

## # Taschenrechner

Ein programmierbarer **UPN-Rechner** (Umgekehrte Polnische Notation, RPN) im Stil der klassischen HP-Taschenrechner, in python3 geschrieben.

**Maintainer:** Andreas Müller ([andreas@poipoi.de](mailto:andreas@poipoi.de))

## Inhaltsverzeichnis

- [Anforderungen](#)
- [Setup](#)
- [Starten](#)
- [Kurzanleitung](#)
- [UPN-Eingabe \(RUN-Modus\)](#)
- [Prgm/Run-Schalter](#)
- [Stack-Lift-Schalter \(SLON / SLOFF\)](#)
- [Programm erstellen \(PRGM-Modus\)](#)
- [Ausführen \(RUN-Modus\)](#)
- [Anzeige-Formate \(wie HP\)](#)
- [Mathematische Funktionen](#)
- [Prozent- und Zeitfunktionen \(HP-41CV\)](#)
- [Befehle \(Auswahl\)](#)
- [Register-Arithmetik \(ST0+ / ST0- / ST0\\* / ST0/\)](#)
- [Stack-Manipulation \(FORTH-artig: DUP / DROP\)](#)
- [Fehlerwiederherstellung \(LASTX / LASTY\)](#)
- [Statistik \(S+ / S- / MEAN / SDEV / CLS\)](#)
- [Zufallszahlen \(RND, Schneiers Fortuna\)](#)
- [Zeit/Datum \(TIME / DATE\)](#)
- [Datumsarithmetik \(DATE+ / DATE- / DDAYS\)](#)
- [GPIO-Pin-Zuordnung \(BOARD / BCM, Raspberry Pi\)](#)
- [GPIO-Geräte \(GPIO <Befehl>, per gpiozero\)](#)
- [Bedingte Tests \(wie HP 9100A\)](#)
- [Flag-Befehle \(SF / CF / FS? / FS?C\)](#)
- [Unterprogramme \(CALL / RTN\)](#)
- [Programm-Aufruf \(XEQ <name>\)](#)
- [Programm anhalten und fortsetzen \(STOP / CONT\)](#)
- [Pause \(PAUSE\)](#)
- [Step \(STEP \[n\]\)](#)
- [Trace \(TRACE\)](#)
- [Step-Limit \(LIMIT ON / LIMIT OFF\)](#)
- [Alarme \(ALMSET / ALMCAT / ALMDEL / ALMON / ALMOFF\)](#)
- [Flowchart \(FLOWCHART\)](#)
- [Grafik \(PT / PT? / LINE / BOX / CIRC / CLG / TEXT / PLOT / GRID / PEN / BG / GRAPH\)](#)
- [Programme & Persistenz](#)
- [Entwicklung](#)
- [Paket \(.deb\) bauen & installieren](#)
- [Projektlayout](#)

### Kernidee:

- 4-stufiger Register-Stack **X, Y, Z, W** plus lastX/lastY.
- **Zwei Modi** über den Prgm/Run-Schalter:
- RUN – Laufzeitmodus: Zahlen eingeben, Operationen sofort ausführen, das gespeicherte Programm starten.
- PRGM – Programmiermodus: eine Folge von Anweisungen (Programm) zusammenstellen, auflisten, löschen.
- Programme mit Labels und Sprüngen (LBL/GT0/ISG/DSG/RTN/PSE), Unterprogrammen (CALL <label> bis RTN) sowie Programm-Aufruf (XEQ <name> – LBL-Label oder Datei aus ~/.taschenrechner/Programme/), bedingten Tests (X=0/X=Y/X<Y/X>Y/X>0/X<0, HP 9100A), Flag-Verzweigungen

(SF/CF/FS?/FS?C: Status-Flag setzen/löschen/ prüfen, FS?C löscht das Flag zusätzlich nach dem Test), Textausgabe (PRINT [Text], mit Platzhaltern %x-%w und %<n> für Register), Programm-Anhalten und -Fortsetzen (STOP/CONT), Pause (PAUSE, wie sleep in der Shell), Schritt-Steuerung (STEP [n], Programm anweisungsweise ausführen), Zeit/Datum (TIME/DATE, lokale Zeit als HH.MMSShh bzw. Datum als DD.MMYYYY – Dezimalzahlen, auch negativ verwendbar) sowie Datumsarithmetik (DATE+/DATE-/DDAYS, Tage addieren/abziehen bzw. Tagesdifferenz), Step-Limit (LIMIT ON/LIMIT OFF, schaltbar – aus für Endlosschleifen), Speicherregistern 0–99, Trigonometrie, Logarithmen und Exponenten (LN, LOG, E^X, Y^X), Rest (MOD), Zufallszahlen (RND/RND <n> per Schneiers Fortuna), GPIO-Pin-Zuordnung (BOARD/BCM für die 40-polige Raspberry-Pi-Steckleiste) und GPIO-Geräte (LED, PWMLED, BUZZER, TONALBUZZER, MOTOR, RGBLED, PIR, LDR, HCSR04, BUTTON – per gpiozero, lazy Import), Alarmer (ALMSET / ALMCAT / ALMDEL / ALMON / ALMOFF) welche via XEQ das zeitgesteuerte Starten von Programmen gestatten sowie ein **Grafik-Fenster** (Tkinter, optional): Punkte/Linien/Boxen/Kreise (PT/LINE/BOX/CIRC), Text (winkel,TEXT), Funktionsgraphen per Programm (PLOT <name> <x0> <x1> <n>), Achsen-Raster mit Abschnitten und optionalem Y-Fenster als Zoom (GRID <n> <m> [<k> <s>], GRID OFF = Auto), Stift/Hintergrund (PEN/BG) und Pixel-Prüfung (PT?); Fenster via GRAPH/GRAPH CLOSE/GRAPH SAVE <name> (PNG, reines Rendering ohne Tkinter).

## Anforderungen

- Python **>= 3.10**
- **Optional** gpiozero (nur für die GPIO-Geräte GPIO <Befehl>):

```
bash pip install -e ".[dev,gpio]" # dev + gpiozero pip install
"gpiozero>=1.6" # oder nur gpiozero
```

Ohne gpiozero läuft der Rechner unverändert weiter; GPIO <Befehl> liefert dann GPIO NOT AVAILABLE.

- **Optional** python3-tk (nur für das Grafik-Fenster GRAPH):

```
bash sudo apt install python3-tk # Debian/Ubuntu/Raspberry Pi OS
```

Ohne python3-tk läuft der Rechner unverändert weiter; GRAPH liefert dann GRAPH NOT AVAILABLE.

## Setup

Entwicklungsumgebung (venv) für Tests/Lint/Typpen – die **laufende** App wird separat über das .deb-Paket installiert (siehe „Paket (.deb) bauen & installieren“).

```
python3 -m venv .venv
source .venv/bin/activate # Windows: .venv\Scripts\activate
pip install -e .[dev]
```

## Starten

```
python -m taschenrechner
# oder nach dem Installieren des .deb Paketes:
taschenrechner
```

**Hinweis:** python -m taschenrechner (aus der venv) läuft auf dem **Quellcode**; taschenrechner (installierte Kopie) läuft auf dem **System-Python** und den

Dateien aus dist-packages.

