
        await asyncio.sleep(delay)
        delay = min(delay * backoff_factor, max_delay)

    return False
```

4.6 MITTEL: WaveformEntity-Integration unvollständig

Problem: Die WaveformEntity existiert im Backend, aber es gibt keinen Frontend-View dafür. Im `chart_view.js` werden Waveforms komplett ignoriert (sie werden ja auch im `_cache_worker` gefiltert).

Lösung: Ein dedizierter `WaveformView` wird benötigt (siehe Roadmap Phase 5).

4.7 NIEDRIG: DataLogger Widget speichert keine Daten im Workspace

Das ist ein bewusster Kompromiss (*Wir speichern die 'rows' absichtlich NICHT), aber es wäre sinnvoll, zumindest die letzten N Zeilen optional zu persistieren. —- ===== 5. Fehlende Module & Systeme ===== ==== 5.1 Automation Engine (Höchste Priorität) ==== Das automation/ Verzeichnis ist leer. Hier fehlt das Herzstück für jeden ernsthaften Laboraufbau: <code python> # automation/script_engine.py """ Hinweis für KI-Umsetzung: - Scripte werden als Python-Dateien im Ordner data/scripts/ abgelegt - Sie bekommen ein API-Objekt injiziert, das sicheren Zugriff auf Geräte bietet - Scripte laufen in eigenen asyncio Tasks - Ein Script darf NIEMALS direkt auf Entities zugreifen, sondern immer über die bereitgestellte API """ import asyncio import logging from typing import Optional, Any class ScriptAPI: """Die sichere Schnittstelle, die jedes Automations-Script bekommt."""* *def init(self, engine, script_name: str): self.engine = engine self.script_name = script_name self.logger = logging.getLogger(f"Script.{script_name}") self._stop_event = asyncio.Event() async def read(self, device_id: str, entity_id: str) → Any: """Liest den aktuellen Wert einer Entity."""* *dev =*

self.engine.device_manager.get_device(device_id) if not dev: raise ValueError(f“Gerät '{device_id}' nicht gefunden”) ent = dev.entities.get(entity_id) if not ent: raise ValueError(f“Entity '{entity_id}' nicht gefunden”) return ent.value async def write(self, device_id: str, entity_id: str, value: Any): “”“Schreibt einen Wert an eine Entity.”“” await self.engine.execute_command(device_id, entity_id, value) async def wait(self, seconds: float): “”“Wartet, kann aber durch Stop unterbrochen werden.”“” try: await asyncio.wait_for(self._stop_event.wait(), timeout=seconds) except asyncio.TimeoutError: pass async def wait_until(self, device_id: str, entity_id: str, condition: str, value: float, timeout: float = 60.0) → bool: “”“ Wartet bis eine Bedingung erfüllt ist. condition: '>', '<', '>=', '⇐', '==', '!=' ”“” import operator ops = {'>': operator.gt, '<': operator.lt, '>=': operator.ge, '⇐': operator.le, '==': operator.eq, '!=': operator.ne} op = ops.get(condition) if not op: raise ValueError(f“Unbekannte Bedingung: {condition}”) start = asyncio.get_event_loop().time() while not self._stop_event.is_set(): current = await self.read(device_id, entity_id) if current is not None and op(float(current), value): return True if asyncio.get_event_loop().time() - start > timeout: self.logger.warning(f“Timeout: {device_id}.{entity_id} {condition} {value}”) return False await asyncio.sleep(0.1) return False def stop(self): self._stop_event.set() class ScriptRunner: “”“Verwaltet die Ausführung von Automations-Scripten.”“” def init(self, engine): self.engine = engine self.logger = logging.getLogger(“ScriptRunner”) self.active_scripts: dict = {} async def run_script(self, script_path: str, name: str = None): if name is None: name = script_path.split(“/”)[-1].replace(“.py”, “”) if name in self.active_scripts: raise RuntimeError(f“Script '{name}' läuft bereits!”) api = ScriptAPI(self.engine, name) with open(script_path, 'r') as f: script_code = f.read() async def _run(): try: exec_globals = { “api”: api, “log”: api.logger, “asyncio”: asyncio, } exec(compile(script_code, script_path, “exec”), exec_globals) main_func = exec_globals.get(“main”) if main_func and asyncio.iscoroutinefunction(main_func): await main_func(api) else: api.logger.error(“Script hat keine 'async def main(api)' Funktion!”) except asyncio.CancelledError: api.logger.info(“Script wurde abgebrochen.”) except Exception as e: api.logger.error(f“Script-Fehler: {e}”, exc_info=True) finally: self.active_scripts.pop(name, None) task = asyncio.create_task(_run()) self.active_scripts[name] = {“task”: task, “api”: api} async def stop_script(self, name: str): entry = self.active_scripts.get(name) if entry: entry[“api”].stop() entry[“task”].cancel() self.active_scripts.pop(name, None) </code> ***Beispiel-Script*** *(data/scripts/charge_test.py): <code python> “”“ Einfacher Lade-Test: Setzt Spannung und Strom, wartet bis Zielspannung erreicht, schaltet ab. ”“” async def main(api): log.info(“=== Lade-Test gestartet ===”) await api.write(“PSU_01”, “output_toggle”, False) await api.wait(0.5) await api.write(“PSU_01”, “u_set”, 12.6) await api.write(“PSU_01”, “i_set”, 2.0) await api.wait(0.5) await api.write(“PSU_01”, “output_toggle”, True) log.info(“Ausgang eingeschaltet. Warte auf 12.5V…”) reached = await api.wait_until(“PSU_01”, “u_out”, “>=”, 12.5, timeout=300) if reached: log.info(“Zielspannung erreicht!”) else: log.warning(“Timeout! Zielspannung nicht erreicht.”) await api.write(“PSU_01”, “output_toggle”, False) log.info(“=== Lade-Test beendet ===”) </code> ==== 5.2 Alarm-Routing & Notification System ====* ***Problem:*** *Alarm-States werden zwar berechnet*

