Dobot Magician: Entwicklung einer Befehlskette

Schrittweise Dokumentation des Chats - von der ersten Idee bis zur funktionierenden Pause

Ziel des Gesprächs

Aus einer leicht lesbaren Liste sollten mehrere Dobot-Befehle automatisch in die Queue eingetragen werden. Zu jedem Bewegungsbefehl sollten Befehlsnummer, Queue-Index und Beschreibung verwaltet werden. Später kam die Anforderung hinzu, nach jedem Befehl eine frei einstellbare Pause einzufügen.

Ausgangspunkt

```
befehle = {}

pos = [180, 160, 50, 0]
i = dType.SetPTPCmd(
    api,
    dType.PTPMode.PTPMOVLXYZMode,
    pos[0], pos[1], pos[2], pos[3],
    isQueued=1,
)[0]

befehle[i] = "Fahre zu Punkt 1"
```

Diese Lösung funktionierte, hatte aber einen Nachteil: Jeder Queue-Befehl und jeder Beschreibungstext mussten einzeln programmiert und dem zurückgegebenen Index manuell zugeordnet werden.

1. Idee einer eigenen Befehlsliste

Die gewünschte Liste sollte wie eine kleine, verständliche Dobot-Sprache aussehen. Jeder Eintrag enthält Befehlsname, Koordinaten und Beschreibung.

```
befehle = [
    ("fahre_zu",   200, 125,  20, 79, "Fahre zum Startpunkt"),
    ("fahre_um",    20,  14,   0,  0, "Kleine Verschiebung"),
    ("springe_auf", 200, 140, -40,  0, "Sprung über den Kasten"),
]
```

2. Automatische Erzeugung der Queue

Die Funktion befehlskette_erstellen() liest die Liste nacheinander. Sie übersetzt den verständlichen Namen in einen PTP-Modus, ruft SetPTPCmd() auf und speichert den zurückgegebenen Queue-Index zusammen mit Befehlsnummer, Position und Text.

```
PTP_BEFEHLE = {
    "fahre_zu": dType.PTPMode.PTPMOVLXYZMode,
    "fahre_um": dType.PTPMode.PTPMOVLXYZINCMode,
    "springe_auf": dType.PTPMode.PTPJUMPXYZMode,
}
```

3. Automatisch erzeugte Verwaltungsstruktur

Die Queue-Indizes werden nicht mehr vom Anwender eingetragen. Sie entstehen durch die Dobot-API und dienen als Schlüssel eines Dictionaries.

```
queue_befehle[bewegungsindex] = {
    "nummer": befehlsnummer,
    "befehl": befehlsname,
    "position": (x, y, z, r),
    "text": str(text),
}
```

4. Anpassung an die vorgegebene Ordnerstruktur

Die ersten Dateien konnten DobotDllType.py nicht zuverlässig finden. Das Projekt verwendet eine feste Struktur mit dem SDK im übergeordneten Ordner.

Ordner/Datei	Bedeutung
Dobot-Python/dobot.py	Eigene verständliche Hilfsbibliothek
Dobot-Python/sdk64/DobotDllType.py	Python-Wrapper der Dobot-API
Dobot-Python/sdk64/DobotDll.dll	64-Bit-DLL
Dobot-Python/befehlskette-1/befehlskette.py	Modul für Befehlsketten
Dobot-Python/befehlskette-1/befehlskette_beispiel.py	Test- und Beispielprogramm

```
PROJEKTORDNER = Path(__file__).resolve().parent
HAUPTORDNER = PROJEKTORDNER.parent
SDK_ORDNER = HAUPTORDNER / "sdk64"

if str(HAUPTORDNER) not in sys.path:
    sys.path.insert(0, str(HAUPTORDNER))

if os.name == "nt":
    os.add_dll_directory(str(SDK_ORDNER))

from sdk64 import DobotDllType as dType
```

Damit wurde DobotDllType.py aus dem richtigen Projektordner importiert und der Windows-DLL-Suchpfad um den Ordner sdk64 erweitert.