Die Versionsnummer zeigt taschenrechner -v (bzw. --version) an und wird danach beendet; sie erscheint außerdem in der Startmeldung. Die Version steht in pyproject.toml bzw. taschenrechner/\_\_init\_\_.py (\_\_version\_\_) und wird von build-deb.sh synchron gehalten.

## Kurzanleitung

### UPN-Eingabe (RUN-Modus)

ENTER setzt das aktive Register X nach Y ab. Danach wird der zweite Wert eingegeben und die Operation ausgeführt. Siehe auch SLON/SLOFF.

```
[RUN] 5
[RUN] ENTER
[RUN] 3
[RUN] +          => X = 8
```

**Leere Eingabe:** Nur Enter drücken (ohne Argumente) wirkt genau wie der ENTER-Befehl – im **RUN-Modus** wird X nach Y abgesetzt, im **PRGM-Modus** wird ENTER als Programmzeile aufgenommen (der Stack bleibt dabei unverändert).

### Prgm/Run-Schalter

MODE legt den Schalter um; PRGM bzw. RUN wählen den Modus direkt. Der Wechsel **in den PRGM-Modus** (PRGM, bzw. MODE, wenn das Ergebnis PRGM ist) zeigt das aktuelle Programm sofort an (LIST).

### Stack-Lift-Schalter (SLON / SLOFF)

SLON schaltet den Stack-Lift ein, SLOFF aus (Default ist SLOFF). Der Schalter ist persistent (wird mit dem Zustand gespeichert) und wirkt im **RUN- und PRGM-Modus** (bzw. bei der Programmausführung).

Im SLON-Modus (Verhalten wie **HP-25**) überschreiben Zahleneingabe, RCL <n>, die Konstanten PI/E, TIME/DATE, RND sowie die Wiederherstellungs-Befehle LASTX/LASTY den X-Wert **nicht**, sondern rollen den alten X-Wert vorher nach Y auf (Stack-Lift, x→y, y→z, z→w), bevor der neue Wert in X geschrieben wird.

Im SLOFF-Modus wird X einfach überschrieben.

### Programm erstellen (PRGM-Modus)

```
[PRGM] LBL L
[PRGM] RCL 0
[PRGM] ENTER
[PRGM] RCL 1
[PRGM] +
[PRGM] STO 0
[PRGM] DSG 1
[PRGM] GTO L
[PRGM] RTN
[PRGM] LIST
```

### Ausführen (RUN-Modus)

Die Summe 1 + 2 + ... + N (Zähler in Register 1, Ergebnis in Register 0):

```
[RUN] 0
```

```
[RUN] STO 0
[RUN] 3
[RUN] STO 1
[RUN] GO          => X = 6 (1+2+3)
```

## Anzeige-Formate (wie HP)

Betreffen nur die Anzeige, nicht den gerechneten Wert (Vollpräzision bleibt erhalten):

| Befehl  | Bedeutung                                    | Beispiel (X=12345.6789) |
|---------|----------------------------------------------|-------------------------|
| NORM    | normale Darstellung (Vollpräzision)          | 12345.6789              |
| FIX <n> | n Stellen nach dem Komma                     | FIX 2 → 12345.68        |
| SCI <n> | n Stellen nach dem Komma, Exponent beliebig  | SCI 4 → 1.2346E4        |
| ENG <n> | n Stellen nach dem Komma, Exponent 0 (mod 3) | ENG 4 → 12.3457E3       |

## Mathematische Funktionen

Die Funktionen wirken auf den Stack: **unäre** wirken nur auf X, **binäre** lesen Y und X und schreiben das Ergebnis nach X. Winkel folgen dem eingestellten Winkelmodus (RAD/DEG, Default RAD).

### Grundrechenarten & Kehrwert:

| Befehl | Bedeutung                                                                    | Beispiel (UPN)      |
|--------|------------------------------------------------------------------------------|---------------------|
| +      | Addition: $Y + X$                                                            | 5 ENTER 3 + → 8     |
| -      | Subtraktion: $Y - X$                                                         | 10 ENTER 4 - → 6    |
| *      | Multiplikation: $Y \times X$                                                 | 6 ENTER 7 * → 42    |
| /      | Division: $Y \div X$ ( $X = 0 \rightarrow \text{DIV BY } 0$ )                | 15 ENTER 4 / → 3.75 |
| MOD    | Rest von $Y/X$ (binär): $Y \bmod X$ ( $X = 0 \rightarrow \text{DIV BY } 0$ ) | 10 ENTER 3 MOD → 1  |
| 1/X    | Kehrwert: $1/X$ (auch als INV; $X = 0 \rightarrow \text{DIV BY } 0$ )        | 4 1/X → 0.25        |

MOD (Rest, binär):  $X \leftarrow Y \bmod X$  – der Rest der Division von Y (Dividend) durch X (Divisor); das Ergebnis hat das Vorzeichen des Divisors (Modulo-Konvention wie in Python), z. B. 10 ENTER 3 MOD → 1, -10 ENTER 3 MOD → 2. Bei  $X = 0 \rightarrow \text{DIV BY } 0$ . Verfügbar im RUN- und PRGM-Modus.

### Potenz & Wurzel:

| Befehl | Bedeutung                                                                                                                        | Beispiel (UPN)          |
|--------|----------------------------------------------------------------------------------------------------------------------------------|-------------------------|
| SQR    | Quadrat: $X^2$                                                                                                                   | 5 SQR → 25              |
| SQT    | Quadratwurzel: $\sqrt{X}$ ( $X < 0 \rightarrow \text{MATH ERROR}$ )                                                              | 9 SQT → 3               |
| $Y^X$  | Potenz (binär): $Y^X$ ( $Y = 0, X < 0 \rightarrow \text{DIV BY } 0$ ; negative Basis + nicht-ganzzahliger Exponent → MATH ERROR) | 2 ENTER 10 $Y^X$ → 1024 |
| $E^X$  | Eulersche Zahl potenziert: $e^X$ (Überlauf → MATH ERROR)                                                                         | 1 $E^X$ → 2.71828...    |

### Logarithmen:

| Befehl | Bedeutung                                                                            | Beispiel (UPN) |
|--------|--------------------------------------------------------------------------------------|----------------|
| LN     | Natürlicher Logarithmus: $\ln(X)$ ( $X \leq 0 \rightarrow \text{MATH ERROR}$ )       | E LN → 1       |
| LOG    | Dekadischer Logarithmus: $\log_{10}(X)$ ( $X \leq 0 \rightarrow \text{MATH ERROR}$ ) | 1000 LOG → 3   |

