ChatGPT

Modular Thorlabs TDC001 Controller Codebase and Generic Device GUI Boilerplate

Modularized Thorlabs TDC001 Controller Codebase

The Thorlabs TDC001 controller code has been reorganized into distinct modules by functionality. Below is the codebase structure with each module's purpose:

- `discovery.py` – Discovers TDC001 devices via USB and scans the local network (LAN) for available backend APIs.
- `api_client.py` – Provides a Python client to call the FastAPI backend (for listing ports, moving the device, etc.).
- `storage.py` – Saves and loads persistent state (last known positions and settings) to a JSON file for session restore.
- `constants.py` – Defines motion **step presets** (encoder counts per mm for different hardware) and unit conversion factors.
- `main_window.py` – Implements the PyQt6 main window UI (network backend selection, device controls, status polling, and session restore logic).
- `popups.py` – Helper dialogs (QMessageBox) for session restore prompts and safety warnings (lost power, moved device).
- `worker.py` – A QThread worker class to run backend API calls asynchronously (keeping the UI responsive).
- `controller.py` – Wrapper class for the TDC001 hardware (uses the Thorlabs APT API to control the device).
- `server.py` – FastAPI web server that exposes the TDC001 control functions (connect, move, home, etc.) as REST API endpoints.
- `run.py` – Application entry-point script that launches the PyQt6 GUI (with optional noVNC support for headless Docker environments).
- `Dockerfile.gui` – Docker build file for the GUI container (PyQt6 + Xvfb/noVNC setup).

Below are the contents of each module:

1. Device Discovery and Scanning (including LAN sweep)

File: `discovery.py`

```
"""Device discovery and scanning for TDC001 (includes serial port and network
LAN scan)."""

import socket
import requests
```

```
from concurrent.futures import ThreadPoolExecutor
from serial.tools import list_ports
from serial.tools.list_ports_common import ListPortInfo
from typing import List

def find_tdc001_ports(
    *,
    vendor_ids: tuple[int, ...] = (0x0403, 0x1313),   # common FTDI / Thorlabs
USB VIDs
    serial_prefix: str = "83",                       # TDC001 cubes usually
start with 83--
) -> List[str]:
    """Return *device strings* (e.g. `/dev/ttyUSB0`) for attached TDC001
cubes."""
    ports: List[str] = []                             # → collected matches

    for p in list_ports.comports():                   # → every serial device
        if p.vid not in vendor_ids:                   # ✱ vendor mismatch
            continue
        if p.serial_number is None:                   # ✱ no serial → skip
            continue
        if not p.serial_number.startswith(serial_prefix):  # ✱ not a cube
            continue
        ports.append(p.device)                        # ✱ good → save
    return ports                                      # → hand back list

def scan_for_backends(port: int = 8000, timeout: float = 0.25, workers: int =
64) -> List[str]:
    """
    Scans the local /24 for running backends responding to /ping.
    """
    # determine local subnet prefix
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    try:
        # use a UDP connect to get our local IP
        s.connect(("8.8.8.8", 80))
        local_ip = s.getsockname()[0]
    finally:
        s.close()

    prefix = ".".join(local_ip.split('.')[:-1])

    def check_host(i: int) -> str:
        url = f"http://{prefix}.{i}:{port}"
        try:
            r = requests.get(f"{url}/ping", timeout=timeout)
            if r.status_code == 200:
```

```
                return url
        except requests.RequestException:
            pass
        return None

    with ThreadPoolExecutor(max_workers=workers) as executor:
        results = executor.map(check_host, range(1, 255))
    return [u for u in results if u]
```

2. Backend API Client Abstraction

File: `api_client.py`

```
import requests
from typing import List

class APIClient:
    """Thin wrapper over the FastAPI-based TDC001 backend."""
    def __init__(self, base_url: str):
        self.base = base_url.rstrip("/")
        self.session = requests.Session()

    def _url(self, path: str) -> str:
        return f"{self.base}/{path.lstrip('/')}"

    def _req(self, method: str, path: str, timeout: float = 3, **kwargs):
        r = self.session.request(method, self._url(path), timeout=timeout,
**kwargs)
        r.raise_for_status()
        return r.json() if r.content else {}

    def list_ports(self) -> List[str]:
        return self._req("GET", "/ports")

    def status(self) -> dict:
        return self._req("GET", "/status")

    def connect(self, port: str):
        return self._req("POST", "/connect", json={"port": port})

    def move_rel(self, steps: int):
        return self._req("POST", "/move_rel", json={"steps": steps},
timeout=150)

    def move_abs(self, position: int):
        return self._req("POST", "/move_abs", json={"position": position},
```

```
timeout=150)

    def home(self):
        return self._req("POST", "/home", timeout=120)

    def flash(self):
        return self._req("POST", "/identify")

    def stop(self):
        return self._req("POST", "/stop")
```

3. Persistent Settings and Session Restore

File: `storage.py`

```
"""Persistent session state storage for TDC001 (settings and positions)."""

from pathlib import Path
import json

# where we persist state (settings + positions)
STORAGE_PATH = Path.home() / ".tdc001_state.json"

def load_state() -> dict:
    """
    Load complete state from disk, returning a dict with "settings" and
"positions".
    """
    try:
        return json.loads(STORAGE_PATH.read_text())
    except FileNotFoundError:
        return {"settings": {}, "positions": {}}

def save_state(state: dict) -> None:
    """
    Atomically write the full state (settings + positions) back to disk.
    """
    STORAGE_PATH.parent.mkdir(parents=True, exist_ok=True)
    STORAGE_PATH.write_text(json.dumps(state, indent=2))

def load_positions() -> dict:
    """
    Return the positions sub-dict from saved state.
    """
    return load_state().get("positions", {})
```

```python
def save_positions(positions: dict) -> None:
    """
    Save only the positions into the state file, preserving existing settings.
    """
    state = load_state()
    state["positions"] = positions
    save_state(state)

def load_settings() -> dict:
    """
    Return the settings sub-dict from saved state.
    """
    return load_state().get("settings", {})

def save_settings(settings: dict) -> None:
    """
    Save only the settings into the state file, preserving existing positions.
    """
    state = load_state()
    state["settings"] = settings
    save_state(state)
```

File: `constants.py`

```python
"""Constants for TDC001 UI presets and unit conversion."""

# preset encoder counts per mm
STEP_PRESETS = {
    "T-Cube 0.5 mm lead": 51200,
    "T-Cube 1.0 mm lead": 25600,
    "MTS28-Z8": 34555,
    "Manual set…": None,
}

# unit conversion factors to mm
UNIT_FACT = {
    "mm": 1.0,
    "µm": 1e-3,
}
```