(_evaluate_alarms()), aber nirgendwo hingeroutet. Es gibt keine Benachrichtigungen, kein Alarm-Log, keine Acknowledges. <code python> # core/alarm_manager.py import asyncio import time import logging from typing import List, Dict, Optional, Callable from dataclasses import dataclass, field from structures.enums import AlarmState @dataclass class AlarmEvent: timestamp: float device_id: str entity_id: str entity_name: str old_state: AlarmState new_state: AlarmState value: float unit: str = “” acknowledged: bool = False ack_time: Optional[float] = None ack_by: str = “” class AlarmManager: def init(self): self.logger = logging.getLogger(“AlarmManager”) self.active_alarms: Dict[str, AlarmEvent] = {} self.alarm_history: List[AlarmEvent] = [] self.max_history = 10000 self._callbacks: List[Callable] = [] def register_callback(self, callback: Callable): “””Registriert einen Handler (z.B. Email, Webhook, Buzzer).””” self._callbacks.append(callback) async def process_alarm_change(self, device_id, entity_id, entity_name, old_state, new_state, value, unit=“”): key = f“{device_id}.{entity_id}” event = AlarmEvent(timestamp=time.time(), device_id=device_id, entity_id=entity_id, entity_name=entity_name, old_state=old_state, new_state=new_state, value=value, unit=unit) if new_state != AlarmState.NORMAL: self.active_alarms[key] = event self.logger.warning(f“ALARM [{new_state.value}]: {device_id}/{entity_name} = {value} {unit}”) else: self.active_alarms.pop(key, None) self.logger.info(f“ALARM CLEAR: {device_id}/{entity_name} zurück auf NORMAL”) self.alarm_history.append(event) if len(self.alarm_history) > self.max_history: self.alarm_history = self.alarm_history[-self.max_history:] for cb in self._callbacks: try: if asyncio.iscoroutinefunction(cb): await cb(event) else: cb(event) except Exception as e: self.logger.error(f“Alarm-Callback Fehler: {e}”) def acknowledge(self, key: str, by: str = “user”): alarm = self.active_alarms.get(key) if alarm: alarm.acknowledged = True alarm.ack_time = time.time() alarm.ack_by = by def get_active_alarms(self) → List[dict]: return [{ “key”: k, “timestamp”: a.timestamp, “device_id”: a.device_id, “entity_id”: a.entity_id, “entity_name”: a.entity_name, “state”: a.new_state.value, “value”: a.value, “unit”: a.unit, “acknowledged”: a.acknowledged } for k, a in self.active_alarms.items()] </code> ==== 5.3 Session Recording / Test-Runner ==== Der SessionManager existiert, wird aber nie wirklich benutzt. Er sollte die Basis für Testprotokolle werden: <code python> # Erweiterung von core/session_manager.py class SessionManager: # … bestehender Code … async def start_recording(self, name: str, targets: list = None, metadata: dict = None) → str: session_id = self.start_session(name) self.targets = targets or [“”] self.metadata = metadata or {} self.metadata[“start_time”] = time.time() return session_id async def stop_recording(self) → dict: summary = { “session_id”: self.current_session_id, “name”: self.current_session_name, “duration”: time.time() - self.start_time, “metadata”: self.metadata } self.stop_session() return summary </code> ==== 5.4 Config Editor (YAML Live-Bearbeitung) ==== <code python> # web/rest.py (Neue Endpoints) @router.post(“/config/devices”) async def api_add_device(request: Request): “””Fügt ein neues Gerät zur Laufzeit hinzu.””” engine = request.app.state.engine body = await request.json() required = [“id”, “driver”] for field in required: if field not in body: raise HTTPException(400, f“Pflichtfeld '{field}' fehlt”) if body[“id”] in engine.device_manager.devices: raise HTTPException(409, f“Gerät*

'{body['id']}' existiert bereits") try: engine.device_manager._spawn_device(body) new_dev = engine.device_manager.devices.get(body["id"]) if new_dev: await new_dev.connect_wrapper() return {"status": "ok", "device_id": body["id"]} except Exception as e: raise HTTPException(500, f"Fehler: {e}") @router.delete("/config/devices/{device_id}") async def api_remove_device(device_id: str, request: Request): """Entfernt ein Gerät zur Laufzeit."""" engine = request.app.state.engine dev = engine.device_manager.devices.get(device_id) if not dev: raise HTTPException(404, "Gerät nicht gefunden") await dev._do_disconnect() del engine.device_manager.devices[device_id] return {"status": "ok", "removed": device_id} </code> —- ===== 6. Entity-System: Analyse & Verbesserungen ===== ==== 6.1 Fehlende Entity-Typen ==== ^ Entity-Typ ^ Beschreibung ^ Beispiel-Anwendung ^ | EnumEntity | Statuswerte mit definiertem Mapping (int → String) | Modbus Status-Register: 0="Idle", 1="Running", 2="Fault" | | BitmaskEntity | Einzelne Bits eines Statusregisters als benannte Flags | DPS5005 Protection Flags, BMS Fehlerregister | | ArrayEntity / VectorEntity | Fest-Array numerischer Werte (z.B. 16 Zellspannungen) | BMS mit 8-16 Zellen, Multi-Temperatur-Sensoren | | ImageEntity | Base64-encoded Bilddaten | Kamera-Modul, Thermal-Imager | | FileEntity | Datei-Upload/Download (z.B. Firmware, CSV) | Firmware-Update für Embedded-Geräte | | TimerEntity | Countdown/Countup mit Start/Stop/Reset | Laufzeitmessung, Alterungstests | | RangeEntity | Zwei zusammengehörige Werte (Min/Max Paar) | Temperatur-Bereich, Spannungsfenster | ***Beispiel-Implementierung EnumEntity:*** *<code python> # structures/entities/logic.py (Ergänzung) class EnumEntity(LogicEntity): """ Mappt Integer-Rohwerte auf lesbare Strings. Ideal für Modbus Status-Register. Beispiel: mode = EnumEntity(name="Betriebsmodus", mapping={0: "Idle", 1: "CV", 2: "CC", 3: "CP"}, mode=AccessMode.RO) """ def init(self, name: str, mapping: dict,* ***kwargs): ui = kwargs.get("ui", UIMeta()) ui.ui_type = UIType.TEXT kwargs["ui"] = ui super().init(name=name,*** *kwargs) self.mapping = mapping self.raw_value = None def update_from_raw(self, raw_int: int): self.raw_value = raw_int self.value = self.mapping.get(raw_int, f"Unknown ({raw_int})") def to_dict(self) → dict: d = super().to_dict() d["mapping"] = self.mapping d["raw_value"] = self.raw_value return d def create_sample(self, device_id, session_id=None): return TextSample(timestamp=self.timestamp, session_id=session_id, entity_id=self.id, device_id=device_id, channel=self.channel, name=self.name, value=str(self.value) if self.value else "") </code>* ***Beispiel-Implementierung BitmaskEntity:*** *<code python> class BitmaskEntity(LogicEntity): """ Interpretiert ein Integer-Register als Sammlung benannter Bits. Beispiel: protection_flags = BitmaskEntity(name="Schutz-Flags", bits={ 0: "OVP aktiv", 1: "OCP aktiv", 2: "OTP aktiv", 3: "Kurzschluss", 7: "Fan-Fehler" }) """ def init(self, name: str, bits: dict,* ***kwargs): ui = kwargs.get("ui", UIMeta()) ui.ui_type = UIType.TEXT kwargs["ui"] = ui super().init(name=name,*** *kwargs) self.bits = bits self.raw_value = 0 self.active_flags: list = [] def update_from_raw(self, raw_int: int): self.raw_value = raw_int self.active_flags = [] for bit_pos, label in self.bits.items(): if raw_int & (1 « bit_pos): self.active_flags.append(label) self.value = ", ".join(self.active_flags) if self.active_flags else "Keine" def to_dict(self) → dict: d = super().to_dict() d["bits"] = self.bits d["raw_value"] = self.raw_value*

d[“active_flags”] = self.active_flags return d </code> ==== 6.2 Bestehende Entity-Probleme ==== === Problem 1: update_entity() ist zu generisch === <code python> # Aktuell in base.py: if isinstance(entity, NumericEntity) and isinstance(raw_value, (int, float)): entity.update_real_value(raw_value) else: entity.value = raw_value # ← Keine Validierung! </code> Das else Catch-All überspringt jegliche Validierung. Ein ToggleEntity akzeptiert so auch Floats, ein SelectEntity akzeptiert ungültige Optionen. ***Lösung:*** *Jede Entity-Klasse bekommt eine accept_value() Methode: <code python> # BaseEntity def accept_value(self, raw_value: Any) → Any: “””Validiert und transformiert den eingehenden Wert.””” return raw_value # ToggleEntity def accept_value(self, raw_value: Any) → bool: if isinstance(raw_value, bool): return raw_value if isinstance(raw_value, (int, float)): return bool(raw_value) if isinstance(raw_value, str): return raw_value.upper() in (“ON”, “TRUE”, “1”) return False # SelectEntity def accept_value(self, raw_value: Any) → str: val = str(raw_value) if val not in self.options: return self.value # Alten Wert behalten return val # Dann in update_entity(): async def update_entity(self, entity_id, raw_value, timestamp=None): entity = self.entities.get(entity_id) if not entity: return if isinstance(entity, NumericEntity) and isinstance(raw_value, (int, float)): entity.update_real_value(raw_value) else: entity.value = entity.accept_value(raw_value) entity.timestamp = timestamp or time.time() sample = entity.create_sample(self.device_id) await self.bus.publish(sample) </code> === Problem 2: channel Default-Logik ist fragil === In _build_entities_from_class() wird der Channel hart auf “System” gesetzt, wenn der Attributname in einer festen Liste steht. Das skaliert nicht: <code python> # Aktuell: ent_copy.channel = “System” if attr_name in [“state”, “diag_msg”, “poll_interval”] else “Ch1” </code>* ***Besser:*** *Der Channel sollte direkt in der Entity-Definition stehen (was bei den meisten Treibern schon der Fall ist). Die Überschreibung in _build_entities_from_class sollte nur greifen, wenn kein Channel explizit gesetzt wurde. === Problem 3: WaveformEntity.create_sample() ignoriert