### Winkelfunktionen (Eingabe und Ergebnis im Winkelmodus RAD/DEG):

| Befehl | Bedeutung                                                | Beispiel (UPN)    |
|--------|----------------------------------------------------------|-------------------|
| SIN    | Sinus                                                    | DEG 30 SIN → 0.5  |
| COS    | Kosinus                                                  | DEG 60 COS → 0.5  |
| TAN    | Tangens                                                  | DEG 45 TAN → 1    |
| ASIN   | Arkussinus ( $  X   > 1 \rightarrow \text{MATH ERROR}$ ) | DEG 0.5 ASIN → 30 |

| Befehl | Bedeutung                                                                             | Beispiel (UPN)                |
|--------|---------------------------------------------------------------------------------------|-------------------------------|
| ACOS   | Arkuskosinus ( $\backslash\backslash X\backslash > 1 \rightarrow \text{MATH ERROR}$ ) | DEG 0.5 ACOS $\rightarrow$ 60 |
| ATAN   | Arkustangens                                                                          | DEG 1 ATAN $\rightarrow$ 45   |

#### Vorzeichen & Betrag:

| Befehl | Bedeutung                        | Beispiel (UPN)             |
|--------|----------------------------------|----------------------------|
| CHS    | Vorzeichen umkehren: $-X$        | 5 CHS $\rightarrow$ -5     |
| SGN    | Vorzeichen: -1, 0 oder +1        | 5 CHS SGN $\rightarrow$ -1 |
| ABS    | Betrag: $\backslash X\backslash$ | 5 CHS ABS $\rightarrow$ 5  |

#### Ganzzahl & Bruchteil:

| Befehl | Bedeutung                              | Beispiel (UPN)                |
|--------|----------------------------------------|-------------------------------|
| INT    | Ganzzahliger Anteil: $\text{trunc}(X)$ | 12.34 INT $\rightarrow$ 12    |
| FRAC   | Dezimalanteil: $X - \text{trunc}(X)$   | 12.34 FRAC $\rightarrow$ 0.34 |
| INC    | Um 1 erhöhen: $X + 1$                  | 5 INC $\rightarrow$ 6         |
| DEC    | Um 1 vermindern: $X - 1$               | 5 DEC $\rightarrow$ 4         |

INT/FRAC nutzen die Trunkation (Rundung Richtung 0): für negative Werte gilt daher z. B. -12.34 FRAC  $\rightarrow$  -0.34 (nicht 0.66).

#### Prozent:

| Befehl | Bedeutung                                                                                             | Beispiel (UPN)                     |
|--------|-------------------------------------------------------------------------------------------------------|------------------------------------|
| PCT    | Prozentzahl: $X / 100$                                                                                | 25 PCT $\rightarrow$ 0.25          |
| %      | Prozent (binär, HP-41CV): $Y \cdot X / 100$                                                           | 200 ENTER 10 % $\rightarrow$ 20    |
| %CH    | Prozentuale Änderung (binär, HP-41CV): $(X-Y) / Y \cdot 100$ ( $Y = 0 \rightarrow \text{DIV BY } 0$ ) | 100 ENTER 120 %CH $\rightarrow$ 20 |
| %12    | Durch 1200 teilen: $X / 1200$                                                                         | 12000 %12 $\rightarrow$ 10         |

#### Koordinatenumrechnung (wirken auf X und Y, Winkel im Winkelmodus RAD/DEG):

| Befehl | Bedeutung                                                                                                                     | Beispiel (UPN)                                                  |
|--------|-------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------|
| POLAR  | Rechtwinklig $\rightarrow$ polar: $X, Y = x, y \rightarrow X = \text{Länge}, Y = \text{Winkel}$                               | 3 ENTER 4 POLAR $\rightarrow X=5, Y=36.87^\circ$ (DEG)          |
| RECT   | Polar $\rightarrow$ rechtwinklig: $X = \text{Länge}, Y = \text{Winkel} \rightarrow X, Y = x, y$ –<br><b>Inverse von POLAR</b> | 36.87 ENTER 5 RECT $\rightarrow X \approx 4, Y \approx 3$ (DEG) |

POLAR und RECT sind Inversen: 3 ENTER 4 POLAR RECT  $\rightarrow X=4, Y=3$  (Round-Trip).

#### Uhrzeit (schalten die Anzeige auf FIX 4 um):

| Befehl | Bedeutung                                                                 | Beispiel (UPN)                     |
|--------|---------------------------------------------------------------------------|------------------------------------|
| HMS    | Uhrzeit: Dezimalstunden $\rightarrow$ H.MMSS (HP-41CV HMS $\rightarrow$ ) | 1.5055... HMS $\rightarrow$ 1.3020 |
| HR     | Uhrzeit: H.MMSS $\rightarrow$ Dezimalstunden (HP-41CV $\leftarrow$ HR)    | 1.3020 HR $\rightarrow$ 1.5055...  |

#### Konstanten (verhalten sich wie Zahleneingabe, respektieren SLON/SLOFF):

| Befehl | Bedeutung                       | Beispiel (UPN)              |
|--------|---------------------------------|-----------------------------|
| PI     | Kreiszahl $\pi = 3.14159...$    | PI $\rightarrow$ 3.14159... |
| E      | Eulersche Zahl $e = 2.71828...$ | E $\rightarrow$ 2.71828...  |

#### Speicher:

| Befehl  | Bedeutung                                  | Beispiel (UPN)             |
|---------|--------------------------------------------|----------------------------|
| CLM <n> | Löschen des Wertes im Speicherregister <n> | CLM 1 setzt Reg. 1 auf 0.0 |

#### Prozent- und Zeitfunktionen (HP-41CV)

1/X (Kehrwert): unär,  $X \leftarrow 1/X$  – wie INV; bei  $X = 0 \rightarrow \text{DIV BY } 0$ .

% (Prozent, binär):  $X \leftarrow Y \cdot X / 100$  („X Prozent von Y“). Y ist die Basis, X der Prozentsatz.

[ RUN] 200                    [ RUN] ENTER            [ RUN] 10                    [ RUN] %  
       => X = 20                (10 % von 200)

%CH (prozentuale Änderung, binär):  $X \leftarrow (X - Y)/Y \cdot 100$  – Änderung von Y (alt) nach X (neu); bei  $Y = 0 \rightarrow \text{DIV BY } 0$ .

[ RUN] 100                    [ RUN] ENTER            [ RUN] 120                    [ RUN] %CH  
       => X = 20                (von 100 auf 120: +20 %)

HMS (HP-41CV HMS→) und HR (HP-41CV ←HR) wandeln zwischen Dezimalstunden und der H.MMSS-Uhrzeit-Darstellung um – **beide** als gewöhnliche Dezimalzahlen (keine eigene Zeiteinheit):

| Befehl | Richtung                | Beispiel                                    |
|--------|-------------------------|---------------------------------------------|
| HMS    | Dezimalstunden → H.MMSS | 1.50555... HMS → 1.3020 (= 1 h 30 min 20 s) |
| HR     | H.MMSS → Dezimalstunden | 1.3020 HR → 1.50555...                      |

H.MMSS-Kodierung: Ganzzahlteil = Stunden, erste zwei Dezimalstellen = Minuten, letzte zwei = Sekunden. 1 h 30 min 20 s ist also 1.3020 (HMS) bzw.  $1 + 30/60 + 20/3600 = 1.50555...$  (dezimal/HR).