4. Modular UI Layout and Controls

File: `main_window.py`

```python
#!/usr/bin/env python3
"""
Main Window Module

This module defines the MainWindow class, which provides a PyQt6 graphical user interface
for controlling a Thorlabs TDC001 device over a networked backend. It refactors the v1.8 UI,
adds full session-restore on boot (backend, port, preset, position), and only issues
lost-power/moved warnings once the device is truly idle. It also auto-returns to your
last position without flicker and preserves your chosen steps/mm preset.
"""

import sys
import datetime
import socket

from PyQt6.QtGui import QIcon, QIntValidator, QDoubleValidator
from PyQt6.QtWidgets import (
    QApplication, QMainWindow, QWidget, QLabel, QComboBox, QLineEdit,
    QPushButton, QHBoxLayout, QVBoxLayout, QGridLayout, QFormLayout,
    QGroupBox, QStatusBar, QMessageBox
)
from PyQt6.QtCore import QTimer, QThread

from api_client import APIClient
from discovery import scan_for_backends
from constants import STEP_PRESETS, UNIT_FACT
from storage import load_positions, save_positions, load_settings, save_settings
from popups import ask_restore_session, ask_restore_preset, warn_lost_power, warn_moved
from worker import Worker

class MainWindow(QMainWindow):
    """Main UI class with boot-time session restore and safe-state checks."""

    def __init__(self, backend_hint=None):
        super().__init__()
        # Window setup
        self.setWindowTitle("Thorlabs TDC001 Controller")
        self.setWindowIcon(QIcon.fromTheme("applications-engineering"))
        self.resize(900, 600)

        # State
        self.api = None
```

```
        self.steps_per_mm = STEP_PRESETS["T-Cube 0.5 mm lead"]
        self.positions = load_positions()      # { "backend|port": {pos, time,
steps_per_mm, homed} }
        self.settings = load_settings()        # { backend, port, preset,
steps_per_mm, date }
        self.session_restored = False
        self._did_post_connect_warn = False

        # Input validators
        self.int_val = QIntValidator(1, 10**6, self)
        self.fl_val  = QDoubleValidator(0.0, 1e6, 6, self)

        # Build UI, discover backends, restore session, start polling
        self._build_ui()
        self._discover_backends(backend_hint)
        self._status_timer = QTimer(self)
        self._status_timer.timeout.connect(self._refresh_status)
        self._status_timer.start(500)
        self._maybe_restore_session()

    def _maybe_restore_session(self):
        """
        On startup, if we have saved backend+port, offer to restore:
          1) backend URL
          2) cube-port
          3) steps/mm preset (or Custom)
          4) return to last absolute position
        """
        # pull saved session settings
        saved_backend = self.settings.get("backend")
        saved_port    = self.settings.get("port")
        saved_preset  = self.settings.get("preset", "Custom")
        saved_spm     = self.settings.get("steps_per_mm", self.steps_per_mm)
        saved_date    = self.settings.get("date", "")

        # nothing to restore?
        if not (saved_backend and saved_port):
            return

        # backend must still be in our dropdown
        if self.cmb_backend.findText(saved_backend) < 0:
            return

        # ask the user
        if not ask_restore_session(
            self,
            saved_backend,
            saved_port,
```

```
            saved_preset,
            saved_spm,
            saved_date,
        ):
            return

        # mark that we *have* just done a boot-restore
        self.session_restored = True

        # 1) pick the saved backend & reload ports
        self.cmb_backend.setCurrentText(saved_backend)
        self._on_backend_change(saved_backend)

        # 2) pick the saved port
        self.cmb_port.setCurrentText(saved_port)

        # 3) restore the preset (or custom)
        if saved_preset in STEP_PRESETS:
            self.cmb_preset.setCurrentText(saved_preset)
        else:
            # force “Custom” and fill in the numeric value
            self.cmb_preset.setCurrentText("Custom")
            self.ed_steps.setText(str(saved_spm))

        # 4) call your existing preset handler so everything stays in sync
        self._on_preset(self.cmb_preset.currentText())

        # 5) connect the device (this also saves the settings again)
        self._connect_device()

        # 6) do one immediate status refresh to avoid flicker
        self._refresh_status()

        # 7) auto-move back to the last saved absolute position
        key = f"{self.api.base}|{saved_port}"
        last = self.positions.get(key)
        if last and "pos" in last:
            # no need to QTimer this if you don't mind the slight delay
            QTimer.singleShot(200, lambda: self._run_async(self.api.move_abs,
last["pos"]))

        # 8) schedule your lost-power / moved-elsewhere safety check
        self._did_post_connect_warn = False
        QTimer.singleShot(500, self._check_post_connect)

    def _connect_device(self):
        """
```

```
        Called when user selects a cube-port:
        - Connects to API
        - Saves session settings
        - Schedules lost-power/moved warnings once device is idle
        """
        port = self.cmb_port.currentText().strip()
        if not port:
            return

        try:
            self.api.connect(port)
            self.statusbar.showMessage("Connected", 2000)
        except Exception as e:
            QMessageBox.critical(self, "Error", f"/connect failed:\n{e}")
            return

        # Persist connection settings for next boot
        self.settings.update({
            "backend":      self.api.base,
            "port":         port,
            "preset":       self.cmb_preset.currentText(),
            "steps_per_mm": self.steps_per_mm,
            "date":         datetime.datetime.now().isoformat()
        })
        save_settings(self.settings)

        # Reset warning guard and schedule safety check
        self._did_post_connect_warn = False
        QTimer.singleShot(500, self._check_post_connect)

    def _check_post_connect(self):
        """
        After connecting:
        1) If busy (initializing/moving), retry in 200 ms
        2) Once idle and not yet warned:
           • If previously homed but now un-homed → lost-power
           • Else if homed and position differs → moved-elsewhere
        """
        port = self.cmb_port.currentText().strip()
        key  = f"{self.api.base}|{port}"
        last = self.positions.get(key)
        if not last or self._did_post_connect_warn:
            return

        st = self.api.status()
        curr_pos   = st["position"]
        curr_homed = st["homed"]
```

```
        busy = st["moving_forward"] or st["moving_reverse"]

        # Retry if still busy
        if busy:
            QTimer.singleShot(200, self._check_post_connect)
            return

        # Fetch saved data
        last_pos   = last["pos"]
        last_homed = last.get("homed", False)
        last_steps = last.get("steps_per_mm", self.steps_per_mm)
        last_mm    = last_pos / last_steps
        last_time  = last["time"]

        # Lost-power case
        if last_homed and not curr_homed:
            self._did_post_connect_warn = True
            if warn_lost_power(self, last_pos, last_mm, last_time):
                def after_home(res, err):
                    if not err:
                        self._run_async(self.api.move_abs, last_pos)
                w = Worker(self.api.home)
                t = QThread(self)
                w.moveToThread(t)
                t.started.connect(w.run)
                w.finished.connect(after_home)
                t.start()

        # Moved-elsewhere case
        if curr_homed and abs(curr_pos - last_pos) > 2:
            self._did_post_connect_warn = True
            curr_mm = curr_pos / self.steps_per_mm
            if warn_moved(self, last_pos, curr_pos, last_mm, curr_mm,
last_time):
                self._run_async(self.api.move_abs, last_pos)

    def _build_ui(self):
        """Construct all widgets, layouts, and connect signals."""
        central = QWidget(self)
        self.setCentralWidget(central)
        main_v = QVBoxLayout(central)
        main_v.setSpacing(10)
        main_v.setContentsMargins(10, 10, 10, 10)

        # Status row
        st_h = QHBoxLayout()