value-Parameter === <code python> # In update_entity wird value=None übergeben: await self.update_entity(“display”, None) # Aber create_sample() greift direkt auf self.time_axis und self.channels zu. # Das ist inkonsistent mit dem Rest des Systems. </code>* ***Lösung:*** *<code python> class WaveformEntity(PhysicalEntity): def update_waveform(self, time_axis: list, channels: dict): self.value = {“time”: time_axis, “channels”: channels} self.time_axis = time_axis self.channels = channels </code> —- ===== 7. REST API: Fehlende & zu erweiternde Endpoints ===== ==== 7.1 Fehlende Endpoints ==== ^ Methode ^ Pfad ^ Beschreibung ^ | GET | /api/devices/{id} | Einzelnes Gerät mit vollem State abrufen | | GET | /api/devices/{id}/entities | Alle Entities eines Geräts | | GET | /api/devices/{id}/entities/{eid} | Einzelne Entity mit aktuellem Wert | | GET | /api/devices/{id}/entities/{eid}/history | Historische Werte (braucht DB) | | POST | /api/devices/{id}/connect | Gerät manuell verbinden | | POST | /api/devices/{id}/disconnect | Gerät manuell trennen | | POST | /api/config/devices | Neues Gerät zur Laufzeit hinzufügen | | DELETE | /api/config/devices/{id} | Gerät zur Laufzeit entfernen | | PATCH | /api/config/devices/{id} | Gerät-Config live ändern | | GET | /api/alarms | Aktive Alarme auflisten | | GET | /api/alarms/history | Alarm-Verlauf | | POST | /api/alarms/{key}/acknowledge | Alarm quittieren | | GET | /api/scripts | Verfügbare Automations-Scripte | | POST | /api/scripts/{name}/run | Script starten | | POST | /api/scripts/{name}/stop*

*| Script stoppen | | GET | /api/scripts/active | Laufende Scripte | | GET | /api/sessions | Alle Sessions auflisten | | POST | /api/sessions/start | Aufnahme starten | | POST | /api/sessions/stop | Aufnahme stoppen | | GET | /api/sessions/{id}/export | Session als CSV/JSON exportieren | | GET | /api/system/status | System-Gesundheit (RAM, CPU, Uptime) | | POST | /api/system/restart | Server-Neustart | ==== 7.2 Zu erweiternde Endpoints ==== === GET /api/devices – Filter & Paginierung === <code python> @router.get("/api/devices") def get_devices(request: Request, state: str = None, category: str = None, include_meta: bool = True, include_state: bool = True): engine = request.app.state.engine result = [] for dev_id, dev in engine.device_manager.devices.items(): if state: current = dev.entities.get("state") if current and current.value != state: continue entry = {"device_id": dev_id} if include_meta: entry["meta"] = asdict(dev.get_manifest()) if include_state: entry["state"] = […] result.append(entry) return result </code> === POST /api/control/batch – Batch-Commands === <code python> @dataclass class BatchCommandReq: commands: List[CommandReq] @router.post("/api/control/batch") async def send_batch_control(request: Request, batch: BatchCommandReq): """Sendet mehrere Befehle atomar.""" engine = request.app.state.engine results = [] for cmd in batch.commands: try: await engine.execute_command(cmd.device_id, cmd.key, cmd.value) results.append({"device_id": cmd.device_id, "key": cmd.key, "status": "ok"}) except Exception as e: results.append({"device_id": cmd.device_id, "key": cmd.key, "status": "error", "detail": str(e)}) return {"results": results} </code> === GET /api/system/status === <code python> @router.get("/api/system/status") def api_system_status(request: Request): import psutil, os engine = request.app.state.engine process = psutil.Process(os.getpid()) devices_online = sum(1 for d in engine.device_manager.devices.values() if d.entities.get("state") and d.entities["state"].value == "ONLINE") return { "uptime_seconds": time.time() - engine._start_time, "memory_mb": round(process.memory_info().rss / 1024 / 1024, 1), "cpu_percent": process.cpu_percent(interval=0.1), "devices_total": len(engine.device_manager.devices), "devices_online": devices_online, "bus_subscribers": len(engine.bus._subscriptions), "cache_entries": len(engine.state_cache), "python_version": sys.version.split()[0], "platform": platform.platform() } </code> —- ===== 8. Frontend: Vue/JS Analyse ===== ==== 8.1 Strukturelle Verbesserungen ==== **Problem: Keine Composables / Shared Logic** Viele Views duplizieren den gleichen Code für Entity-Zugriff. Dieser Block existiert fast identisch in live_view.js, settings.js, multi_view.js, sensor.js: <code javascript> const device = computed[1]; const entityState = computed[2]; </code> **Lösung:** Ein shared Composable: <code javascript> static/js/composables/useDevice.js import { globalStore } from '/static/js/store.js'; export function useDevice(deviceId) { const { computed } = Vue; const device = computed[3]*

1)

) ⇒ globalStore.devices[props.deviceId]); const entityMeta = computed(() ⇒ device.value?.meta.entities.find(e ⇒ e.id === props.entity_id

2)

) ⇒ device.value?.state.find(s ⇒ s.sensor === props.entity_id

3)

) ⇒ globalStore.devices[deviceId.value || deviceId]);

