Dobot Magician

Befehlskette Version 3.3.5

Ausführliche Beschreibung des aktuellen Projektzustands

Hauptmodul	befehlskette_v3_3_5.py - Version 3.3.5
Beispielprogramm	befehlskette_beispiel_v3_3_5.py - Programmversion 3.3.5
Kommunikation	esp32_kommunikation_v1_4.py - Version 1.4
ESP32-Programm	esp32_dobot_steuerung_v1_3.py - MicroPython-Version 1.3
Simulator	esp-simulator.py / esp-simulator_v1_3.py - Version 1.3
Dokumentationsstand	25.07.2026

Kurzfassung: Die Version 3.3.5 verbindet eine vorab geprüfte, steuerbare Dobot-Befehlskette mit einem ESP32 oder einem TCP-Simulator. Neben Roboterbewegungen, Sauger, HOME, Marken, Sprüngen und Bedingungen kann die Befehlskette jetzt mit `esp_senden` **frei wählbare Textbefehle an den ESP32 übertragen.**

Diese Dokumentation beschreibt den gegenwärtigen Entwicklungsstand. Die Python-Dateien bleiben die verbindliche technische Quelle. Praktische Ergebnisse aus den laufenden Tests sind nachzutragen, sobald sie bestätigt sind.

Inhaltsverzeichnis

1. Projektziel und aktueller Reifegrad

Das Projekt soll Dobot-Abläufe in einer für Unterricht und Projekte verständlichen Python-Schreibweise beschreiben. Die eigentliche Robotersteuerung wird durch eine Befehlsliste aus einzeiligen Tupeln formuliert. Ein separates Modul prüft diese Liste, zeigt sie verständlich an und führt sie steuerbar aus.

- Der Dobot kann Bewegungs-, HOME- und Saugerbefehle ausführen.
- Abläufe können mit Marken, Sprüngen und Bedingungen strukturiert werden.
- PAUSE, WEITER, HALT und STATUS können von Tastatur oder ESP32 kommen.
- Einmalige ESP32-Meldungen und dauerhaft gespeicherte Werte werden getrennt behandelt.
- Der ESP32 oder Simulator erhält laufend Informationen über den Programmzustand.
- Version 3.3.5 ergänzt mit esp_senden die aktive Befehlsrichtung vom PC zum ESP32.

Reifegrad: Die Softwarestruktur und Syntaxprüfungen sind weit entwickelt. Die neuen Dateien für ESP32 1.3 und Simulator 1.3 wurden logisch und syntaktisch vorbereitet. Der praktische Systemtest mit realem Dobot, echtem ESP32 und Simulator läuft derzeit und ist noch nicht vollständig abgeschlossen.

1.1 Abgrenzung

Die Befehlskette ist kein Ersatz für Sicherheitsplanung, Arbeitsraumprüfung oder eine mechanische Kollisionsbetrachtung. Eine syntaktisch korrekte Koordinate kann für den konkreten Aufbau trotzdem unzulässig oder gefährlich sein.

2. Versions- und Dateistand

Datei	Version	Aufgabe
`befehlskette_v3_3_5.py`	3.3.5	Prüfung, Darstellung und steuerbare Ausführung der Befehlsliste.
`befehlskette_beispiel_v3_3_5.py`	3.3.5	Konfiguration, Startmenü, Kommunikation, Dobot-Verbindung und Beispielablauf.
`esp32_kommunikation_v1_4.py`	1.4	Gemeinsame COM-/TCP-Kommunikation, Queues und permanenter Wertespeicher.
`esp32_dobot_steuerung_v1_3.py`	1.3	MicroPython-Programm für Taster, LEDs und PC-Befehle.
`esp-simulator.py`	1.3	TCP-Ersatz für den ESP32 mit denselben LED- und Statusfunktionen.

Das Kommunikationsmodul bleibt bei Version 1.4, weil seine vorhandene Methode `senden()` bereits beliebige Textzeilen übertragen kann. Die Neuerung von 3.3.5 liegt im Befehlskettenmodul und in den Gegenstellen ESP32/Simulator.

2.1 Empfohlene Ordnerstruktur

```
Dobot-Python/
├── sdk64/
│   ├── __init__.py
│   ├── DobotDllType.py
│   └── DobotDll.dll
└── befehlskette-v3_3_5/
    ├── befehlskette_v3_3_5.py
    ├── befehlskette_beispiel_v3_3_5.py
    ├── esp32_kommunikation_v1_4.py
    ├── esp32_dobot_steuerung_v1_3.py
    └── esp-simulator.py
```

Die Python-Architektur muss zur verwendeten Dobot-DLL passen. Der Dateiname des SDK-Ordners allein beweist nicht, ob die DLL 32- oder 64-Bit ist.