        self.lbl_status = QLabel("Status: —")
```

```
        self.lbl_homed   = QLabel("Homed: ✗")
        self.lbl_pos     = QLabel("Pos: — cnt | — mm")
        for lbl in (self.lbl_status, self.lbl_homed, self.lbl_pos):
            st_h.addWidget(lbl)
        st_h.addStretch()
        main_v.addLayout(st_h)

        # Network group
        net_g = QGroupBox("Network")
        net_f = QFormLayout(net_g)
        self.cmb_backend = QComboBox()
        self.cmb_backend.setEditable(True)
        self.cmb_backend.setPlaceholderText("http://<ip>:8000")
        self.cmb_backend.currentTextChanged.connect(self._on_backend_change)
        btn_add = QPushButton("Add Backend")
        btn_add.clicked.connect(self._add_backend)
        self.cmb_port = QComboBox()
        self.cmb_port.currentIndexChanged.connect(self._connect_device)
        net_f.addRow("Backend:", self.cmb_backend)
        net_f.addRow("", btn_add)
        net_f.addRow("Cube port:", self.cmb_port)
        main_v.addWidget(net_g)

        # Motion group
        mot_g = QGroupBox("Motion")
        grid = QGridLayout(mot_g)
        grid.setHorizontalSpacing(8)
        grid.setVerticalSpacing(4)

        # Steps/mm preset
        grid.addWidget(QLabel("Steps/mm preset:"), 0, 0)
        self.cmb_preset = QComboBox()
        self.cmb_preset.addItems(STEP_PRESETS.keys())
        self.cmb_preset.currentTextChanged.connect(self._on_preset)
        grid.addWidget(self.cmb_preset, 0, 1)
        self.ed_steps = QLineEdit(str(self.steps_per_mm))
        self.ed_steps.setValidator(self.int_val)
        grid.addWidget(self.ed_steps, 0, 2)

        # Relative move
        grid.addWidget(QLabel("Move relative:"), 1, 0)
        self.ed_rel = QLineEdit("0"); self.ed_rel.setValidator(self.fl_val)
        grid.addWidget(self.ed_rel, 1, 1)
        self.cmb_unit_rel = QComboBox();
self.cmb_unit_rel.addItems(UNIT_FACT.keys())
        grid.addWidget(self.cmb_unit_rel, 1, 2)
        btn_neg = QPushButton("−"); btn_neg.clicked.connect(lambda:
```

```
self._move_rel(-1))
        btn_pos = QPushButton("+"); btn_pos.clicked.connect(lambda:
self._move_rel(1))
        grid.addWidget(btn_neg, 1, 3); grid.addWidget(btn_pos, 1, 4)

        # Absolute move
        grid.addWidget(QLabel("Move absolute:"), 2, 0)
        self.ed_abs = QLineEdit("0"); self.ed_abs.setValidator(self.fl_val)
        grid.addWidget(self.ed_abs, 2, 1)
        self.cmb_unit_abs = QComboBox();
self.cmb_unit_abs.addItems(UNIT_FACT.keys())
        grid.addWidget(self.cmb_unit_abs, 2, 2)
        btn_go = QPushButton("Go to absolute");
btn_go.clicked.connect(self._move_abs)
        grid.addWidget(btn_go, 2, 3, 1, 2)

        # Control buttons
        btn_home = QPushButton("Home");  btn_home.clicked.connect(lambda:
self._run_async(self.api.home))
        btn_flash = QPushButton("Flash"); btn_flash.clicked.connect(lambda:
self._run_async(self.api.flash))
        btn_stop = QPushButton("STOP")
        btn_stop.setStyleSheet("background:#d9534f;color:white;font-
weight:bold;")
        btn_stop.clicked.connect(lambda: self._run_async(self.api.stop))
        grid.addWidget(btn_home, 3, 0); grid.addWidget(btn_flash, 3, 1)
        grid.addWidget(btn_stop, 3, 2, 1, 3)

        main_v.addWidget(mot_g)
        main_v.addStretch()

        # Status bar
        self.statusbar = QStatusBar(self)
        self.setStatusBar(self.statusbar)

    def _discover_backends(self, hint=None):
        """Populate backend dropdown: hint, localhost, docker alias, LAN
scan."""
        urls = []
        if hint and hint != "auto":
            urls.append(hint)
        else:
            urls += ["http://127.0.0.1:8000", "http://host.docker.internal:
8000"]
            try:
                ip = socket.gethostbyname(socket.gethostname())
                urls.append(f"http://{ip}:8000")
            except:
```

```
                pass
            urls += scan_for_backends()
        for u in dict.fromkeys(urls):
            self.cmb_backend.addItem(u)
        if self.cmb_backend.count():
            self._on_backend_change(self.cmb_backend.currentText())

    def _add_backend(self):
        """Save user-typed backend URL into dropdown."""
        url = self.cmb_backend.currentText().strip()
        if url and self.cmb_backend.findText(url) == -1:
            self.cmb_backend.addItem(url)
            self.cmb_backend.setCurrentText(url)

    def _on_backend_change(self, url):
        """When backend changes, fetch available cube-ports."""
        if not url:
            return
        self.api = APIClient(url)
        self.statusbar.showMessage("Loading ports...", 2000)
        try:
            ports = self.api.list_ports()
        except Exception as e:
            QMessageBox.critical(self, "Error", f"/ports failed:\n{e}")
            ports = []
        self.cmb_port.clear()
        self.cmb_port.addItems(ports)
        self.statusbar.showMessage(f"Ports: {ports}", 2000)

    def _on_preset(self, name):
        """Handle preset change: update steps_per_mm and save."""
        val = STEP_PRESETS.get(name)
        if val is None:
            self.ed_steps.setReadOnly(False)
        else:
            self.steps_per_mm = val
            self.ed_steps.setText(str(val))
            self.ed_steps.setReadOnly(True)
        self.settings["preset"]       = name
        self.settings["steps_per_mm"] = self.steps_per_mm
        save_settings(self.settings)

    def _to_counts(self, txt, cmb):
        """Convert text+unit into encoder counts."""
        try:
            dist = float(txt) * UNIT_FACT[cmb.currentText()]
        except:
            dist = 0.0
```

```
        try:
            self.steps_per_mm = int(self.ed_steps.text())
        except:
            pass
        return round(dist * self.steps_per_mm)

    def _move_rel(self, sign):
        """Move relative in background thread."""
        cnt = sign * abs(self._to_counts(self.ed_rel.text(), self.cmb_unit_rel))
        self.statusbar.showMessage("Moving relative...", 2000)
        self._run_async(self.api.move_rel, cnt)

    def _move_abs(self):
        """Move absolute in background thread."""
        cnt = self._to_counts(self.ed_abs.text(), self.cmb_unit_abs)
        self.statusbar.showMessage("Moving absolute...", 2000)
        self._run_async(self.api.move_abs, cnt)

    def _refresh_status(self):
        """Poll backend, update labels, and persist if idle & homed."""
        if not self.api:
            self.lbl_status.setText("Status: no backend")
            self.lbl_homed.setText("Homed: ✗")
            return
        try:
            st = self.api.status()
        except Exception as e:
            self.lbl_status.setText(f"Status: ⚠ {e}")
            return
        busy = st["moving_forward"] or st["moving_reverse"]
        homed_flag = st["homed"]
        pos = st["position"]
        self.lbl_status.setText("Status: moving" if busy else "Status: idle")
        self.lbl_homed.setText(f"Homed: {'✓' if homed_flag else '✗'}")
        mm = pos / self.steps_per_mm
        self.lbl_pos.setText(f"Pos: {pos} cnt | {mm:.3f} mm")
        if homed_flag and not busy:
            port = self.cmb_port.currentText().strip()
            key  = f"{self.api.base}|{port}"
            self.positions[key] = {
                "pos": pos,
                "time": datetime.datetime.now().isoformat(),
                "steps_per_mm": self.steps_per_mm,
                "homed": True
            }
            save_positions(self.positions)
```

```
    def _run_async(self, fn, *args):
        """Run a backend call in a Worker/QThread to keep UI responsive."""
        if not callable(fn):
            return
        worker = Worker(fn, *args)
        thread = QThread(self)
        worker.moveToThread(thread)
        thread.started.connect(worker.run)
        worker.finished.connect(lambda res, err: self._on_done(thread, worker,
res, err))
        thread.start()

    def _on_done(self, thread, worker, res, err):
        """Cleanup after task and report errors or 'Done'."""
        thread.quit()
        thread.wait()
        worker.deleteLater()
        thread.deleteLater()
        if err:
            QMessageBox.critical(self, "Error", str(err))
        else:
            self.statusbar.showMessage("Done", 2000)
            self._refresh_status()

if __name__ == "__main__":
    from run import main
    main()
```