```
  const isOnline = computed(() => {
      const s = device.value?.state?.find(s => s.sensor === 'state');
      return s?.value === 'ONLINE';
  });
  const getEntity = (entityId) => {
      return computed(() => ({
          meta: device.value?.meta?.entities?.find(e => e.id === entityId),
          state: device.value?.state?.find(s => s.sensor === entityId),
          value: device.value?.state?.find(s => s.sensor ===
entityId)?.value ?? null,
          timestamp: device.value?.state?.find(s => s.sensor ===
entityId)?.timestamp ?? null,
          alarmState: device.value?.state?.find(s => s.sensor ===
entityId)?.alarm_state ?? 'NORMAL'
      }));
  };
  const getDisplayValue = (entityId) => {
      return computed(() => {
          const s = device.value?.state?.find(x => x.sensor === entityId);
          if (!s || s.value === null) return '--';
          const meta = device.value?.meta?.entities?.find(e => e.id ===
entityId);
          if (meta?.ui?.ui_type === 'number' && typeof s.value === 'number')
{
              return Number(s.value).toFixed(meta.accuracy || 2);
          }
          if (meta?.ui?.ui_type === 'toggle') {
              return s.value ? 'ON' : 'OFF';
          }
          return s.value;
      });
  };
  return { device, isOnline, getEntity, getDisplayValue };
```

} </code> ==== 8.2 Store-Verbesserungen ====