Beide schalten die Anzeige auf FIX 4 um, damit Minuten- und Sekundenstellen sichtbar bleiben (z. B. 1.3020 statt 1.302). Verfügbar im RUN- **und** PRGM-Modus.

## Befehle (Auswahl)

**Global:** MODE, PRGM, RUN, SLON/SLOFF (Stack-Lift), LIMIT ON/OFF (Step-Limit), FLOWCHART (ASCII-Flowchart – siehe unten), HELP, RESET, QUIT, SAVE <name>, LOAD <name>, PROGRAMS, FIX/SCI/ENG <n>, NORM

**RUN:** Zahlen, ENTER, + - \* /,  $Y^X$ , MOD, %, %CH, SQR SQT INV 1/X CHS SGN ABS INT FRAC INC DEC PCT %12, HMS, HR, LN LOG  $E^X$ , SIN COS TAN ASIN ACOS ATAN, RAD DEG, CHT RDN RUP  $X \leftrightarrow Y$  LASTX LASTY DUP DROP, PI E, STO <0-99>, RCL <0-99>, STO+ <0-99>, STO- <0-99>, STO\* <0-99>, STO/ <0-99>, S+ S- MEAN SDEV CLS (Statistik, Register 11–16), RND, RND <n> (Zufallszahlen, Schneiers Fortuna), BOARD <bcm>, BCM <board> (GPIO-Pin-Zuordnung, Raspberry Pi), SF, CF (Status-Flag setzen/löschen), CLX CLR CLSTK (X bzw. Stack löschen), EXEC, CONT, STEP (Schritt-Steuerung – siehe unten), XEQ <name> (Programm-Aufruf – siehe unten), TRACE [on|off], PAUSE (Pause, Zeit aus X – siehe unten), TIME/DATE (lokale Zeit bzw. Datum als Dezimalzahl – siehe unten), DATE+/DATE-/DDAYS (Datumsarithmetik, binär – siehe unten), ALMSET/ALMCAT/ALMDEL/ALMON/ALMOFF (Alarmer – siehe unten)

**PRGM:** jede Eingabe hängt ans Programm an, sofern sie kein Befehl ist: <Anweisung> (z. B. +, MOD, STO 1, LBL L, GTO L, CALL L, RTN, PSE, STOP, PAUSE, FIX 2, S+, S-, MEAN, SDEV, CLS, RND, TIME, DATE, DATE+, DATE-, DDAYS (Datumsarithmetik), BOARD <bcm>, BCM <board>), PRINT [Text] (Text ausgeben, mit Platzhaltern %x-%w/%<n>; ohne Text leere Zeile),  $X=0$  /  $X=Y$  /  $X<Y$  /  $X>Y$  /  $X>0$  /  $X<0$  (bedingte Tests), SF / CF / FS? / FS?C (Flag setzen/löschen/prüfen, siehe oben), <Zahl> (Konstante), ADD <Anweisung> (redundant, wird aus Gewohnheit akzeptiert), LIST, DEL <n> bzw. DEL <a>-<b> (Zeile n bzw. die Zeilen a–b löschen), DELETE <n> (wie DEL <n>), INS <n> (Einfüge-Modus vor Zeile n; ESC beendet), CLR/CLEAR (Programm leeren), COUNT (Anzahl Schritte), EXEC/GO/RUNP (Programm starten), CONT (angehaltenes Programm fortsetzen), STEP (Schritt-Steuerung, siehe unten), TRACE.

**Automatische Anzeige (LIST):** Jede Änderung des Programms und der Wechsel in den PRGM-Modus zeigt das Programm sofort mit Zeilennummern an:

| Aktion                                           | Effekt                                                                                           |
|--------------------------------------------------|--------------------------------------------------------------------------------------------------|
| PRGM (oder MODE → PRGM)                          | Programm wird angezeigt (LIST)                                                                   |
| INS <n> (Start des Einfüge-Modus)                | Programm wird angezeigt (LIST)                                                                   |
| jede eingefügte Programmzeile (im Einfüge-Modus) | Programm wird angezeigt (LIST)                                                                   |
| ESC (Einfüge-Modus beenden)                      | Programm wird angezeigt (LIST)                                                                   |
| DEL <n> bzw. DEL <a>-<b>                         | Programm wird <b>nach dem Löschen</b> angezeigt (LIST); beim Bereich erst nach der letzten Zeile |

Der **Einfüge-Punkt** bleibt dabei an seiner Stelle (rückt um die eingefügte Zeile nach, springt aber nicht ans Programmende); die automatische LIST verändert ihn nicht. Der Wechsel in den RUN-Modus (RUN bzw. MODE → RUN) zeigt das Programm bewusst **nicht** an.

### Register-Arithmetik (STO+ / STO- / STO\* / STO/)

Verrechnet den Wert im X-Register mit dem Speicherregister <n> und legt das Ergebnis **im Register** ab – der Stack bleibt dabei **unverändert** (Register ist die Basis, X der Operand, wie bei HP):

| Befehl   | Wirkung                                                                         |
|----------|---------------------------------------------------------------------------------|
| STO+ <n> | $\text{Reg}(n) = \text{Reg}(n) + X$                                             |
| STO- <n> | $\text{Reg}(n) = \text{Reg}(n) - X$                                             |
| STO* <n> | $\text{Reg}(n) = \text{Reg}(n) \times X$                                        |
| STO/ <n> | $\text{Reg}(n) = \text{Reg}(n) \div X$ ( $X = 0 \rightarrow \text{DIV BY } 0$ ) |

Beispiel: 1 STO+ 2 erhöht den Wert in Register 2 um 1. Verfügbar im RUN- und PRGM-Modus (Register 0–99).

### Stack-Manipulation (FORTH-artig: DUP / DROP)

| Befehl | Wirkung                                                                                       |
|--------|-----------------------------------------------------------------------------------------------|
| DUP    | X nach Y duplizieren; X bleibt stehen ( $x \rightarrow y, y \rightarrow z, z \rightarrow w$ ) |
| DROP   | X entfernen und abrollen: $y \rightarrow x, z \rightarrow y, w \rightarrow z, w = 0$          |

Beispiel:

```
[ RUN] 1          => X = 1
[ RUN] ENTER      => X = 0, Y = 1
[ RUN] 2          => X = 2, Y = 1
[ RUN] DUP        => X = 2, Y = 2, Z = 1
[ RUN] DROP       => X = 2, Y = 1, Z = 0 (die zweite 2 wird entfernt)
```

Verfügbar im RUN- und PRGM-Modus (wie CHT/RDN/RUP/ $X \leftrightarrow Y$ ).