File: `popups.py`

```
"""Popup dialog helpers for user prompts and warnings in TDC001 UI."""

import datetime
from PyQt6.QtWidgets import QMessageBox

def _format_date(iso_dt: str) -> str:
    """
    Convert an ISO-8601 timestamp into US-style MM/DD/YYYY hh:mm:ss AM/PM.
    Falls back to the raw string on parse errors.
    """
    try:
        dt = datetime.datetime.fromisoformat(iso_dt)
        return dt.strftime("%m/%d/%Y %I:%M:%S %p")
    except Exception:
        return iso_dt
```

```
def ask_restore_session(
    parent,
    backend: str,
    port: str,
    preset_name: str,
    steps_per_mm: int,
    iso_dt: str
) -> bool:
    """
    On startup, ask whether to restore the last session.
    Shows backend, port, named preset, numeric steps/mm, and US-formatted
timestamp.
    """
    ts = _format_date(iso_dt)
    msg = (
        "Would you like to restore your last session?\n\n"
        f"Backend: {backend}\n"
        f"Port: {port}\n"
        f"Preset: {preset_name}\n"
        f"Steps/mm: {steps_per_mm}\n"
        f"Last used: {ts}"
    )
    choice = QMessageBox.question(
        parent,
        "Restore session",
        msg,
        QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No
    )
    return (choice == QMessageBox.StandardButton.Yes)

def ask_restore_preset(
    parent,
    old_steps: int,
    iso_dt: str
) -> bool:
    """
    When manually selecting a previously-used backend+port,
    ask if you'd like to restore the old steps/mm setting.
    """
    ts = _format_date(iso_dt)
    msg = (
        f"You previously used Steps/mm = {old_steps} on {ts} for this device.
\n\n"
        "Would you like to restore that preset now?"
    )
    choice = QMessageBox.question(
        parent,
        "Restore preset",
```

```
        msg,
        QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No
    )
    return (choice == QMessageBox.StandardButton.Yes)

def warn_lost_power(
    parent,
    last_pos: int,
    last_mm: float,
    iso_dt: str
) -> bool:
    """
    Warn that the device was powered off (homed=False).
    Offer to home and return to the last saved position.
    """
    ts = _format_date(iso_dt)
    msg = (
        "It appears the device was powered off since last use.\n"
        f"You were at {last_pos} cnt ({last_mm:.3f} mm) on {ts}.\n\n"
        "You must home before absolute moves.  Home and return to that
position?"
    )
    choice = QMessageBox.question(
        parent,
        "Device lost power",
        msg,
        QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No
    )
    return (choice == QMessageBox.StandardButton.Yes)

def warn_moved(
    parent,
    last_pos: int,
    curr_pos: int,
    last_mm: float,
    curr_mm: float,
    iso_dt: str
) -> bool:
    """
    Warn that the device has been moved elsewhere since last use.
    Offer to return to your last saved position.
    """
    ts = _format_date(iso_dt)
    msg = (
        f"Device has moved since {ts}:\n"
        f"  Last: {last_pos} cnt ({last_mm:.3f} mm)\n"
        f"  Now:  {curr_pos} cnt ({curr_mm:.3f} mm)\n\n"
        "Would you like to return it to your last saved position?"
```

```
    )
    choice = QMessageBox.question(
        parent,
        "Device moved",
        msg,
        QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No
    )
    return (choice == QMessageBox.StandardButton.Yes)
```

5. Worker Thread System for Async API Calls

File: `worker.py`

```
"""Background worker thread for asynchronous device API calls (using
QThread)."""

from PyQt6.QtCore import QObject, pyqtSignal

class Worker(QObject):
    """Runs any function in a QThread and emits (result, error)."""
    finished = pyqtSignal(object, object)

    def __init__(self, fn, *args):
        super().__init__()
        self.fn = fn
        self.args = args

    def run(self):
        try:
            res = self.fn(*self.args)
            err = None
        except Exception as e:
            res, err = None, e
        self.finished.emit(res, err)
```

6. Boot-Time Automation (Restore Position, Device Homing, etc.)

Boot-time automation is handled in the UI and backend logic above. On startup, the **MainWindow** will attempt to restore the last session (backend URL, device port, preset, and position) via the `_maybe_restore_session` method. If a saved session exists, it automatically reconnects to the device, reapplies the preset (steps/mm), and triggers an absolute move back to the last saved position. After reconnection, the UI schedules safety checks (`_check_post_connect`) to warn if the device was powered off (not homed anymore) or moved since last use. If the device lost power, the code offers to home it and return to the last position; if it moved, it offers to move it back. These prompts are implemented in

popups.py (e.g. `warn_lost_power`, `warn_moved`). The backend's startup logic will also automatically connect to the first detected TDC001 device and ensure it is homed/enabled as needed.

7. Startup Entrypoint and Docker Configuration

File: `run.py` – This script initializes the Qt application, optionally launches Xvfb and websockify for noVNC (if `USE_NOVNC` is set), and opens the main window:

```
"""Startup entry point for the TDC001 GUI application (supports noVNC in
Docker)."""

import os, sys, argparse, shutil, subprocess
from PyQt6.QtWidgets import QApplication
from main_window import MainWindow

def main():
    p = argparse.ArgumentParser(description="TDC001 Docker-UI launcher")
    p.add_argument("--backend", default=os.getenv("BACKEND_URL", "auto"))
    args = p.parse_args()

    # optional Xvfb+noVNC support for headless Docker
    if os.getenv("USE_NOVNC"):
        if shutil.which("Xvfb") and shutil.which("websockify"):
            subprocess.Popen(["Xvfb", ":99", "-screen", "0", "1280x800x24"])
            subprocess.Popen(["websockify", "6080", "localhost:5900"])

    app = QApplication(sys.argv)
    w   = MainWindow(args.backend)
    w.show()
    sys.exit(app.exec())

if __name__ == "__main__":
    main()
```