```
// store.js — Erweitert um Computed Helper

export const globalStore = reactive({
    devices: {},
    wsConnected: false,
    wsReconnectCount: 0,
    lastMessageTime: null,

    getDeviceIds() {
        return Object.keys(this.devices);
    },

    getOnlineDevices() {
        return Object.values(this.devices).filter(d => {
            const s = d.state?.find(s => s.sensor === 'state');
```

```
            return s?.value === 'ONLINE';
        });
    },

    getEntityValue(deviceId, entityId) {
        const dev = this.devices[deviceId];
        if (!dev) return null;
        const s = dev.state?.find(s => s.sensor === entityId);
        return s?.value ?? null;
    }
});
```

==== 8.3 Fehlende Frontend-Views ====

View	Beschreibung	Priorität
`AlarmView`	Aktive Alarme, Historie, Quittierung	HOCH
`WaveformView`	Oszilloskop-Darstellung für WaveformEntity	MITTEL
`ScriptEditor`	Monaco-Editor für Automation Scripts mit Run/Stop	MITTEL
`SessionView`	Recording starten/stoppen, Sessions verwalten	MITTEL
`SystemDashboard`	RAM, CPU, Device-Übersicht, Bus-Statistiken	NIEDRIG
`HistoryView`	Historische Daten aus der DB abfragen und plotten	NIEDRIG (braucht DB)
`DeviceWizard`	Neues Gerät über UI hinzufügen (Config-Editor)	NIEDRIG

===== 9. Priorisierte Roadmap ===== ==== Phase 1: Fundament stabilisieren (1-2 Wochen) ====

Nr	Aufgabe	Dateien	Aufwand
1.1	EventBus Topic-Filtering einbauen	`core/event_bus.py`	2h
1.2	Entity `accept_value()` Validierung	`structures/entities/*.py`	3h
1.3	Store.js State als Map (Race-Condition Fix)	`static/js/store.js`	1h
1.4	`useDevice()` Composable extrahieren	`static/js/composables/useDevice.js`	2h
1.5	Manifest um capabilities & transport_type erweitern	`devices/base.py`	1h
1.6	Reconnect mit Exponential Backoff	`devices/transport/base.py`	1h
1.7	`EnumEntity` und `BitmaskEntity` implementieren	`structures/entities/logic.py`	3h

==== Phase 2: Persistenz & Historie (2-3 Wochen) ====

Nr	Aufgabe	Dateien	Aufwand
2.1	SQLite Storage Backend implementieren	`core/storage/sqlite_backend.py`	4h
2.2	Storage Worker in Engine integrieren (Batch-Insert)	`core/engine.py`	3h
2.3	History REST Endpoints	`web/rest.py`	3h
2.4	Data Retention Policy (Auto-Cleanup alter Daten)	`core/storage/sqlite_backend.py`	2h
2.5	Frontend: HistoryView mit Zeitbereichs-Auswahl	`static/views/history_view.js`	6h

==== Phase 3: Alarm-System (1-2 Wochen) ====

Nr	Aufgabe	Dateien	Aufwand
3.1	AlarmManager implementieren	`core/alarm_manager.py`	4h

Nr	Aufgabe	Dateien	Aufwand
3.2	`_evaluate_alarms()` mit AlarmManager verknüpfen	`structures/entities/physical.py`	2h
3.3	Alarm REST Endpoints	`web/rest.py`	2h
3.4	Alarm-Events über WebSocket an Frontend	`web/realtime.py`	2h
3.5	Frontend: AlarmView mit Sound-Feedback	`static/views/alarm_view.js`	4h
3.6	Alarm-Hysterese Logik verfeinern	`structures/entities/physical.py`	2h

==== Phase 4: Automation & Scripting (2-3 Wochen) ====

Nr	Aufgabe	Dateien	Aufwand
4.1	ScriptAPI und ScriptRunner implementieren	`automation/script_engine.py`	6h
4.2	Script REST Endpoints (list, run, stop)	`web/rest.py`	3h
4.3	Script-Log über WebSocket ans Frontend	`web/realtime.py`	2h
4.4	Frontend: ScriptEditor mit Ace/Monaco	`static/views/script_editor.js`	8h
4.5	Session Manager vollständig anbinden	`core/session_manager.py`	3h
4.6	Session Export (CSV, XLSX)	`web/rest.py`	4h

==== Phase 5: Transport & Hardware Erweiterungen (fortlaufend) ====

Nr	Aufgabe	Dateien	Aufwand
5.1	VISA/SCPI Transport (pyvisa)	`devices/transport/visa_transport.py`	6h
5.2	USB HID Transport fertigstellen	`devices/transport/hid_transport.py`	4h
5.3	UDP Transport	`devices/transport/udp_client.py`	3h
5.4	MQTT Transport (für IoT-Sensoren)	`devices/transport/mqtt_transport.py`	4h
5.5	WaveformView im Frontend	`static/views/waveform_view.js`	8h

==== Phase 6: Developer Experience & Qualität (fortlaufend) ====

Nr	Aufgabe	Dateien	Aufwand
6.1	Unit Tests für Entity-System	`tests/test_entities.py`	4h
6.2	Integration Tests für Transport/Framing	`tests/test_framing.py`	4h
6.3	API Tests mit httpx/TestClient	`tests/test_api.py`	3h
6.4	Treiber-Template-Generator (CLI Tool)	`tools/create_driver.py`	3h
6.5	OpenAPI Schema verfeinern (Pydantic Models)	`web/rest.py`	4h
6.6	Hot-Reload für Treiber (ohne Server-Neustart)	`core/device_manager.py`	6h

===== 10. Anhang: Zusätzliche Code-Beispiele ===== ==== 10.1 Treiber-Template-Generator ====

```
# tools/create_driver.py
"""
Generiert ein Treiber-Skelett basierend auf User-Input.
Aufruf: python -m tools.create_driver --name MyDevice --category psu --
transport serial
"""

TEMPLATE = '''"""
ionpy Driver: {class_name}
Auto-generiert am {date}
"""

from devices.base import AbstractDevice
from structures.enums import AccessMode, UIMode
```