### Fehlerwiederherstellung (LASTX / LASTY)

LASTX und LASTY holen den X- bzw. Y-Wert **vor der letzten Operation** (in last\_x/last\_y gemerkt) nach X – typischerweise zur Wiederherstellung nach einem Fehler:

| Befehl | Wirkung                                                 |
|--------|---------------------------------------------------------|
| LASTX  | $X = \text{last\_x}$ (X-Wert vor der letzten Operation) |
| LASTY  | $X = \text{last\_y}$ (Y-Wert vor der letzten Operation) |

- Beide respektieren den **Stack-Lift-Schalter** (SLON): bei SLON wandert der alte X-Wert vorher nach Y auf ( $x \rightarrow y, y \rightarrow z, z \rightarrow w$ ), danach wird der geholtte Wert in X geschrieben (wie Zahleneingabe/RCL/PI/E); bei SLOFF wird X einfach ersetzt.
- last\_x/last\_y bleiben dabei **unverändert**; verfügbar im RUN- und PRGM-Modus.

Beispiel:

```
[ RUN] 5          [ RUN] ENTER      [ RUN] 3          [ RUN] +  
      => X = 8      (last_x = 3, last_y = 5)  
[ RUN] LASTX      => X = 3      (alter X-Wert wiederhergestellt)  
[ RUN] LASTY      => X = 5      (alter Y-Wert wiederhergestellt)
```

## Statistik (S+ / S- / MEAN / SDEV / CLS)

Die statistischen Funktionen (HP-15C-artig) arbeiten auf den festen **Statistikregistern 11–16**:

| Register | Bedeutung                                      |
|----------|------------------------------------------------|
| 11       | Summe der X-Werte ( $\Sigma X$ )               |
| 12       | Summe der X-Quadrate ( $\Sigma X^2$ )          |
| 13       | Summe der Y-Werte ( $\Sigma Y$ )               |
| 14       | Summe der Y-Quadrate ( $\Sigma Y^2$ )          |
| 15       | Summe der Produkte $X \cdot Y$ ( $\Sigma XY$ ) |
| 16       | Anzahl der Werte $n$                           |

| Befehl | Wirkung                                                                                                                                                                                                                                                  |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| S+     | Akkumuliert das Datenpaar $(X, Y)$ : $R11 += X$ , $R12 += X^2$ , $R13 += Y$ , $R14 += Y^2$ , $R15 += X \cdot Y$ , $R16 += 1$ ; legt danach $n$ ( $R16$ ) auf den Stack ( $X \rightarrow Y$ , $Y \rightarrow Z$ , $Z \rightarrow W$ , $n \rightarrow X$ ) |
| S-     | Wie S+, aber <b>abziehend</b> ( $R16 -= 1$ ) – macht eine fehlerhafte S+-Eingabe rückgängig; legt danach $n$ auf den Stack                                                                                                                               |
| MEAN   | Mittelwert: $X = \Sigma X / n$ ( $\bar{x}$ ), $Y = \Sigma Y / n$ ( $\bar{y}$ ); $n = 0 \rightarrow \text{DIV BY } 0$                                                                                                                                     |
| SDEV   | Standardabweichung der Stichprobe (Teiler $n-1$ ): $X = s(x)$ , $Y = s(y)$ ; $n < 2 \rightarrow \text{DIV BY } 0$                                                                                                                                        |
| CLS    | Setzt die Statistikregister 11–16 auf 0 (Clear Summe); der Stack bleibt unverändert                                                                                                                                                                      |

Verfügbar im RUN- **und** PRGM-Modus. Die Statistikregister liegen im normalen Speicher und werden daher mit state.json persistiert und von RESET geleert.

Beispiel (RUN) – die Datenpaare sind  $(X, Y) = (10, 1), (20, 2), (30, 3)$ :

```
[ RUN] CLS          Statistikregister 11..16 leeren  
[ RUN] 1          [ RUN] ENTER      [ RUN] 10          [ RUN] S+  
      => X = 1 (n), Y = 10, Z = 1  
[ RUN] 2          [ RUN] ENTER      [ RUN] 20          [ RUN] S+  
      => X = 2 (n), Y = 20, Z = 2  
[ RUN] 3          [ RUN] ENTER      [ RUN] 30          [ RUN] S+  
      => X = 3 (n), Y = 30, Z = 3  
[ RUN] MEAN        => X = 20 ( $\bar{x} = (10+20+30)/3$ )  
                  Y = 2 ( $\bar{y} = (1+2+3)/3$ )  
[ RUN] SDEV        => X = 10 ( $s(x) = 10$ )  
                  Y = 1 ( $s(y) = 1$ )  
[ RUN] 3          [ RUN] ENTER      [ RUN] 30          [ RUN] S-  
      => X = 2 (n) (das dritte Datenpaar (30, 3) wird wieder entfernt)
```

## Zufallszahlen (RND, Schneiers Fortuna)

Zufallszahlen liefert der Generator **Fortuna** nach Schneier & Ferguson (*Applied Cryptography*, Kapitel 9) – reine Standardbibliothek, UI-frei (taschenrechner/fortuna.py):

- AES-256 im CTR-Modus mit **Forward-Sekret** (9.4),
- 32 Entropie-Pools mit Reseed-Regel und Rate-Limit (9.5; Entropie-Quelle standardmäßig `os.urandom`),
- 64-Byte-**Seed-Datei** für die Kontinuität über Neustarts (9.6),

- **Rejection-Sampling** für gleichverteilte ganze Zahlen (9.7).

| Befehl  | Wirkung in X                                              | Beispiel              |
|---------|-----------------------------------------------------------|-----------------------|
| RND     | gleichverteilter Zufallswert in [0, 1) (53-Bit-Präzision) | RND → z. B. 0.4021... |
| RND <n> | gleichverteilte <b>ganze</b> Zahl in {0, ..., n-1}        | RND 6 → 0...5         |

<n> muss ganzzahlig und  $\geq 1$  sein, sonst SYNTAX ERROR. Verfügbar im RUN- und PRGM-Modus (im PRGM-Modus eine normale Programmzeile, z. B. in Schleifen). RND respektiert den **Stack-Lift-Schalter** (SLON): bei SLON wandert der alte X-Wert nach Y, bevor der neue Wert in X geschrieben wird (wie Zahleneingabe/RCL/PI/E); bei SLOFF wird X ersetzt. Beispiel (Würfeln, Ergebnis 1...6):

```
[ RUN] 1          [ RUN] ENTER    [ RUN] RND 6          [ RUN] +
      => X = 1 + Zufallswert (0...5), also 1...6
```

**Seed-Datei:** Der Fortuna-Zustand ist **kein** Teil von state.json. Beim Start liest die CLI die Seed-Datei ~/.taschenrechner/seed.bin (64 Bytes) ein, reseedet den Generator mit ihrem Inhalt und schreibt 64 neue Zufallsbytes zurück (fehlt sie, werden 64 neue Bytes erzeugt und abgelegt). RESET ersetzt den Generator durch einen frischen; die Seed-Datei bleibt erhalten.