File: `Dockerfile.gui` – Example Dockerfile for the GUI container (Python + PyQt6 + X11 & noVNC setup):

```
FROM python:3.10-slim

# Environment settings for X (disable MIT-SHM) and optional backend URL
ENV QT_X11_NO_MITSHM=1

# Install X11 server, VNC, websockify, and minimal window manager
RUN apt-get update && apt-get install -y \
    xvfb x11vnc websockify novnc fluxbox xauth libgl1-mesa-glx \
    && rm -rf /var/lib/apt/lists/*
```

```
# Install Python dependencies for GUI
RUN pip install PyQt6 requests

# Copy application code
COPY . /app
WORKDIR /app

# Launch the GUI (noVNC will be enabled via USE_NOVNC env variable)
CMD ["python", "run.py"]
```

Docker Compose Snippet – Frontend container configuration (serving noVNC on port 6080 and linking to the backend):

```
services:
  tdc001_ui:
    build: ./tdc001_ui
    environment:
      - USE_NOVNC=1
# launch Xvfb and websockify for browser access
      - BACKEND_URL=http://tdc001_api:8000
# backend service URL (or "auto" for LAN scan)
    ports:
      - "6080:6080"                    # noVNC web interface
    depends_on:
      - tdc001_api
```

Generic Device Controller GUI Boilerplate

The following is a generalized version of the codebase above, designed to be a reusable boilerplate for any lab device. All modules share the same structure as the TDC001-specific version, but with generic naming and placeholder implementations where device-specific logic is required. Clear `TODO` comments indicate where to adapt the code for a particular hardware API:

- `discovery.py` – Scans for connected devices (e.g. serial ports) and for available backend servers on the local network.
- `api_client.py` – Thin client for sending commands to the device's FastAPI backend.
- `storage.py` – Persists session settings and device positions in a JSON file (for restoring state across runs).
- `constants.py` – Example presets and unit conversion factors (to be adjusted for the specific device's units).
- `main_window.py` – PyQt6 main window GUI, providing network selection, device control UI, continuous status updates, and session restoration.

- `popups.py` – Generic dialog prompts for session restore and device safety checks (may be modified depending on device behavior).
- `worker.py` – Threaded worker for executing backend calls asynchronously (same as in TDC001 version).
- `controller.py` – Abstract controller class for the device. This is where you integrate the actual device API (all motion commands are defined here).
- `server.py` – FastAPI server exposing generic device control endpoints. Adapt this to call your device's controller methods.
- `run.py` – Startup script for the GUI application (unchanged logic, aside from name).
- `Dockerfile.gui` – Dockerfile for building the generic GUI container.

Below are the boilerplate code files:

1. Device Discovery and Scanning

File: `discovery.py`

```
"""Device discovery: local device scanning and LAN backend detection."""

import socket
import requests
from concurrent.futures import ThreadPoolExecutor
from serial.tools import list_ports
from typing import List

def find_devices() -> List[str]:
    """Return a list of connected device ports (e.g., serial ports) on this
machine."""
    ports = []
    for p in list_ports.comports():
        ports.append(p.device)
    # TODO: add filtering by vendor or device signature if needed
    return ports

def scan_for_backends(port: int = 8000, timeout: float = 0.25, workers: int =
64) -> List[str]:
    """Scan the local network (/24 subnet) for backend servers listening on the
given port."""
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    try:
        s.connect(('8.8.8.8', 80))
        local_ip = s.getsockname()[0]
    finally:
        s.close()
    prefix = '.'.join(local_ip.split('.')[:-1])

    def check_host(i: int) -> str:
```

```
        url = f'http://{prefix}.{i}:{port}'
        try:
            r = requests.get(f'{url}/ping', timeout=timeout)
            if r.status_code == 200:
                return url
        except requests.RequestException:
            pass
        return None

    with ThreadPoolExecutor(max_workers=workers) as executor:
        results = executor.map(check_host, range(1, 255))
    return [u for u in results if u]
```

2. Backend API Client Abstraction

File: `api_client.py`

```
import requests
from typing import List

class APIClient:
    """Thin wrapper over the device's FastAPI backend API."""
    def __init__(self, base_url: str):
        self.base = base_url.rstrip("/")
        self.session = requests.Session()

    def _url(self, path: str) -> str:
        return f"{self.base}/{path.lstrip('/')}"

    def _req(self, method: str, path: str, timeout: float = 3, **kwargs):
        resp = self.session.request(method, self._url(path), timeout=timeout,
**kwargs)
        resp.raise_for_status()
        return resp.json() if resp.content else {}

    def list_ports(self) -> List[str]:
        return self._req("GET", "/ports")

    def status(self) -> dict:
        return self._req("GET", "/status")

    def connect(self, port: str):
        return self._req("POST", "/connect", json={"port": port})

    def move_rel(self, steps: int):
        return self._req("POST", "/move_rel", json={"steps": steps},
```

```
timeout=150)

    def move_abs(self, position: int):
        return self._req("POST", "/move_abs", json={"position": position},
timeout=150)

    def home(self):
        return self._req("POST", "/home", timeout=120)

    def flash(self):
        return self._req("POST", "/identify")

    def stop(self):
        return self._req("POST", "/stop")
```

3. Persistent Settings and Session Restore

File: `storage.py`

```
"""Persistent session state storage for device (settings and positions)."""

from pathlib import Path
import json

# where we persist state (settings + positions)
STORAGE_PATH = Path.home() / ".device_state.json"

def load_state() -> dict:
    """
    Load complete state from disk, returning a dict with "settings" and
"positions".
    """
    try:
        return json.loads(STORAGE_PATH.read_text())
    except FileNotFoundError:
        return {"settings": {}, "positions": {}}

def save_state(state: dict) -> None:
    """
    Atomically write the full state (settings + positions) back to disk.
    """
    STORAGE_PATH.parent.mkdir(parents=True, exist_ok=True)
    STORAGE_PATH.write_text(json.dumps(state, indent=2))

def load_positions() -> dict:
    """
```

```
    Return the positions sub-dict from saved state.
    """
    return load_state().get("positions", {})

def save_positions(positions: dict) -> None:
    """
    Save only the positions into the state file, preserving existing settings.
    """
    state = load_state()
    state["positions"] = positions
    save_state(state)

def load_settings() -> dict:
    """
    Return the settings sub-dict from saved state.
    """
    return load_state().get("settings", {})

def save_settings(settings: dict) -> None:
    """
    Save only the settings into the state file, preserving existing positions.
    """
    state = load_state()
    state["settings"] = settings
    save_state(state)
```

File: `constants.py`

```
"""Constants for device presets and unit conversion."""

STEP_PRESETS = {
    "Preset 1": 1000,
    "Preset 2": 5000,
    "Manual set…": None,
}

UNIT_FACT = {
    "mm": 1.0,
    "µm": 1e-3,
}
```