3. Gesamtarchitektur

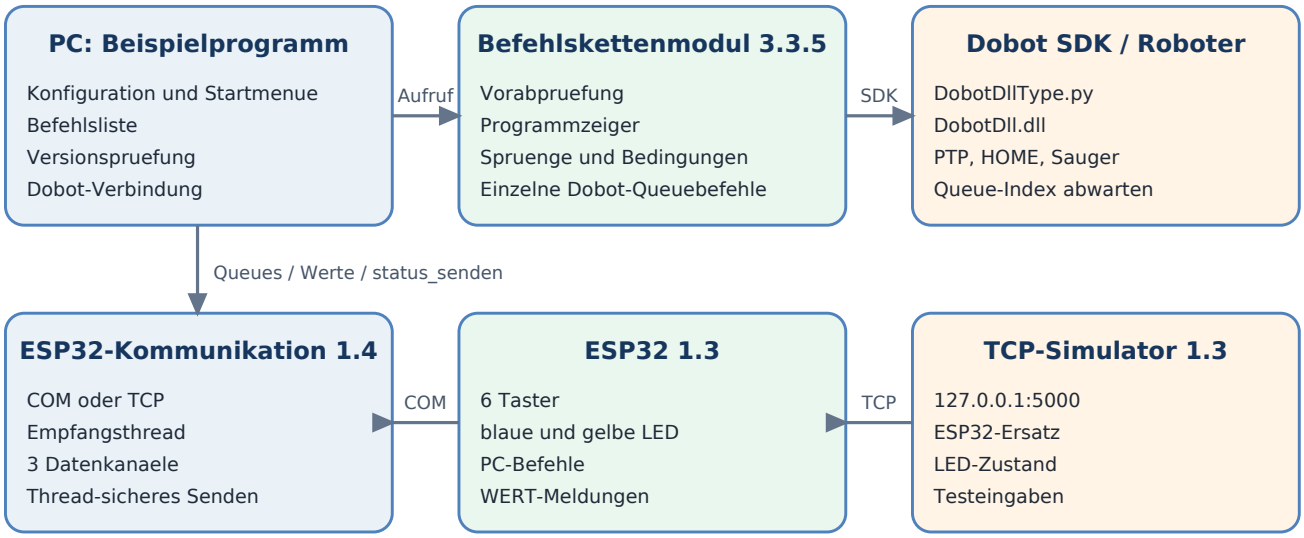

Abbildung 1: Module, Kommunikationswege und Verantwortlichkeiten.

Die Architektur trennt die Aufgaben bewusst. Das Beispielprogramm kümmert sich um Konfiguration und Programmstart. Das Befehlskettenmodul verarbeitet die Ablaufbeschreibung. Das Kommunikationsmodul kapselt COM und TCP. Der Dobot wird ausschließlich durch den Hauptablauf gesteuert; Kommunikationsthreads greifen nicht direkt auf die Roboter-API zu.

3.1 Warum die Dobot-Befehle einzeln eingereiht werden

Frühere Fassungen konnten alle Roboterbefehle vor dem Start in die Dobot-Queue übertragen. Das ist schnell, verhindert aber echte Programmsprünge und bedingte Abläufe. Ab der erweiterten Ablaufsteuerung wird jeder Roboterbefehl einzeln in die Queue gestellt und bis zu seinem Queue-Index abgewartet. Erst danach entscheidet der Programmzeiger, welcher Listeneintrag folgt.

- Vorteil: Marken, Rücksprünge und Bedingungen sind möglich.
- Vorteil: PAUSE, WEITER, HALT und STATUS können zwischen Befehlen verarbeitet werden.
- Nachteil: Der Ablauf ist komplexer und hängt stärker von korrekter Queue-Überwachung ab.

3.2 Drei Datenwege vom ESP32 zum PC

Kanal	Beispiele	Verhalten
Steuerbefehle	`PAUSE, WEITER, HALT, STATUS`	Werden in die Steuerbefehls-Queue gelegt und beeinflussen unmittelbar den Ablauf.
Einmalige Meldungen	`FREIGABE, FREI_1, FERTIG`	Werden in die Meldungs-Queue gelegt. warte_bis entnimmt passende Ereignisse.
Dauerhafte Werte	`WERT;TEMPERATUR;23.7`	Werden im ESPWerteSpeicher abgelegt und bleiben bis zur nächsten Aktualisierung vorhanden.

4. Start und Grundkonfiguration

Im oberen Teil des Beispielprogramms werden nur die projektspezifischen Einstellungen geändert. Der übrige Programmteil ist als gemeinsame Infrastruktur gedacht.

```
COM_MODUS = "serial"  # "serial" oder "tcp"

DOBOT_PORT = "COM10"
DOBOT_BAUDRATE = 115200

# Serieller ESP32:
ESP32_COM_PORT = "COM26"
ESP32_COM_BAUDRATE = 115200

# TCP-Simulator:
ESP32_TCP_HOST = "127.0.0.1"
ESP32_TCP_PORT = 5000

STANDARD_PAUSE_MS = 500
TIMEOUT_SEKUNDEN = 90.0
MAX_SCHRITTE = 1000
```

- Dobot und ESP32 müssen im seriellen Modus verschiedene COM-Ports verwenden.
- COM_MODUS = serial aktiviert den echten ESP32.
- COM_MODUS = tcp aktiviert den lokalen Simulator.
- MAX_SCHRITTE begrenzt Schleifen und schützt vor unbeabsichtigten Endlosschleifen.
- TIMEOUT_SEKUNDEN betrifft das Warten auf Dobot-Queuebefehle.