## Zeit/Datum (TIME / DATE)

Die Funktionen legen die **aktuelle lokale Zeit** bzw. das **aktuelle lokale Datum** als gewöhnliche Dezimalzahl in X ab (Implementierung: taschenrechner/zeit.py, reine Standardbibliothek datetime):

| Befehl | Format    | Kodierung                                                                                             | Beispiel                |
|--------|-----------|-------------------------------------------------------------------------------------------------------|-------------------------|
| TIME   | HH.MMSShh | Stunden ganzzahlig, dann je 2 Stellen: Minuten (0..59), Sekunden (0..59), Hundertstelsekunden (0..99) | 13.453025 = 13:45:30.25 |
| DATE   | DD.MMYYYY | Tag ganzzahlig, dann 2 Stellen Monat (01..12) und 4 Stellen Jahr                                      | 25.092026 = 25.09.2026  |

- Bei DATE belegt der Monat die Stellen 1–2 hinter dem Komma ( $\div 100$ ) und das Jahr die Stellen 3–6, also ab der **5. Nachkommastelle** ( $\div 1.000.000$  – nicht  $\div 10.000$ , sonst würden Monats- und Jahresziffern kollidieren:  $25 + 9/100 + 2026/1.000.000 = 25.092026$ ).
- Die Werte sind **gewöhnliche Floats** und damit auch **negativ** verwendbar (z. B. Zeitberechnungen: TIME ... CHS, Subtraktion zweier Zeitpunkte).
- Beide respektieren den **Stack-Lift-Schalter** (SLON): bei SLON wandert der alte X-Wert nach Y, bevor der neue Wert in X geschrieben wird (wie Zahleneingabe/RCL/PI/E); bei SLOFF wird X ersetzt.
- Verfügbar im RUN- und PRGM-Modus (argumentlos; TIME 5 → SYNTAX ERROR; im PRGM-Modus eine normale Programmzeile).

```
[ RUN] TIME          => X = z. B. 13.453025  (13:45:30.25, lokale Zeit)
[ RUN] DATE          => X = z. B. 25.092026  (25.09.2026, lokales Datum)
```

## Datumsarithmetik (DATE+ / DATE- / DDAYS)

Die Funktionen rechnen auf **Datumswerten** (DD.MMYYYY, wie DATE) – Tage addieren/abziehen bzw. die Tagesdifferenz bestimmen. Sie sind **binäre Operationen** (lesen Y und X, schreiben das Ergebnis nach X und rollen den Stack auf:  $z \rightarrow y$ ,  $w \rightarrow z$ ,  $w \leftarrow 0$ , wie +/- – der alte X- und Y-Wert sind Operanden und werden dabei verbraucht). Implementierung: taschenrechner/zeit.py (add\_days/sub\_days/diff\_days, reine Standardbibliothek datetime).

| Befehl | Bedeutung    | Eingabe (Stack)     | Ergebnis in X |
|--------|--------------|---------------------|---------------|
| DATE+  | Datum + Tage | Y = Datum, X = Tage | Y + X Tage    |

| Befehl | Bedeutung                             | Eingabe (Stack)     | Ergebnis in X            |
|--------|---------------------------------------|---------------------|--------------------------|
|        | (zukünftiges Datum)                   |                     |                          |
| DATE-  | Datum – Tage<br>(vergangenes Datum)   | Y = Datum, X = Tage | Y – X Tage               |
| DDAYS  | Anzahl der Tage<br>zwischen den Daten | X und Y = Daten     | \\ Y – X  (Tage, Betrag) |

- Tage dürfen auch Bruchteile sein (z. B. 30.5 = 30 Tage 12 h); der Tagesbruch wird beim Datumsergebnis auf volle Tage abgegrenzt.
- Ungültige Datumsangaben (z. B. Monat 13, Tag 32) liefern INVALID DATE.
- Verfügbar im RUN- **und** PRGM-Modus (argumentlos; die Werte kommen vom Stack; im PRGM-Modus eine normale Programmzeile).

```
[ RUN] 25.092026      [ RUN] ENTER      [ RUN] 30          [ RUN] DATE+
=> X = 25.102026    (25.09.2026 + 30 Tage = 25.10.2026)
```

```
[ RUN] 25.092026      [ RUN] ENTER      [ RUN] 30          [ RUN] DATE-
=> X = 26.082026    (25.09.2026 - 30 Tage = 26.08.2026)
```

```
[ RUN] 25.092026      [ RUN] ENTER      [ RUN] 25.102026  [ RUN] DDAYS
=> X = 30              (Tage zwischen 25.09.2026 und 25.10.2026)
```

## GPIO-Pin-Zuordnung (BOARD / BCM, Raspberry Pi)

Die Befehle wandeln zwischen dem **physikalischen Pin** an der 40-poligen Steckleiste („Board-Pin“, 1..40) und der **GPIO-Nummer** (BCM-Nummer, 0..27) um. Die Zuordnung ist fest in taschenrechner/gpio.py hinterlegt (Quelle: gpio-pinbelegung.md) und umfasst nur die GPIO-Leitungen – Stromversorgung (3V3/5V) und Masse sind nicht enthalten.

| Befehl    | Argument           | Ergebnis in X                    | Beispiel      |
|-----------|--------------------|----------------------------------|---------------|
| BOARD <n> | BCM-Nummer (0..27) | physikalischer Board-Pin (1..40) | BOARD 18 → 12 |
| BCM <n>   | Board-Pin (1..40)  | BCM-Nummer (0..27)               | BCM 12 → 18   |

- <n> muss eine (ganze) Zahl sein, sonst SYNTAX ERROR.
- Ein Pin ohne GPIO-Leitung (Stromversorgung: Board-Pin 1/17 = 3V3, 2/4 = 5V; Masse: 6/9/14/20/25/30/34/39) bzw. ein Wert außerhalb des gültigen Bereichs liefert INVALID PIN.
- Verfügbar im RUN- **und** PRGM-Modus (im PRGM-Modus eine normale Programmzeile). Wie Zahl-/Konstanten-/Register-Eingabe respektiert der Befehl den Stack-Lift-Schalter (SLON).
- Die Zuordnung ist reine Standardbibliothek (kein GPIO-Hardwarezugriff) und läuft daher auch auf X86; spätere, hardwarenahe GPIO-Befehle bauen auf dem Modul taschenrechner/gpio.py auf.

Beispiel (RUN) – BCM-Nummer in den physikalischen Pin umrechnen:

```
[ RUN] BOARD 18      => X = 12    (BCM 18 liegt an Board-Pin 12)
[ RUN] BCM 12        => X = 18    (Board-Pin 12 ist BCM 18)
[ RUN] BCM 4         => INVALID PIN (Board-Pin 4 = 5V, kein GPIO)
[ RUN] BCM 1         => INVALID PIN (Board-Pin 1 = 3V3, kein GPIO)
```

## GPIO-Geräte (GPIO <Befehl>, per gpiozero)

Mit GPIO werden **Eingabe-, Ausgabe- und Sensor-Geräte** über die Bibliothek [gpiozero](#) gesteuert (LED, PWMLED, BUZZER, TONALBUZZER, MOTOR, RGBLED, PIR, LDR, HCSR04, BUTTON). gpiozero **wird lazy importiert**: Der Rechner läuft auch auf Maschinen ohne GPIO und ohne gpiozero (dort wirft jeder GPIO-Befehl GPIO NOT AVAILABLE). Auf einem Raspberry Pi mit installiertem gpiozero greifen die Befehle direkt auf die Hardware zu.