4. Modular UI Layout and Controls

File: `main_window.py`

```
#!/usr/bin/env python3
"""Main Window Module

This module defines the MainWindow class, providing a PyQt6 GUI for controlling
a generic lab device via a networked backend.
It supports session restore on startup (restoring the last used backend, port,
preset, and position) and includes safety prompts if the device lost power or
moved since last use.
"""

import sys
import datetime
import socket

from PyQt6.QtGui import QIcon, QIntValidator, QDoubleValidator
from PyQt6.QtWidgets import (
    QApplication, QMainWindow, QWidget, QLabel, QComboBox, QLineEdit,
    QPushButton, QHBoxLayout, QVBoxLayout, QGridLayout, QFormLayout,
    QGroupBox, QStatusBar, QMessageBox
)
from PyQt6.QtCore import QTimer, QThread

from api_client import APIClient
from discovery import scan_for_backends
from constants import STEP_PRESETS, UNIT_FACT
from storage import load_positions, save_positions, load_settings, save_settings
from popups import ask_restore_session, ask_restore_preset, warn_lost_power,
warn_moved
from worker import Worker

class MainWindow(QMainWindow):
    """Main UI class for the device controller GUI (with session restore and
safety checks)."""

    def __init__(self, backend_hint=None):
        super().__init__()
        # Window setup
        self.setWindowTitle("Device Controller")
        self.setWindowIcon(QIcon.fromTheme("applications-engineering"))
        self.resize(900, 600)

        # State
        self.api = None
        self.steps_per_mm = STEP_PRESETS["Preset 1"]
        self.positions = load_positions()      # { "backend|port": {pos, time,
steps_per_mm, homed} }
        self.settings = load_settings()        # { backend, port, preset,
```

```
steps_per_mm, date }
        self.session_restored = False
        self._did_post_connect_warn = False

        # Input validators
        self.int_val = QIntValidator(1, 10**6, self)
        self.fl_val  = QDoubleValidator(0.0, 1e6, 6, self)

        # Build UI, discover backends, restore session, start polling
        self._build_ui()
        self._discover_backends(backend_hint)
        self._status_timer = QTimer(self)
        self._status_timer.timeout.connect(self._refresh_status)
        self._status_timer.start(500)
        self._maybe_restore_session()

    def _maybe_restore_session(self):
        """
        On startup, if we have saved backend+port, offer to restore:
          1) backend URL
          2) cube-port
          3) steps/mm preset (or Custom)
          4) return to last absolute position
        """
        saved_backend = self.settings.get("backend")
        saved_port    = self.settings.get("port")
        saved_preset  = self.settings.get("preset", "Custom")
        saved_spm     = self.settings.get("steps_per_mm", self.steps_per_mm)
        saved_date    = self.settings.get("date", "")

        if not (saved_backend and saved_port):
            return

        if self.cmb_backend.findText(saved_backend) < 0:
            return

        if not ask_restore_session(
            self,
            saved_backend,
            saved_port,
            saved_preset,
            saved_spm,
            saved_date,
        ):
            return

        self.session_restored = True
```

```
        # 1) select saved backend & load ports
        self.cmb_backend.setCurrentText(saved_backend)
        self._on_backend_change(saved_backend)

        # 2) select saved port
        self.cmb_port.setCurrentText(saved_port)

        # 3) restore preset or custom steps/mm
        if saved_preset in STEP_PRESETS:
            self.cmb_preset.setCurrentText(saved_preset)
        else:
            self.cmb_preset.setCurrentText("Custom")
            self.ed_steps.setText(str(saved_spm))
        self._on_preset(self.cmb_preset.currentText())

        # 4) connect to the device (also saves settings)
        self._connect_device()

        # 5) immediate status refresh to avoid flicker
        self._refresh_status()

        # 6) move back to last position
        key = f"{self.api.base}|{saved_port}"
        last = self.positions.get(key)
        if last and "pos" in last:
            QTimer.singleShot(200, lambda: self._run_async(self.api.move_abs,
last["pos"]))

        # 7) schedule lost-power/moved-elsewhere check
        self._did_post_connect_warn = False
        QTimer.singleShot(500, self._check_post_connect)

    def _connect_device(self):
        """
        Called when user selects a cube-port:
        - Connects to API
        - Saves session settings
        - Schedules lost-power/moved warnings once device is idle
        """
        port = self.cmb_port.currentText().strip()
        if not port:
            return

        try:
            self.api.connect(port)
            self.statusbar.showMessage("Connected", 2000)
        except Exception as e:
            QMessageBox.critical(self, "Error", f"/connect failed:\n{e}")
```

```
            return

        self.settings.update({
            "backend":      self.api.base,
            "port":         port,
            "preset":       self.cmb_preset.currentText(),
            "steps_per_mm": self.steps_per_mm,
            "date":         datetime.datetime.now().isoformat()
        })
        save_settings(self.settings)

        self._did_post_connect_warn = False
        QTimer.singleShot(500, self._check_post_connect)

    def _check_post_connect(self):
        """
        After connecting:
        1) If busy (initializing/moving), retry in 200 ms
        2) Once idle and not yet warned:
           • If previously homed but now un-homed → lost-power
           • Else if homed and position differs → moved-elsewhere
        """
        port = self.cmb_port.currentText().strip()
        key  = f"{self.api.base}|{port}"
        last = self.positions.get(key)
        if not last or self._did_post_connect_warn:
            return

        st = self.api.status()
        curr_pos   = st["position"]
        curr_homed = st["homed"]
        busy = st["moving_forward"] or st["moving_reverse"]

        if busy:
            QTimer.singleShot(200, self._check_post_connect)
            return

        last_pos   = last["pos"]
        last_homed = last.get("homed", False)
        last_steps = last.get("steps_per_mm", self.steps_per_mm)
        last_mm    = last_pos / last_steps
        last_time  = last["time"]

        if last_homed and not curr_homed:
            self._did_post_connect_warn = True
            if warn_lost_power(self, last_pos, last_mm, last_time):
                def after_home(res, err):
```

```
                    if not err:
                        self._run_async(self.api.move_abs, last_pos)
                w = Worker(self.api.home)
                t = QThread(self)
                w.moveToThread(t)
                t.started.connect(w.run)
                w.finished.connect(after_home)
                t.start()

        if curr_homed and abs(curr_pos - last_pos) > 2:
            self._did_post_connect_warn = True
            curr_mm = curr_pos / self.steps_per_mm
            if warn_moved(self, last_pos, curr_pos, last_mm, curr_mm,
last_time):
                self._run_async(self.api.move_abs, last_pos)

    def _build_ui(self):
        """Construct all widgets, layouts, and connect signals."""
        central = QWidget(self)
        self.setCentralWidget(central)
        main_v = QVBoxLayout(central)
        main_v.setSpacing(10)
        main_v.setContentsMargins(10, 10, 10, 10)

        # Status row
        st_h = QHBoxLayout()
        self.lbl_status = QLabel("Status: —")
        self.lbl_homed   = QLabel("Homed: ✗")
        self.lbl_pos     = QLabel("Pos: — cnt | — mm")
        for lbl in (self.lbl_status, self.lbl_homed, self.lbl_pos):
            st_h.addWidget(lbl)
        st_h.addStretch()
        main_v.addLayout(st_h)

        # Network group
        net_g = QGroupBox("Network")