4.1 Versionsprüfung

Das Beispielprogramm erwartet ausdrücklich Befehlskettenversion 3.3.5 und Kommunikationsmodul 1.4. Stimmen die geladenen Versionen nicht, wird vor einer Roboterbewegung abgebrochen. Dadurch werden versehentlich importierte ältere Dateien sichtbar.

4.2 Startmenü

```
Befehlskette starten?
  s = Start
  a = Abbruch
  t = Testen
Eingabe:
```

- s: Erst jetzt wird der Dobot verbunden und die geprüfte Befehlskette ausgeführt.
- a: Das Programm endet ohne Dobot-Verbindung.
- t: Texte werden wiederholt an ESP32 oder Simulator gesendet. q beendet nur den Sendetest und kehrt zum Startmenü zurück.
- Nach dem Sendetest werden Steuer- und Meldungs-Queue geleert. Dauerhafte ESP-Werte bleiben absichtlich erhalten.

5. Vollständige Befehlsreferenz der Version 3.3.5

Befehl	Grundform	Zweck
fahre_zu	("fahre_zu", x, y, z, r, Text[, Pause_ms])	Absolute lineare Fahrt zu kartesischen Koordinaten.
fahre_um	("fahre_um", dx, dy, dz, dr, Text[, Pause_ms])	Relative lineare Bewegung.
springe_auf	("springe_auf", x, y, z, r, Text[, Pause_ms])	JUMP-Bewegung zu einer absoluten Position.
sauger_ein	("sauger_ein"[, Text[, Pause_ms]])	Sauger und Vakuum einschalten.
sauger_aus	("sauger_aus"[, Text[, Pause_ms]])	Sauger und Vakuum ausschalten.
sauger_status	("sauger_status"[, Text[, Pause_ms]])	Aktuellen Saugerzustand anzeigen.
home	("home"[, Text[, Pause_ms]])	HOME-Fahrt ausführen.
geschwindigkeit	("geschwindigkeit", Geschw.[, Beschl.], Text[, Pause_ms])	PTP-Geschwindigkeit und Beschleunigung in Prozent setzen.
warte_bis	("warte_bis", Meldung[, Text[, Timeout_s]])	Auf eine einmalige ESP-/Simulator-Meldung warten.
warte_bis_wert	("warte_bis_wert", Name[, Operator], Sollwert[, Text[, Timeout_s]])	Auf einen dauerhaften Wertvergleich warten.
wert_anzeigen	("wert_anzeigen", Name[, Text])	Gespeicherten ESP-Wert anzeigen.
wenn_wert	("wenn_wert", Name[, Operator], Sollwert, Wahr-Befehl, Falsch-Befehl)	Einen eingebetteten Befehl abhängig vom Wert ausführen.
marke	("marke", Name)	Eine Sprungstelle definieren.
gehe_zu_befehl	("gehe_zu_befehl", Markenname)	Den Programmzeiger zu einer Marke setzen.
esp_senden	("esp_senden", Nachricht[, Text])	Frei wählbare Textzeile an ESP32 oder Simulator senden.

5.1 Vergleichsoperatoren

Für wenn_wert und warte_bis_wert stehen folgende Operatoren zur Verfügung:

```
==    !=    <    <=    >    >=
```

Die Aliase = und <> werden zu == beziehungsweise != normalisiert. Ohne Operator wird aus Gründen der Rückwärtskompatibilität == verwendet.

5.2 Regeln für eingebettete Befehle in wenn_wert

- Wahr- und Falsch-Zweig sind jeweils genau ein normaler Befehlstupel.
- Eine Marke darf nicht unmittelbar als Zweig verwendet werden.
- Eine weitere wenn_wert-Bedingung darf nicht direkt verschachtelt werden.
- Ein Sprungbefehl ist als Zweig zulässig und wird im Beispielprogramm verwendet.

5.3 Neuer Befehl esp_senden

```
("esp_senden", "LED_GELB_EIN")
("esp_senden", "LED_GELB_AUS", "Gelbe LED ausschalten")
("esp_senden", "DISPLAY;Dobot arbeitet", "Text an Display senden")
```

- Die Nachricht darf nicht leer sein.
- Zeilenumbrüche sind nicht erlaubt, weil eine Textzeile genau eine Protokollnachricht darstellt.
- Fehlt die ESP-Verbindung oder schlägt senden() fehl, muss die Ausführung als Fehler behandelt werden.
- Der Text kann auch in einem Wahr- oder Falsch-Zweig von wenn_wert verwendet werden.