Die Befehle beginnen mit dem Präfix GPIO (dazu der Unterbefehl), damit sie nicht mit anderen Taschenrechner-Funktionen kollidieren.

**Boolesche Werte, Zustände und Pegel.** Alle Zustände, Pegel und Wahrheitswerte laufen als **Ziffern** über den Stack:

| Symbol    | Ziffer | Bedeutung                 |
|-----------|--------|---------------------------|
| TRUE      | 1      | wahr                      |
| FALSE     | 0      | falsch                    |
| AN / HIGH | 1      | ein / Hochpegel           |
| AUS / LOW | 0      | aus / Tiefpegel           |
| TOGGLE    | 2      | dritter Zustand: umkehren |

Weitere „dritte Zustände“ (wenn ein Gerät mehr als zwei Zustände kennt) würden fortgesetzt mit 3, 4, ... nummeriert.

**Parameterübergabe.** Der **Stack ist der Hauptweg** der Parameterübergabe und der Ergebnisausgabe:

| Fall                 | Herkunft / Ablage                           |
|----------------------|---------------------------------------------|
| 1 Parameter          | Stack X                                     |
| 2 Parameter          | Stack X (erster), Y (zweiter)               |
| 3 Parameter          | Stack X, Y, Z                               |
| 4 Parameter          | Stack X, Y, Z, W                            |
| mehr als 4 Parameter | Speicherregister 0–99 (in der Befehls-Doku) |

Ergebnisse werden **unter X** abgelegt (bei mehreren Werten in X, Y, Z, W); das respektiert den **Stack-Lift-Schalter** (SLON/SLOFF) wie Zahl- und Register-Eingabe. Ein ausgelesenes value landet in X; ein ausgelesenes Flag/Zustand als 0 oder 1 unter X.

**Gerätetypen und benötigte Pins** (alle BCM-Nummern, auf dem Stack):

| Typ         | Pins (Stack)       | gpiozero-Klasse |
|-------------|--------------------|-----------------|
| LED         | X                  | LED             |
| PWMLED      | X                  | PWMLED          |
| BUZZER      | X                  | Buzzer          |
| TONALBUZZER | X                  | TonalBuzzer     |
| MOTOR       | X (vor), Y (rück)  | Motor           |
| RGBLED      | X,Y,Z = r,g,b      | RGBLED          |
| PIR         | X                  | MotionSensor    |
| LDR         | X                  | LightSensor     |
| HCSR04      | X (Trig), Y (Echo) | DistanceSensor  |
| BUTTON      | X                  | Button          |

**Befehle** (alle mit dem Präfix GPIO):

| Befehl              | Parameter (Stack)                      | Ergebnis / Wirkung                                        |
|---------------------|----------------------------------------|-----------------------------------------------------------|
| GPIO CREATE <Typ>   | Pin(s) auf Stack (s. Typ-Tabelle)      | neue <b>Geräte-ID</b> in X; Gerät wird „aktuell“          |
| GPIO SEL            | Geräte-ID in X                         | dieses Gerät wird „aktuell“                               |
| GPIO AN / GPIO HIGH | –                                      | Gerät an (Pegel 1)                                        |
| GPIO AUS / GPIO LOW | –                                      | Gerät aus (Pegel 0)                                       |
| GPIO TOGGLE         | –                                      | Zustand umkehren (2)                                      |
| GPIO WERT           | –                                      | value auslesen → X (RGBLED: X,Y,Z = r,g,b)                |
| GPIO ZUSTAND        | –                                      | boolescher Zustand 0/1 auslesen → X                       |
| GPIO SET            | value in X (RGBLED: X,Y,Z)             | value schreiben (digital 0/1, PWM 0.0–1.0, Motor -1... 1) |
| GPIO BLINK          | on_time=X, off_time=Y, n=Z (n≤0: ewig) | Blinken                                                   |

| Befehl             | Parameter (Stack)                         | Ergebnis / Wirkung                                               |
|--------------------|-------------------------------------------|------------------------------------------------------------------|
| GPIO VOR           | Geschwindigkeit in X (0..1)               | Motor vorwärts                                                   |
| GPIO RUECK         | Geschwindigkeit in X (0..1)               | Motor rückwärts                                                  |
| GPIO STOP          | –                                         | Motor anhalten / TonalBuzzer stumm                               |
| GPIO REVERSE       | –                                         | Motor-Drehrichtung umkehren                                      |
| GPIO TON [Ton]     | Ton optional (Name z. B. A4, Hertz, MIDI) | Ton abspielen (TonalBuzzer)                                      |
| GPIO BEAM          | –                                         | Buzzer im Takt (beep)                                            |
| GPIO WHEN <Event>  | – (Programmzeile)                         | Event eingetreten? → nächste Zeile ausführen, sonst überspringen |
| GPIO WARTEN <Cond> | – (Programmzeile)                         | blockierend warten, bis Bedingung erfüllt                        |
| GPIO CLOSE         | –                                         | aktuelles Gerät schließen (Pins freigeben)                       |
| GPIO CLOSEALL      | –                                         | alle Geräte schließen                                            |

**Töne (GPIO TON, TONALBUZZER).** TON akzeptiert drei Arten von Argumenten: einen **namentlichen Ton** (z. B. A4), eine **Frequenz in Hz** (z. B. 440) oder eine **MIDI-Notenzahl** (0–127; 69 = A4).

*Namentlicher Ton* = Tonbuchstabe A–G + optionaler Halbton # (scharf) oder b (flat) + Oktav-Ziffer 0–9. Beispiel: C#4, Eb4, Bb4. Halbtöne sind enharmonisch austauschbar: C#4 = Db4 (beide 277,18 Hz), F#4 = Gb4, A#4 = Bb4.

*Spielbarer Bereich.* Das TONALBUZZER-Gerät wird mit mid\_tone=A4 und octaves=4 angelegt (gpio.py:create), d. h. es spielt von A0 (**27,5 Hz**) bis A8 (**7040 Hz**). Töne außerhalb dieses Bereichs werfen GPIO ERROR (gpiozero: *tone is out of the device's range*).