```
from structures.metadata import UIMeta
from structures.entities.physical import NumericEntity
from structures.entities.logic import ToggleEntity, TextEntity

class {class_name}(AbstractDevice):
    """TODO: Beschreibung des Geräts"""

    STACK_BLUEPRINTS = {{
        "{transport}": {{
            "transport": {{"type": "{transport}", "params":
{{{transport_params}}}}},
            "framing": {{"type": "{framing}", "params": {{}}}}
        }}
    }}

    # SENSOREN (Read-Only)
    # example = NumericEntity(name="Beispiel", unit="V",
ui=UIMeta(group="Monitor"))

    # AKTOREN (Read-Write)
    # output = ToggleEntity(name="Ausgang", mode=AccessMode.RW,
ui=UIMeta(group="Steuerung"))

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # TODO: Command-Handler registrieren

    async def device_logic(self) -> bool:
        # TODO: Hardware-Kommunikation implementieren
        return False

    async def emergency_stop(self):
        self._logger.warning("NOTAUS empfangen!")
        # TODO: Hardware-Outputs deaktivieren
'''
```

==== 10.2 System-Status Widget (Frontend) ====

```
// static/views/system_status.js
export default {
    template: `
    <div style="padding: 15px; height: 100%; overflow: auto; background:
#18181b; color: #eee;">
        <div v-for="(val, key) in statusData" :key="key"
             style="display: flex; justify-content: space-between; padding:
6px 0;
                    border-bottom: 1px solid #27272a;">
            <span style="color: #a1a1aa;">{{ key }}</span>
            <span style="font-family: var(--font-mono); color: #0ea5e9;
                         font-weight: bold;">{{ val }}</span>
        </div>
```

```
        <div v-if="!statusData"
             style="color: #71717a; text-align: center; padding:
20px;">Lade...</div>
    </div>`,
    setup() {
        const { ref, onMounted, onUnmounted } = Vue;
        const statusData = ref(null);
        let timer;

        const fetchStatus = async () => {
            try {
                const res = await fetch('/api/system/status');
                statusData.value = await res.json();
            } catch (e) { console.error(e); }
        };

        onMounted(() => { fetchStatus(); timer = setInterval(fetchStatus,
5000); });
        onUnmounted(() => clearInterval(timer));

        return { statusData };
    }
};
```

==== 10.3 VISA Transport (für professionelle Messgeräte) ====

```
# devices/transport/visa_transport.py
"""
Hinweis für KI-Umsetzung:
- pyvisa muss installiert sein (pip install pyvisa pyvisa-py)
- Unterstützt USB-TMC, GPIB, TCP/LAN (LXI) und Serial
- Resource-String Beispiele:
    USB:   "USB0::0x1AB1::0x04CE::DS1ZA1234::INSTR"
    TCP:   "TCPIP::192.168.1.5::INSTR"
    GPIB:  "GPIB0::1::INSTR"
"""

import asyncio
from typing import Optional
from devices.transport.base import AbstractTransport

class VISATransport(AbstractTransport):
    def __init__(self, resource_string: str, timeout_ms: int = 5000,
**kwargs):
        super().__init__(**kwargs)
        self.resource_string = resource_string
        self.timeout_ms = timeout_ms
        self._instrument = None
        self._rm = None

    async def connect(self) -> bool:
```

```
        try:
            import pyvisa
            self._rm = pyvisa.ResourceManager('@py')
            self._instrument = self._rm.open_resource(self.resource_string)
            self._instrument.timeout = self.timeout_ms

            idn = self._instrument.query("*IDN?").strip()
            self.logger.info(f"VISA verbunden: {idn}")
            self.is_connected = True
            return True
        except Exception as e:
            self.logger.error(f"VISA Connect Fehler: {e}")
            self.is_connected = False
            return False

    async def disconnect(self):
        if self._instrument:
            try: self._instrument.close()
            except: pass
        if self._rm:
            try: self._rm.close()
            except: pass
        self._instrument = None
        self._rm = None
        self.is_connected = False

    async def write(self, data: bytes):
        if self._instrument and self.is_connected:
            try:
                loop = asyncio.get_event_loop()
                await loop.run_in_executor(None, self._instrument.write_raw,
data)
            except Exception as e:
                self.logger.error(f"VISA Write Fehler: {e}")
                self.is_connected = False

    async def read(self, n: int, timeout: float = 1.0) -> bytes:
        if not self._instrument or not self.is_connected:
            return b""
        try:
            loop = asyncio.get_event_loop()
            data = await asyncio.wait_for(
                loop.run_in_executor(None, self._instrument.read_raw, n),
                timeout=timeout
            )
            return data
        except asyncio.TimeoutError:
            return b""
        except Exception as e:
            self.logger.error(f"VISA Read Fehler: {e}")
            self.is_connected = False
```