6. Prüfung vor dem Verbindungsaufbau

Die Funktion befehlskette_pruefen() verarbeitet die gesamte Befehlsliste, bevor eine Verbindung zum Dobot aufgebaut wird. Das ist ein zentraler Sicherheits- und Unterrichtsvorteil: Schreibfehler werden sichtbar, bevor sich der Roboter bewegen kann.

- Unbekannte Befehlsnamen werden abgelehnt.
- Anzahl und Typ der Tupelwerte werden geprüft.
- Pausen und Timeouts müssen zulässige Zahlenwerte besitzen.
- Geschwindigkeit und Beschleunigung müssen zwischen 1 und 100 Prozent liegen.
- Markennamen dürfen nicht leer und nicht doppelt vorhanden sein.
- Jedes Sprungziel muss existieren, auch innerhalb von wenn_wert-Zweigen.
- Vergleichsoperatoren werden kontrolliert.
- esp_senden-Nachrichten dürfen nicht leer sein und keinen Zeilenumbruch enthalten.

> Die Prüfung bestätigt Syntax und Ablaufstruktur. Sie kann nicht feststellen, ob eine Koordinate mechanisch sicher erreichbar ist.

7. Laufzeitsteuerung: PAUSE, WEITER, HALT und STATUS

Eingabe	Bedeutung	Wirkung
`p / PAUSE`	Pause	Laufende Ausführung wird angehalten; Queue-Ausführung wird gestoppt beziehungsweise der Ablauf wartet.
`w / WEITER`	Fortsetzen	Pausierten Ablauf fortsetzen.
`h / HALT`	Endgültiger Abbruch	Befehlskette beendet sich mit Zustand HALT. Vor Neustart Aufbau und Roboter prüfen.
`? / STATUS`	Statusabfrage	Aktuelle Programmposition und nächster Befehl werden ausgegeben.

Die Steuerbefehle können von der PC-Tastatur oder vom ESP32/Simulator kommen. Beide Quellen legen normalisierte Einträge mit Quellenangabe in dieselbe Queue.

8. ESP32-Kommunikation Version 1.4

Das Modul kapselt sowohl die serielle Verbindung als auch die TCP-Verbindung. Beide Varianten verwenden dieselbe Nachrichtenverarbeitung und dieselben Datenstrukturen.

- ESP32SerielleSteuerung öffnet einen COM-Port mit pySerial und kann Wiederverbindungen versuchen.
- ESP32TCPSteuerung verbindet sich als TCP-Client mit dem Simulator.
- Ein Empfangsthread liest Textzeilen, ohne den Dobot-Hauptablauf zu blockieren.
- Ein Sende-Lock verhindert, dass mehrere Threads gleichzeitig ineinander verschachtelte Textzeilen schreiben.
- ESPWerteSpeicher verwendet ein RLock und speichert Wert, Originalname, Quelle, Zeit und Änderungsnummer.

8.1 Handshake

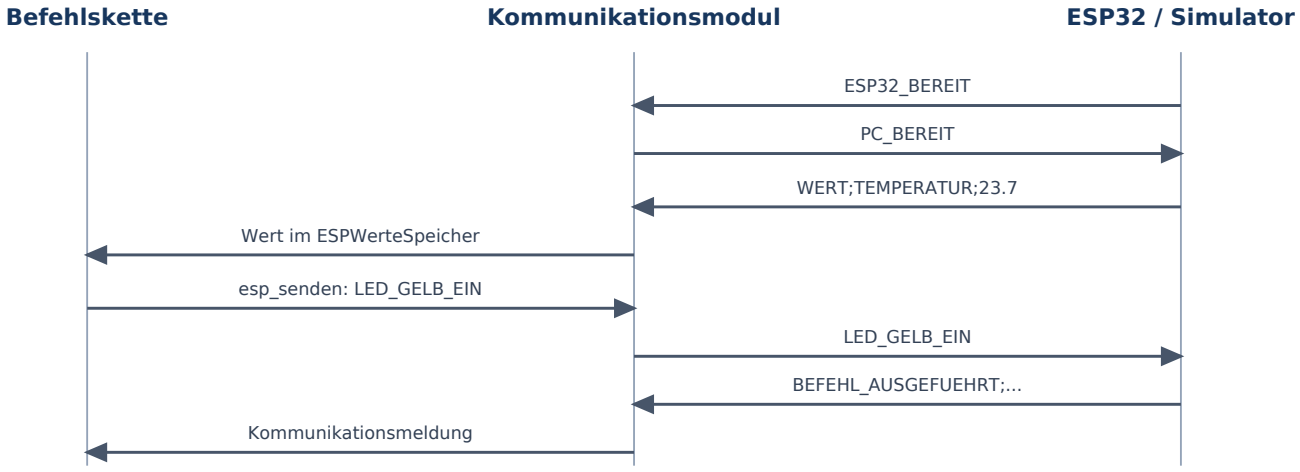

Abbildung 2: Beispielhafter Nachrichtenfluss zwischen Befehlskette, Kommunikationsmodul und ESP32/Simulator.

```
ESP32 -> PC: ESP32_BEREIT
PC -> ESP32: PC_BEREIT
ESP32 -> PC: PC_BEREIT_BESTAETIGT
```