        net_f = QFormLayout(net_g)
        self.cmb_backend = QComboBox()
        self.cmb_backend.setEditable(True)
        self.cmb_backend.setPlaceholderText("http://<ip>:8000")
        self.cmb_backend.currentTextChanged.connect(self._on_backend_change)
        btn_add = QPushButton("Add Backend")
        btn_add.clicked.connect(self._add_backend)
        self.cmb_port = QComboBox()
        self.cmb_port.currentIndexChanged.connect(self._connect_device)
        net_f.addRow("Backend:", self.cmb_backend)
        net_f.addRow("", btn_add)
```

```
        net_f.addRow("Device port:", self.cmb_port)
        main_v.addWidget(net_g)

        # Motion group
        mot_g = QGroupBox("Motion")
        grid = QGridLayout(mot_g)
        grid.setHorizontalSpacing(8)
        grid.setVerticalSpacing(4)

        # Steps/mm preset
        grid.addWidget(QLabel("Steps/mm preset:"), 0, 0)
        self.cmb_preset = QComboBox()
        self.cmb_preset.addItems(STEP_PRESETS.keys())
        self.cmb_preset.currentTextChanged.connect(self._on_preset)
        grid.addWidget(self.cmb_preset, 0, 1)
        self.ed_steps = QLineEdit(str(self.steps_per_mm))
        self.ed_steps.setValidator(self.int_val)
        grid.addWidget(self.ed_steps, 0, 2)

        # Relative move
        grid.addWidget(QLabel("Move relative:"), 1, 0)
        self.ed_rel = QLineEdit("0"); self.ed_rel.setValidator(self.fl_val)
        grid.addWidget(self.ed_rel, 1, 1)
        self.cmb_unit_rel = QComboBox();
self.cmb_unit_rel.addItems(UNIT_FACT.keys())
        grid.addWidget(self.cmb_unit_rel, 1, 2)
        btn_neg = QPushButton("–"); btn_neg.clicked.connect(lambda:
self._move_rel(-1))
        btn_pos = QPushButton("+"); btn_pos.clicked.connect(lambda:
self._move_rel(1))
        grid.addWidget(btn_neg, 1, 3); grid.addWidget(btn_pos, 1, 4)

        # Absolute move
        grid.addWidget(QLabel("Move absolute:"), 2, 0)
        self.ed_abs = QLineEdit("0"); self.ed_abs.setValidator(self.fl_val)
        grid.addWidget(self.ed_abs, 2, 1)
        self.cmb_unit_abs = QComboBox();
self.cmb_unit_abs.addItems(UNIT_FACT.keys())
        grid.addWidget(self.cmb_unit_abs, 2, 2)
        btn_go = QPushButton("Go to absolute");
btn_go.clicked.connect(self._move_abs)
        grid.addWidget(btn_go, 2, 3, 1, 2)

        # Control buttons
        btn_home = QPushButton("Home");  btn_home.clicked.connect(lambda:
self._run_async(self.api.home))
        btn_flash = QPushButton("Flash"); btn_flash.clicked.connect(lambda:
self._run_async(self.api.flash))
```

```
        btn_stop = QPushButton("STOP")
        btn_stop.setStyleSheet("background:#d9534f;color:white;font-
weight:bold;")
        btn_stop.clicked.connect(lambda: self._run_async(self.api.stop))
        grid.addWidget(btn_home, 3, 0); grid.addWidget(btn_flash, 3, 1)
        grid.addWidget(btn_stop, 3, 2, 1, 3)

        main_v.addWidget(mot_g)
        main_v.addStretch()

        # Status bar
        self.statusbar = QStatusBar(self)
        self.setStatusBar(self.statusbar)

    def _discover_backends(self, hint=None):
        """Populate backend dropdown: hint, localhost, docker alias, LAN
scan."""
        urls = []
        if hint and hint != "auto":
            urls.append(hint)
        else:
            urls += ["http://127.0.0.1:8000", "http://host.docker.internal:
8000"]
            try:
                ip = socket.gethostbyname(socket.gethostname())
                urls.append(f"http://{ip}:8000")
            except:
                pass
            urls += scan_for_backends()
        for u in dict.fromkeys(urls):
            self.cmb_backend.addItem(u)
        if self.cmb_backend.count():
            self._on_backend_change(self.cmb_backend.currentText())

    def _add_backend(self):
        """Save user-typed backend URL into dropdown."""
        url = self.cmb_backend.currentText().strip()
        if url and self.cmb_backend.findText(url) == -1:
            self.cmb_backend.addItem(url)
            self.cmb_backend.setCurrentText(url)

    def _on_backend_change(self, url):
        """When backend changes, fetch available cube-ports."""
        if not url:
            return
        self.api = APIClient(url)
        self.statusbar.showMessage("Loading ports...", 2000)
        try:
```

```
            ports = self.api.list_ports()
        except Exception as e:
            QMessageBox.critical(self, "Error", f"/ports failed:\n{e}")
            ports = []
        self.cmb_port.clear()
        self.cmb_port.addItems(ports)
        self.statusbar.showMessage(f"Ports: {ports}", 2000)

    def _on_preset(self, name):
        """Handle preset change: update steps_per_mm and save."""
        val = STEP_PRESETS.get(name)
        if val is None:
            self.ed_steps.setReadOnly(False)
        else:
            self.steps_per_mm = val
            self.ed_steps.setText(str(val))
            self.ed_steps.setReadOnly(True)
        self.settings["preset"]       = name
        self.settings["steps_per_mm"] = self.steps_per_mm
        save_settings(self.settings)

    def _to_counts(self, txt, cmb):
        """Convert text+unit into encoder counts."""
        try:
            dist = float(txt) * UNIT_FACT[cmb.currentText()]
        except:
            dist = 0.0
        try:
            self.steps_per_mm = int(self.ed_steps.text())
        except:
            pass
        return round(dist * self.steps_per_mm)

    def _move_rel(self, sign):
        """Move relative in background thread."""
        cnt = sign * abs(self._to_counts(self.ed_rel.text(), self.cmb_unit_rel))
        self.statusbar.showMessage("Moving relative...", 2000)
        self._run_async(self.api.move_rel, cnt)

    def _move_abs(self):
        """Move absolute in background thread."""
        cnt = self._to_counts(self.ed_abs.text(), self.cmb_unit_abs)
        self.statusbar.showMessage("Moving absolute...", 2000)
        self._run_async(self.api.move_abs, cnt)

    def _refresh_status(self):
        """Poll backend, update labels, and persist if idle & homed."""
        if not self.api:
```

```
            self.lbl_status.setText("Status: no backend")
            self.lbl_homed.setText("Homed: ✗")
            return
        try:
            st = self.api.status()
        except Exception as e:
            self.lbl_status.setText(f"Status: ⚠{e}")
            return
        busy = st["moving_forward"] or st["moving_reverse"]
        homed_flag = st["homed"]
        pos = st["position"]
        self.lbl_status.setText("Status: moving" if busy else "Status: idle")
        self.lbl_homed.setText(f"Homed: {'✓' if homed_flag else '✗'}")
        mm = pos / self.steps_per_mm
        self.lbl_pos.setText(f"Pos: {pos} cnt | {mm:.3f} mm")
        if homed_flag and not busy:
            port = self.cmb_port.currentText().strip()
            key  = f"{self.api.base}|{port}"
            self.positions[key] = {
                "pos": pos,
                "time": datetime.datetime.now().isoformat(),
                "steps_per_mm": self.steps_per_mm,
                "homed": True
            }
            save_positions(self.positions)

    def _run_async(self, fn, *args):
        """Run a backend call in a Worker/QThread to keep UI responsive."""
        if not callable(fn):
            return
        worker = Worker(fn, *args)
        thread = QThread(self)
        worker.moveToThread(thread)
        thread.started.connect(worker.run)
        worker.finished.connect(lambda res, err: self._on_done(thread, worker,
res, err))
        thread.start()

    def _on_done(self, thread, worker, res, err):
        """Cleanup after task and report errors or 'Done'."""
        thread.quit()
        thread.wait()
        worker.deleteLater()
        thread.deleteLater()
        if err:
            QMessageBox.critical(self, "Error", str(err))
        else:
```

```
            self.statusbar.showMessage("Done", 2000)
            self._refresh_status()
```