```
            return b""

    def check_hardware_exists(self, verbose=False) -> bool:
        try:
            import pyvisa
            rm = pyvisa.ResourceManager('@py')
            resources = rm.list_resources()
            exists = self.resource_string in resources
            if not exists and verbose:
                self.logger.warning(f"VISA Resource '{self.resource_string}'
nicht gefunden.")
                if resources:
                    self.logger.info("Verfügbare VISA Resources:")
                    for r in resources:
                        self.logger.info(f"  -> {r}")
            rm.close()
            return exists
        except Exception:
            return False
```

==== 10.4 Unit-Test Grundstruktur ====

```
# tests/test_entities.py
"""
Hinweis für KI-Umsetzung:
- pytest verwenden (pip install pytest pytest-asyncio)
- Jede Entity-Klasse braucht Tests für:
  1. Initialisierung mit Defaults
  2. Wert-Setzung und Validierung
  3. to_dict() Serialisierung
  4. create_sample() Korrektheit
  5. Edge-Cases (None, NaN, leere Strings)
"""

import pytest
import math
from structures.entities.physical import NumericEntity
from structures.entities.logic import ToggleEntity, SelectEntity
from structures.metadata import LinearCalibration, AlarmConfig
from structures.enums import AccessMode, AlarmState

class TestNumericEntity:
    def test_basic_creation(self):
        ent = NumericEntity(name="Spannung", unit="V", accuracy=2)
        ent.id = "voltage"
        assert ent.name == "Spannung"
        assert ent.unit == "V"
        assert ent.value is None

    def test_calibration(self):
        calib = LinearCalibration(factor=0.01, offset=0)
```

```
        ent = NumericEntity(name="V", unit="V", calib=calib)
        ent.id = "v"
        ent.update_real_value(1250)
        assert ent.value == 12.5

    def test_alarm_evaluation(self):
        alarms = AlarmConfig(hihi=30.0, hi=25.0, lo=5.0, lolo=2.0)
        ent = NumericEntity(name="Temp", unit="°C", alarms=alarms)
        ent.id = "temp"

        ent.update_real_value(20.0)
        assert ent.alarm_state == AlarmState.NORMAL

        ent.update_real_value(26.0)
        assert ent.alarm_state == AlarmState.HI

        ent.update_real_value(31.0)
        assert ent.alarm_state == AlarmState.HIHI

    def test_nan_handling(self):
        ent = NumericEntity(name="X", unit="")
        ent.id = "x"
        ent.update_real_value(float('nan'))
        sample = ent.create_sample("test_dev")
        assert sample.value == 0.0

    def test_limits_validation(self):
        ent = NumericEntity(name="V", unit="V", mode=AccessMode.RW,
hw_max=50.0)
        ent.id = "v"
        with pytest.raises(ValueError):
            ent.validate_and_set_target(60.0)

    def test_to_dict_completeness(self):
        ent = NumericEntity(name="V", unit="V", hw_min=0, hw_max=50,
accuracy=3)
        ent.id = "voltage"
        d = ent.to_dict()
        assert "hw_min" in d
        assert "hw_max" in d
        assert "accuracy" in d
        assert d["type"] == "NumericEntity"


class TestToggleEntity:
    def test_toggle_values(self):
        ent = ToggleEntity(name="Switch", mode=AccessMode.RW)
        ent.id = "sw"
        ent.value = True
        sample = ent.create_sample("dev")
        assert sample.value == "ON"
```

```
        ent.value = False
        sample = ent.create_sample("dev")
        assert sample.value == "OFF"


class TestSelectEntity:
    def test_valid_selection(self):
        ent = SelectEntity(name="Mode", options=["A", "B", "C"],
mode=AccessMode.RW)
        ent.id = "mode"
        ent.validate_and_set("B")
        assert ent.value == "B"

    def test_invalid_selection(self):
        ent = SelectEntity(name="Mode", options=["A", "B"],
mode=AccessMode.RW)
        ent.id = "mode"
        with pytest.raises(ValueError):
            ent.validate_and_set("D")
```

===== Fazit ===== ionpy ist ein technisch ambitioniertes Projekt mit einer soliden Architektur-Basis. Die deklarative Entity-Definition, der modulare Transport-Stack und das professionelle Frontend sind bemerkenswert. Die größten Hebel für die nächste Entwicklungsstufe sind:

1. **Persistenz** (SQLite → später InfluxDB/TimescaleDB)
2. **Automation Engine** (Scripting für automatisierte Tests)
3. **Alarm-Routing** (Von der Berechnung zum Benachrichtigungssystem)
4. **Entity-Validierung** (Jede Entity validiert ihre eigenen Eingaben)
5. **Test-Coverage** (Aktuell: 0% → Ziel: >60% für Core-Module)

Die Software hat das Potenzial, sich von einem persönlichen Lab-Tool zu einer vollwertigen Open-Source-SCADA/Lab-Plattform zu entwickeln.