8.2 Werteformat

```
WERT;POSITION;POS_A
WERT;TASTER;1
WERT;TEMPERATUR;23.7
WERT;LED_gelb;0
```

Zahlen werden automatisch in int oder float umgewandelt. Andere Inhalte bleiben Texte. Wertnamen werden beim Suchen ohne Beachtung der Groß-/Kleinschreibung normalisiert, der Originalname bleibt im gespeicherten Eintrag erhalten.

8.3 Einmalige Nachrichten

Jede andere empfangene Zeile, die weder Steuerbefehl noch WERT-Zeile ist, wird als Kommunikationsmeldung behandelt. Zusätzlich wird sie als LETZTE_NACHRICHT im Wertespeicher abgelegt.

9. Statusmeldungen vom PC an ESP32 oder Simulator

Meldung	Bedeutung
`BEFEHLSKETTE_GEPRUEFT;Anzahl`	Die Befehlsliste wurde vollständig geprüft.
`PROGRAMM_GESTARTET;3.3.5`	Die eigentliche Ausführung beginnt.
`BEFEHL_GESTARTET;Nr;Name;Text`	Ein Programmbefehl startet.
`BEFEHL_FERTIG;Nr;Name`	Ein Programmbefehl ist beendet.
`MARKE;Name`	Eine Marke wurde erreicht.
`SPRUNG;Ziel`	Der Programmzeiger springt zu einer Marke.
`WARTE_AUF;MELDUNG;Name`	Der Ablauf wartet auf eine einmalige Meldung.
`WARTE_AUF;WERT;Name;Operator;Soll`	Der Ablauf wartet auf einen Wertvergleich.
`BEDINGUNG;Name;Ist;Operator;Soll;Ergebnis;...`	Eine wenn_wert-Bedingung wurde ausgewertet.
`BEFEHLSKETTE_BEENDET;Grund`	Das Befehlskettenmodul ist fertig oder wurde angehalten.
`PROGRAMM_BEENDET;Grund`	Das Beispielprogramm meldet seinen endgültigen Abschluss.

Semikolons, Zeilenumbrüche und Wagenrückläufe in freien Textfeldern werden für Statusmeldungen bereinigt, damit das Zeilenprotokoll eindeutig bleibt.

10. ESP32-Steuerprogramm Version 1.3

Das MicroPython-Programm bildet die reale Gegenstelle zur Befehlskette. Es liest Taster, steuert zwei LEDs, sendet Zustandsänderungen und verarbeitet Befehle vom PC.

GPIO	Bauteil	Funktion
25	Taster PAUSE	sendet PAUSE
26	Taster WEITER	sendet WEITER
27	Taster HALT	sendet HALT
33	Taster STATUS	sendet STATUS
18	Taster FREI 1	sendet FREI_1
32	Taster FREI 2	sendet FREI_2
2	Blaue LED	Online-Anzeige nach PC_BEREIT
19	Gelbe LED	Simulations- und PC-gesteuerter Ausgang

Die Taster arbeiten mit internem Pull-up: 0 bedeutet gedrückt, 1 bedeutet nicht gedrückt. Die Klasse Taste entprellt die Eingänge und sendet nur den Druckvorgang als Steuerbefehl.

10.1 Verarbeitete PC-Befehle

Befehl	Reaktion
`LED_GELB_EIN`	Zufallssimulation stoppen, gelbe LED einschalten, Erfolg melden.
`LED_GELB_AUS`	Zufallssimulation stoppen, gelbe LED ausschalten, Erfolg melden.
`LED_GELB_UMSCHALTEN`	Zufallssimulation stoppen und LED-Zustand umschalten.
`LED_GELB_STATUS`	WERT;LED_gelb;0/1 senden.
`SIMULATION_LED_START`	Simulation mit Standardbereich 30 bis 60 s starten.
`SIMULATION_LED_START;min;max`	Simulation mit angegebenem ganzzahligen Zeitbereich starten.
`SIMULATION_LED_STOP`	Zufallssimulation beenden.
`ESP32_STATUS`	Version, blaue LED, gelbe LED und Simulationszustand senden.
`PING`	PONG senden.
`HILFE`	Liste der unterstützten Befehle senden.