5. Fehler: DobotDll.dll wurde nicht gefunden

Beim Aufruf von dType.load() erschien zunächst FileNotFoundError. Die SDK-Datei wurde zwar importiert, aber CDLL('DobotDll.dll') suchte die DLL nicht zuverlässig im Verzeichnis sdk64.

```
FileNotFoundError:
Could not find module 'DobotDll.dll'
(or one of its dependencies).
```

Zusätzlich zeigte die Ausgabe zeitweise zwei verschiedene Projektkopien: D:\Github\... und D:\Daten\Github\.... Dadurch bestand die Gefahr, dass eine Datei bearbeitet, aber eine andere importiert wurde.

6. Kleine Folgefehler im Beispielprogramm

Nach dem erfolgreichen Laden der DLL traten mehrere NameError-Meldungen auf. Sie waren keine Dobot-Fehler, sondern fehlende Konstanten oder Importe.

Fehlermeldung	Ursache	Korrektur
PORT is not defined	COM-Port fehlte	PORT = "COM10"
VERSION is not defined	Versionskonstante nicht importiert	VERSION aus befehlskette importieren
STANDARD_PAUSE_MS is not defined	Standardpause fehlte	STANDARD_PAUSE_MS = 1000

7. Erweiterung um Pausen

Ein Befehl darf sechs oder sieben Einträge besitzen. Ohne siebten Eintrag wird die Standardpause verwendet. Mit einem siebten Eintrag wird die Pause für diesen Befehl überschrieben.

```
STANDARD_PAUSE_MS = 1000

befehle = [
    # Standardpause: 1000 ms
    ("fahre_zu", 180, 160, 50, 0, "Fahre zu Punkt 1"),

    # Individuelle Pause: 2500 ms
    ("fahre_zu", 240, 140, 70, 0, "Fahre zu Punkt 2", 2500),

    # Keine Pause
    ("fahre_zu", 200, 180, 50, 0, "Fahre zu Punkt 3", 0),
]
```

Wichtig war, die Pause nicht mit time.sleep() im Python-Programm zu erzeugen. Die Queue würde während einer Python-Pause weiterlaufen. Stattdessen musste ein echter WAIT-Befehl in die Dobot-Queue eingetragen werden.

8. Fehlerhafte Alarmprüfung

Beim ersten Test begann der Arm sich zu bewegen und stoppte sofort. Gleichzeitig meldete Python einen Alarm, obwohl am Dobot-Fuß keine Alarmanzeige erschien.

```
alarmdaten = dType.GetAlarmsState(api)

if alarmdaten and any(alarmdaten):
    dType.SetQueuedCmdStopExec(api)
    raise RuntimeError(...)
```

Die Rückgabe von GetAlarmsState() war verschachtelt. any(alarmdaten) prüfte die äußere Struktur und konnte deshalb einen Alarm melden, obwohl die eigentlichen Alarmbytes alle null waren. Außerdem stand die Alarmabfrage zunächst innerhalb der Schleife über alle Bewegungsindizes und wurde unnötig oft ausgeführt.

```
alarmdaten = dType.GetAlarmsState(api)
alarmbytes = alarmdaten[0] if alarmdaten else []

if any(alarmbytes):
    ...
```

9. Fehler in der Ausführungsschleife

Nach der Korrektur der Alarmprüfung trat ein weiterer Programmfehler auf. aktueller_index und bewegungsindex wurden verwendet, ohne an der betreffenden Stelle gesetzt zu sein.

```
while True:
    aktueller_index = dType.GetQueuedCmdCurrentIndex(api)[0]

    for bewegungsindex in sorted(queue_befehle):
        if (
            bewegungsindex <= aktueller_index
            and bewegungsindex not in angezeigte_befehle
        ):
            ...
```

Diese Fassung liest in jedem Schleifendurchlauf zuerst den aktuellen Queue-Index und prüft danach alle gespeicherten Bewegungsindizes.