File: `popups.py`

```
"""Popup dialog helpers for user prompts and warnings in device UI."""

import datetime
from PyQt6.QtWidgets import QMessageBox

def _format_date(iso_dt: str) -> str:
    """
    Convert an ISO-8601 timestamp into US-style MM/DD/YYYY hh:mm:ss AM/PM.
    Falls back to the raw string on parse errors.
    """
    try:
        dt = datetime.datetime.fromisoformat(iso_dt)
        return dt.strftime("%m/%d/%Y %I:%M:%S %p")
    except Exception:
        return iso_dt

def ask_restore_session(
    parent,
    backend: str,
    port: str,
    preset_name: str,
    steps_per_mm: int,
    iso_dt: str
) -> bool:
    """
    On startup, ask whether to restore the last session.
    Shows backend, port, named preset, numeric steps/mm, and US-formatted
timestamp.
    """
    ts = _format_date(iso_dt)
    msg = (
        "Would you like to restore your last session?\n\n"
        f"Backend: {backend}\n"
        f"Port: {port}\n"
        f"Preset: {preset_name}\n"
        f"Steps/mm: {steps_per_mm}\n"
        f"Last used: {ts}"
    )
    choice = QMessageBox.question(
        parent,
        "Restore session",
```

```
        msg,
        QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No
    )
    return (choice == QMessageBox.StandardButton.Yes)

def ask_restore_preset(
    parent,
    old_steps: int,
    iso_dt: str
) -> bool:
    """
    When manually selecting a previously-used backend+port,
    ask if you'd like to restore the old steps/mm setting.
    """
    ts = _format_date(iso_dt)
    msg = (
        f"You previously used Steps/mm = {old_steps} on {ts} for this device.
\n\n"
        "Would you like to restore that preset now?"
    )
    choice = QMessageBox.question(
        parent,
        "Restore preset",
        msg,
        QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No
    )
    return (choice == QMessageBox.StandardButton.Yes)

def warn_lost_power(
    parent,
    last_pos: int,
    last_mm: float,
    iso_dt: str
) -> bool:
    """
    Warn that the device was powered off (homed=False).
    Offer to home and return to the last saved position.
    """
    ts = _format_date(iso_dt)
    msg = (
        "It appears the device was powered off since last use.\n"
        f"You were at {last_pos} cnt ({last_mm:.3f} mm) on {ts}.\n\n"
        "You must home before absolute moves.  Home and return to that
position?"
    )
    choice = QMessageBox.question(
        parent,
        "Device lost power",
```

```
        msg,
        QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No
    )
    return (choice == QMessageBox.StandardButton.Yes)

def warn_moved(
    parent,
    last_pos: int,
    curr_pos: int,
    last_mm: float,
    curr_mm: float,
    iso_dt: str
) -> bool:
    """
    Warn that the device has been moved elsewhere since last use.
    Offer to return to your last saved position.
    """
    ts = _format_date(iso_dt)
    msg = (
        f"Device has moved since {ts}:\n"
        f"  Last: {last_pos} cnt ({last_mm:.3f} mm)\n"
        f"  Now:  {curr_pos} cnt ({curr_mm:.3f} mm)\n\n"
        "Would you like to return it to your last saved position?"
    )
    choice = QMessageBox.question(
        parent,
        "Device moved",
        msg,
        QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No
    )
    return (choice == QMessageBox.StandardButton.Yes)
```

5. Worker Thread System for Async API Calls

File: `worker.py`

```
"""Background worker thread for asynchronous device API calls (using
QThread)."""

from PyQt6.QtCore import QObject, pyqtSignal

class Worker(QObject):
    """Runs any function in a QThread and emits (result, error)."""
    finished = pyqtSignal(object, object)

    def __init__(self, fn, *args):
```

```
        super().__init__()
        self.fn = fn
        self.args = args

    def run(self):
        try:
            res = self.fn(*self.args)
            err = None
        except Exception as e:
            res, err = None, e
        self.finished.emit(res, err)
```

6. Boot-Time Automation (Restore Position, Device Homing, etc.)

The generic UI implements the same startup restoration logic. The **MainWindow** class will attempt to restore the last session on launch, and uses the same `_maybe_restore_session` and `_check_post_connect` approach as the TDC001 version. You can tailor these safety checks to your device – for example, if the device doesn't require homing, the `warn_lost_power` dialog and homing sequence can be removed or adjusted. By default, the boilerplate assumes a homing mechanism and absolute position reference (common for motion stages).

7. Startup Entrypoint and Docker Configuration

File: `run.py`

```
"""Startup entry point for the device GUI application (supports noVNC in
Docker)."""

import os, sys, argparse, shutil, subprocess
from PyQt6.QtWidgets import QApplication
from main_window import MainWindow

def main():
    p = argparse.ArgumentParser(description="Device Controller UI launcher")
    p.add_argument("--backend", default=os.getenv("BACKEND_URL", "auto"))
    args = p.parse_args()

    # optional Xvfb+noVNC support for headless Docker
    if os.getenv("USE_NOVNC"):
        if shutil.which("Xvfb") and shutil.which("websockify"):
            subprocess.Popen(["Xvfb", ":99", "-screen", "0", "1280x800x24"])
            subprocess.Popen(["websockify", "6080", "localhost:5900"])

    app = QApplication(sys.argv)
    w   = MainWindow(args.backend)
    w.show()
```

```
    sys.exit(app.exec())

if __name__ == "__main__":
    main()
```

File: `Dockerfile.gui`

```
FROM python:3.10-slim

# Environment settings for X (disable MIT-SHM) and optional backend URL
ENV QT_X11_NO_MITSHM=1

# Install X11 server, VNC, websockify, and minimal window manager
RUN apt-get update && apt-get install -y \
    xvfb x11vnc websockify novnc fluxbox xauth libgl1-mesa-glx \
    && rm -rf /var/lib/apt/lists/*

# Install Python dependencies for GUI
RUN pip install PyQt6 requests

# Copy application code
COPY . /app
WORKDIR /app

# Launch the GUI (noVNC will be enabled via USE_NOVNC env variable)
CMD ["python", "run.py"]
```

Docker Compose Snippet

```
services:
  device_ui:
    build: ./device_ui
    environment:
      - USE_NOVNC=1
      - BACKEND_URL=http://device_api:8000
    ports:
      - "6080:6080"
    depends_on:
      - device_api
```