10.2 Nicht blockierende LED-Simulation

Version 1.2/1.3 verzichtet für die LED-Simulation auf einen zusätzlichen MicroPython-Thread. Stattdessen wird in jedem Durchlauf der Hauptschleife mit ticks_ms, ticks_add und ticks_diff geprüft, ob die nächste Zufallszeit erreicht wurde. Dadurch bleiben Taster und PC-Eingang ansprechbar.

```
while True:
    tasten_pruefen()
    simulation_led_pruefen()
    ueberwache()
    pc_nachrichten_lesen()
    time.sleep_ms(SCHLEIFENPAUSE_MS)
```

10.3 Rückmeldungen

```
BEFEHL_AUSGEFUEHRT;LED_GELB_EIN
BEFEHL_FEHLER;SIMULATION_LED_START;...
UNBEKANNTER_BEFEHL;Mein Text
ESP32_STATUS;VERSION=1.3;LED_BLAU=1;LED_GELB=0;SIMULATION=1
```

11. TCP-Simulator Version 1.3

Der Simulator ist ein TCP-Server auf 127.0.0.1:5000. Er ersetzt den echten ESP32 für PC-Tests und verarbeitet dieselben LED-, Status-, Hilfe- und Simulationsbefehle.

- Er verwaltet simulierte Zustände für blaue LED, gelbe LED und Zufallssimulation.
- Ein Empfangsthread verarbeitet PC-Nachrichten.
- Ein Simulationsthread schaltet die gelbe LED in zufälligen Abständen um.
- Ein Sende-Lock schützt parallele Ausgaben auf dem TCP-Socket.
- Lokale Tastatureingaben erzeugen Steuerbefehle, Meldungen und Werte.

11.1 Eingaben des Simulators

Eingabe	Gesendete Nachricht / Wirkung
`p, w, h, ?`	PAUSE, WEITER, HALT, STATUS
`a, b`	WERT;POSITION;POS_A beziehungsweise POS_B
`f, f1, f2`	FREIGABE, FREI_1, FREI_2
`t1, t0`	WERT;TASTER;1 beziehungsweise 0
`temp25, temp35`	WERT;TEMPERATUR;25 beziehungsweise 35
`g1, g0, gu, gs`	Gelbe LED setzen, umschalten oder Status senden
`sim 3 6`	Zufallssimulation zwischen 3 und 6 s starten
`sim_stop`	Zufallssimulation beenden
`q`	Simulator beenden

Beliebige andere Eingaben können direkt als ESP-Nachricht an das PC-Programm gesendet werden. Das erleichtert Tests neuer Protokollbefehle.

12. Ablauf des mitgelieferten Beispielprogramms

```
befehle = [
    ("geschwindigkeit", 40, 40, "Geschwindigkeit und Beschleunigung auf 40 % setzen", 0),
    ("home", "HOME-Fahrt ausführen", 500),
    ("marke", "auf_freigabe_warten"),
    ("warte_bis", "FREIGABE", "Warte auf FREIGABE vom COM-/TCP-Client", None),
    ("wert_anzeigen", "TEMPERATUR", "Vom ESP gespeicherte Temperatur anzeigen"),
    ("wenn_wert", "TEMPERATUR", ">=", 30, ("wert_anzeigen", "TEMPERATUR", "Temperaturwarnung: mindestens 30 °C")
, ("wert_anzeigen", "TEMPERATUR", "Temperatur liegt unter 30 °C")),
    ("wert_anzeigen", "POSITION", "Vom ESP gespeicherte Position anzeigen"),
    ("wenn_wert", "POSITION", "==", "POS_A", ("fahre_zu", 240, 140, 70, 0, "Fahre zu Position A", 500), ("gehe_z
u_befehl", "POS_B")),
    ("gehe_zu_befehl", "POSITION_ENDE"),
    ("marke", "POS_B"),
    ("fahre_zu", 180, 160, 50, 0, "Fahre zu Position B", 500),
    ("marke", "POSITION_ENDE"),
    ("gehe_zu_befehl", "auf_freigabe_warten"),
]
```

Der Ablauf arbeitet als Endlosschleife, die durch MAX_SCHRITTE = 1000 begrenzt wird. Vor jeder Runde wartet er auf FREIGABE. Temperatur und Position werden aus dem permanenten ESPWerteSpeicher gelesen. Abhängig von POSITION fährt der Dobot zu A oder B.

> Der neue Befehl esp_senden ist im Beispiel nur als auskommentierter zusätzlicher Befehl dokumentiert. Für den praktischen LED-Test muss er bewusst in die Befehlsliste eingefügt werden.

12.1 Sinnvolle Testvariante mit esp_senden

```
befehle = [
    ("esp_senden", "LED_GELB_EIN", "Gelbe LED am ESP32 einschalten"),
    ("warte_bis", "FREI_1", "Warte auf Taster FREI 1", None),
    ("esp_senden", "LED_GELB_AUS", "Gelbe LED am ESP32 ausschalten"),
]
```

Alle Tupel bleiben entsprechend der vereinbarten Schreibweise vollständig in jeweils einer Zeile.