10. Diagnose über die Queue-Indizes

Zur Fehlersuche wurde jede Änderung des aktuellen Queue-Index ausgegeben. Dadurch war zu sehen, dass die erste Bewegung korrekt erreicht und ausgeführt wurde.

```
Aktueller Queue-Index: 26
Aktueller Queue-Index: 27
Befehl 1: Fahre zu Punkt 1 (Queue-Index 27)
```

Die vollständige Queue war dabei folgendermaßen aufgebaut:

Index	Befehl
27	Bewegung 1
28	Pause nach Bewegung 1
29	Bewegung 2
30	Pause nach Bewegung 2
31	Bewegung 3

11. Entscheidende Ursache: falsche Zeiteinheit

Der Dobot blieb nach der ersten Bewegung scheinbar hängen. Tatsächlich arbeitete er den WAIT-Befehl ab. Der Python-Wrapper SetWAITCmd() erwartet in der verwendeten DobotDllType.py die Wartezeit in Sekunden und rechnet intern selbst in Millisekunden um.

```
# Falsch:
warteindex = dType.SetWAITCmd(
    api,
    pause_ms,
    isQueued=1,
)[0]
```

Eingabewert	Tatsächliche Pause
1000	1000 Sekunden = 16 Minuten 40 Sekunden
2500	2500 Sekunden = 41 Minuten 40 Sekunden

Der Stillstand war daher kein Hängenbleiben, sondern eine extrem lange, korrekt abgearbeitete Pause.

12. Endgültige Korrektur

Die eigene Schnittstelle arbeitet weiterhin mit gut verständlichen Millisekunden. Nur beim Aufruf der Dobot-API wird der Wert in Sekunden umgerechnet.

```
warteindex = dType.SetWAITCmd(
    api,
    pause_ms / 1000.0,
    isQueued=1,
)[0]
```

Damit bedeuten 1000 weiterhin eine Sekunde und 2500 weiterhin 2,5 Sekunden.

13. Empfohlener Aufbau des Beispielprogramms

Alle Werte, die beim Testen häufig verändert werden, stehen gemeinsam am Anfang. Der technische Teil folgt darunter und soll normalerweise nicht geändert werden.

```
# ------------------------------------------------------------
# Diese Zeilen anpassen
# ------------------------------------------------------------

PORT = "COM10"
BAUDRATE = 115200
STANDARD_PAUSE_MS = 1000

befehle = [
    ("fahre_zu", 180, 160, 50, 0, "Fahre zu Punkt 1"),
    ("fahre_zu", 240, 140, 70, 0, "Fahre zu Punkt 2", 2500),
    ("fahre_zu", 200, 180, 50, 0, "Fahre zu Punkt 3", 0),
]

# ------------------------------------------------------------
# Ab hier nichts mehr ändern
# ------------------------------------------------------------
```

14. Ergebnis

Erreicht	Beschreibung
Lesbare Befehlsliste	Befehle werden wie eine kleine Dobot-Sprache notiert.
Automatische Queue-Verwaltung	Queue-Indizes werden von der API übernommen.
Ausgabe in Thonny	Befehlsnummer, Text und Queue-Index werden angezeigt.
Standardpause	Eine gemeinsame Pause gilt für alle Befehle ohne Sonderwert.
Individuelle Pause	Jeder Befehl kann die Standardpause überschreiben.
Keine Pause	Der Wert 0 unterdrückt den WAIT-Befehl.
Ordnerstruktur	sdk64 und DLL werden aus dem richtigen Projektordner geladen.

15. Zentrale Erkenntnisse

1. Queue-Indizes sollten automatisch verwaltet werden.
2. Eine Pause zwischen Queue-Befehlen gehört als WAIT-Befehl in die Queue.
3. Rückgabestrukturen der Dobot-API müssen genau geprüft werden.
4. Mehrere gleichnamige Projektkopien führen leicht zu falschen Importen.
5. Die Einheit eines API-Parameters muss aus der konkreten Wrapper-Implementierung abgelesen

werden.

6. Diagnoseausgaben der Queue-Indizes sind bei scheinbarem Stillstand besonders hilfreich.

Endstand des Moduls in diesem Chat: **befehlskette.py Version 2.2**