13. Empfohlene Betriebsarten

13.1 Reiner Kommunikationstest ohne Dobot

- COM_MODUS auf serial oder tcp einstellen.
- ESP32-Programm beziehungsweise Simulator starten.
- Beispielprogramm starten und im Menü t wählen.
- Befehle wie PING, HILFE, ESP32_STATUS, LED_GELB_EIN und LED_GELB_AUS eingeben.
- Mit q zum Startmenü zurückkehren. Den Dobot nicht starten.

13.2 Test mit TCP-Simulator

- Zuerst esp-simulator.py starten; er wartet auf 127.0.0.1:5000.
- Danach das Beispielprogramm mit COM_MODUS = tcp starten.
- Im Simulator temp25 oder temp35, a oder b und f eingeben.
- Programmausgaben, Statusmeldungen und simulierte LED-Reaktionen vergleichen.

13.3 Test mit echtem ESP32

- esp32_dobot_steuerung_v1_3.py als main.py oder über den vorgesehenen MicroPython-Start laden.
- Thonny darf den seriellen Port nicht gleichzeitig blockieren, wenn das PC-Programm ihn öffnen soll.
- COM_MODUS = serial und korrekten ESP32_COM_PORT einstellen.
- Zuerst den Sendetest ausführen, danach die Befehlskette starten.

14. Fehlerbehandlung und Sicherheitsmechanismen

Mechanismus	Schutzwirkung
Vorabprüfung	Syntax- und Strukturfehler werden vor der Dobot-Verbindung erkannt.
Versionsvergleich	Falsche Modulversionen führen zu einem verständlichen Abbruch.
Getrennte COM-Ports	Dobot und ESP32 können nicht versehentlich denselben Port erhalten.
MAX_SCHRITTE	Beendet unbeabsichtigte Endlosschleifen nach einer festen Schrittzahl.
Timeouts	Dobot-Queue, warte_bis und warte_bis_wert können begrenzt werden.
HALT-Zustand	Beendet den Ablauf endgültig und fordert eine Kontrolle vor dem Neustart.
finally-Aufräumen	Queue wird gestoppt, Kommunikation beendet und Dobot getrennt.
ESP32-Befehlsfehler	Ungültige Parameter und unbekannte Befehle werden zurückgemeldet.

14.1 Wichtige praktische Sicherheitsregeln

- Vor jeder HOME-Fahrt muss der Arbeitsraum frei sein.
- Neue Koordinaten zunächst mit geringer Geschwindigkeit und großem Sicherheitsabstand testen.
- HALT ist kein Ersatz für einen hardwareseitigen Not-Aus.
- Nach HALT oder Alarm Roboterstellung, Werkstück und Arbeitsplatte kontrollieren.
- Saugerzustand und Greifhöhe vor Pick-and-Place-Abläufen separat prüfen.

15. Systematischer Testplan für den aktuellen Stand

Nr.	Test	Erwartetes Ergebnis	Status
1	Versions- und Importtest	3.3.5 und 1.4 werden angezeigt; richtige Dateien werden geladen.	offen
2	Vorabprüfung mit korrekter Liste	Liste wird vollständig angezeigt, noch keine Dobot-Bewegung.	offen
3	Vorabprüfung mit absichtlichem Fehler	Verständlicher ValueError vor Dobot-Verbindung.	offen
4	TCP-Handschlag	ESP32_BEREIT, PC_BEREIT und Bestätigung sichtbar.	offen
5	Serieller Handschlag	Blaue ESP32-LED leuchtet; Bestätigung am PC.	offen
6	Sendetest PING/HILFE/STATUS	PONG, Befehlsliste und Statuszeile werden empfangen.	offen
7	LED_GELB_EIN/AUS	LED beziehungsweise Simulatorzustand ändert sich und wird bestätigt.	offen
8	Ungültiger LED-Simulationsbereich	BEFEHL_FEHLER ohne Programmabsturz.	offen
9	Steuerbefehle PAUSE/WEITER/STATUS	Ablauf reagiert von Tastatur und ESP32 identisch.	offen
10	HALT	Befehlskette endet mit HALT und fordert Kontrolle.	offen
11	warte_bis FREIGABE	Ablauf bleibt stehen und setzt nach Ereignis fort.	offen
12	werte speichern	TEMPERATUR und POSITION werden korrekt aktualisiert.	offen
13	wenn_wert A/B	Richtiger Bewegungszweig wird gewählt.	offen
14	esp_senden in Befehlskette	Text wird übertragen; ESP32/Simulator reagiert.	offen
15	Simulator-Echtgerät-Vergleich	Gleiche Befehle erzeugen gleichwertige Protokollantworten.	offen

Die Statusspalte bleibt bewusst offen. Diese Dokumentation soll nicht behaupten, dass noch nicht beobachtete Hardwaretests bereits erfolgreich waren.

16. Bekannte Grenzen und offene Punkte

- Die Gesamtintegration 3.3.5 + ESP32 1.3 + Simulator 1.3 wird aktuell praktisch getestet.
- Das Beispielprogramm enthält esp_senden noch nicht aktiv in seiner Standardbefehlsliste.
- Die status_senden-Rückmeldungen werden vom ESP32 1.3 gespeichert, aber noch nicht auf einem Display oder mit mehreren LEDs visualisiert.

- Das Protokoll besitzt noch keine Nachrichten-ID und keine verbindliche Bestätigung, auf die die Befehlskette aktiv wartet.
- Ein Kommunikationsabbruch während esp_senden muss praktisch auf COM und TCP geprüft werden.
- Mechanische Arbeitsraumgrenzen werden nicht durch die Befehlskettenprüfung kontrolliert.
- Der Simulator bildet elektrische Besonderheiten, Entprellung und USB-Portkonflikte des echten ESP32 nicht vollständig ab.

16.1 Technischer Punkt: Bestätigung versus Versand

Der Befehl esp_senden bewertet zunächst, ob die Textzeile an die Kommunikationsschicht übergeben werden konnte. Die spätere Antwort BEFEHL_AUSGEFUEHRT bestätigt zwar die Gegenstelle, wird aber derzeit als normale Kommunikationsmeldung empfangen. Die Befehlskette wartet nicht automatisch auf diese Bestätigung. Für besonders kritische Aktorbefehle wäre später ein eigener bestätigter Sendebefehl sinnvoll.

17. Entwicklung von 3.3 bis 3.3.5

Version	Hauptänderung
3.3	ESP32 über serielle COM-Verbindung als zweite Steuerquelle neben der Tastatur.
3.3.1	Umschaltung zwischen echtem ESP32 über COM und TCP-Simulator.
3.3.2	Marken, Sprünge, warte_bis, HOME und Geschwindigkeitsbefehl.
3.3.3	Permanente ESP-Werte sowie wert_anzeigen, warte_bis_wert und wenn_wert.
3.3.4	Vollständige Vorabprüfung, Operatoren == != < <= > >= und laufende Statusmeldungen.
3.3.4.1	Startmenü mit Start, Abbruch und Kommunikationstest; Hauptmodul bleibt 3.3.4.
3.3.5	esp_senden für frei wählbare PC-zu-ESP32-Nachrichten; Sendetest bis q.

Merksatz: anbinden - Kommunikationsweg wählen - Ablauf steuern - Werte verarbeiten - prüfen und rückmelden - aktiv an den ESP32 senden.

18. Empfohlene nächste Entwicklungsschritte

Für eine stabile Freigabe der Version 3.3.5 sind zunächst keine großen neuen Funktionen nötig. Vorrang hat der vollständige Test der vorhandenen Funktionen.

- Testprotokoll mit Datum, Betriebsart, Eingabe, Ausgabe und Ergebnis führen.
- Simulator und echter ESP32 mit identischer Befehlsfolge vergleichen.
- esp_senden im Beispielprogramm mit LED_GELB_EIN und LED_GELB_AUS aktiv testen.
- Kommunikationsabbruch während des Sendens gezielt provozieren.
- Nach erfolgreichem Test README und PDF um bestätigte Ergebnisse ergänzen.
- Erst danach über Version 3.3.6 oder 3.4 entscheiden.

18.1 Mögliche spätere Erweiterung

Ein neuer Befehl wie esp_senden_und_warten könnte eine Nachricht senden und auf eine genau definierte Bestätigung warten. Dazu wären Nachrichten-IDs oder eindeutig zuordenbare Bestätigungstexte sinnvoll. Das ist eine mögliche Weiterentwicklung, aber nicht Bestandteil von 3.3.5.

Anhang A: Kurze Testbefehle

```
# Freier Sendetest im Startmenü:
PING
HILFE
ESP32_STATUS
LED_GELB_EIN
LED_GELB_STATUS
LED_GELB_AUS
SIMULATION_LED_START;3;6
SIMULATION_LED_STOP
q

# Simulator-Eingaben:
temp25
a
f
?
p
w
h
```

Anhang B: Beispiel für Wertbedingungen

```
("warte_bis_wert", "TASTER", "==", 1, "Warte auf gedrückten Taster", None)
("wenn_wert", "TEMPERATUR", ">=", 30, ("esp_senden", "LED_GELB_EIN", "Warnanzeige einschalten"), ("e
sp_senden", "LED_GELB_AUS", "Warnanzeige ausschalten"))
```

Anhang C: Abschlussbewertung

Die Version 3.3.5 ist ein wichtiger Entwicklungsschritt: Der ESP32 ist nicht mehr nur Eingabegerät und Empfänger von Statusmeldungen, sondern kann gezielt durch Befehle aus dem Roboterablauf angesteuert werden. Die Trennung aus Befehlskettenmodul, Kommunikationsmodul, realem ESP32-Programm und Simulator ist übersichtlich und gut für Unterrichtstests geeignet. Vor einer endgültigen Freigabe müssen die laufenden Praxistests abgeschlossen und dokumentiert werden.