*Die zwölf Töne der chromatischen Tonleiter* (jeder in jeder Oktave 0–8; Referenzfrequenzen in Oktave 4):

| Ton      | C     | C#    | D     | D#    | E     | F     | F#    | G     | G#    | A     | A#    | B     |
|----------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| Hz       | 261,6 | 277,1 | 293,6 | 311,1 | 329,6 | 349,2 | 369,9 | 392,0 | 415,3 | 440,0 | 466,1 | 493,8 |
| (Okt. 4) | 3     | 8     | 6     | 3     | 3     | 3     | 9     | 0     | 0     | 0     | 6     | 8     |

Jede Oktave verschiebt die Frequenz um den Faktor 2 (eine Oktave höher =  $\times 2$ , eine tiefer =  $\div 2$ ). Beispielwerte: A0=27,5 · A1=55 · A2=110 · A3=220 · A4=440 · A5=880 · A6=1760 · A7=3520 · A8=7040 Hz.

**Events und Bedingungen** (Vollständige Liste; WHEN testet, WARTEN wartet):

| Event / Bedingung | Gerät(e)                | Bedeutung                    |
|-------------------|-------------------------|------------------------------|
| PRESSED           | BUTTON                  | gedrückt                     |
| RELEASED          | BUTTON                  | losgelassen                  |
| HELD              | BUTTON                  | gedrückt gehalten            |
| MOTION            | PIR (MotionSensor)      | Bewegung erkannt             |
| NO MOTION         | PIR (MotionSensor)      | keine Bewegung               |
| LIGHT             | LDR (LightSensor)       | hell                         |
| DARK              | LDR (LightSensor)       | dunkel                       |
| IN_RANGE          | HCSR04 (DistanceSensor) | im Messbereich               |
| OUT_OF_RANGE      | HCSR04 (DistanceSensor) | außerhalb des Messbereichs   |
| ACTIVATED         | (Eingabe allgemein)     | aktiviert (niedrig → hoch)   |
| DEACTIVATED       | (Eingabe allgemein)     | deaktiviert (hoch → niedrig) |

**Event-Handler über XEQ.** WHEN <Event> steuert die **nächste** Programmzeile: ist das Event eingetreten, wird sie ausgeführt; ist es nicht eingetreten, wird sie übersprungen. Der typische Aufbau ist WHEN gefolgt von einem XEQ, das

den Handler aufruft – die XEQ-Zeile programmiert der Benutzer selbst:

```
# Zeile 1: Event testen; Zeile 2: Handler aufrufen (benutzerdefiniert)
GPIO WHEN PRESSED
XEQ BUTTON1PRESSED
RTN
LBL BUTTON1PRESSED
    GPIO AN          # LED an (Beispiel-Handler)
    RTN
```

Das Event-Flag wird bei der Prüfung geleert (Kanten-Trigger): jedes Eintreten löst den Handler genau einmal aus. WHEN und WARTEN sind **Programmzeilen** (PROGRAM ONLY im RUN-Modus). WARTEN <Cond> blockiert solange, bis die Bedingung erfüllt ist; der Benutzer bricht das Warten mit Strg+C ab.

**Beispiele (RUN-Modus).** Jede Zeile ist **ein** kompletter Befehl (ENTER ist ein eigener Befehl, der X nach Y absetzt – nicht Teil der Zahleneingabe):

```
# LED an Pin 17
17          # Pin (BCM 17) in X
GPIO CREATE LED  # LED erzeugen -> X = Geräte-ID
GPIO AN        # LED an
GPIO WERT      # value auslesen -> X = 1
GPIO AUS       # LED aus
GPIO CLOSE     # Pin freigeben

# PWM-LED an Pin 18, 50 % Helligkeit
18          # Pin (BCM 18) in X
GPIO CREATE PWMLD  # PWMLD erzeugen -> X = Geräte-ID
0.5         # 50 % Helligkeit (value 0.0..1.0) in X
GPIO SET       # Helligkeit setzen
GPIO WERT      # auslesen -> X = 0.5
GPIO CLOSE

# Bewegungsmelder (PIR) an Pin 4
4           # Pin (BCM 4) in X
GPIO CREATE PIR   # PIR erzeugen -> X = Geräte-ID
GPIO ZUSTAND     # 1 = Bewegung erkannt, 0 = keine -> X
GPIO WERT        # gemittelter Wert 0..1 -> X
GPIO CLOSE

# Buzzer an Pin 3 (klingt, bis AUS)
3           # Pin (BCM 3) in X
GPIO CREATE BUZZER # Buzzer erzeugen -> X = Geräte-ID
GPIO AN         # Buzzer an
GPIO AUS        # Buzzer aus
GPIO CLOSE

# Abstandssensor (HCSR04): Trigger an Pin 17, Echo an Pin 18
18          # Echo-Pin (BCM 18) zuerst -> Y
ENTER
17          # Trigger-Pin (BCM 17) dann -> X
GPIO CREATE HCSR04 # HCSR04 erzeugen (X=Trig, Y=Echo) -> X = Geräte-ID
GPIO WERT        # Distanz 0..1 (max_distance-Default 1 m -> X in m)
GPIO CLOSE
```

**Beispiele (Programm / PRGM-Modus).** Zeitliche Effekte (BLINK, BEAM, VOR, ...) laufen im Hintergrund und dauern länger als das Programm – ein zu schnelles CLOSE/AUS/RTN macht den Effekt unsichtbar. Deshalb nach jeder solchen

Wirkung eine PSE (Pause, 1 s). WARTEN/WHEN sind nur im Programm verfügbar (PROGRAM ONLY).

```
# Buzzer: 3× 0,1 s blinken
# -- Programm (PRGM-Modus) --
RCL 0
GPIO CREATE BUZZER    # Buzzer erzeugen -> X = Geräte-ID
3                    # n = 3 Takte
ENTER
0.1                  # off_time (s)
ENTER
0.1                  # on_time (s)
GPIO BLINK            # 3× 0,1 s ein/aus (Hintergrund-Thread)
PSE                   # ~1 s warten, bis das Blinken verklingt
GPIO CLOSE
RTN
# -- Start (RUN-Modus) --
16                   # Buzzer-Pin (BCM 16)
STO 0
EXEC
```

```
# Motor: vorwärts, dann anhalten
# -- Programm (PRGM-Modus) --
RCL 1
ENTER
RCL 0
GPIO CREATE MOTOR     # Motor erzeugen (X=vorwärts, Y=rückwärts)
1                     # Geschwindigkeit 1.0 (Vollgas)
GPIO VOR              # vorwärts
PSE                   # ~1 s laufen lassen
GPIO STOP             # anhalten
GPIO CLOSE
RTN
# -- Start (RUN-Modus) --
20                   # vorwärts-Pin (BCM 20)
STO 0
21                   # rückwärts-Pin (BCM 21)
STO 1
EXEC
```

```
# PIR: blockierend auf Bewegung warten, dann Buzzer
# -- Programm (PRGM-Modus) --
RCL 0
GPIO CREATE PIR       # PIR erzeugen (aktuell) -> X = Geräte-ID
GPIO WARTEN MOTION    # blockierend, bis Bewegung erkannt (Strg+C bricht ab)
RCL 1
GPIO CREATE BUZZER    # Buzzer erzeugen (aktuell)
GPIO AN
PSE                   # ~1 s klingen lassen
GPIO AUS
GPIO CLOSEALL
RTN
# -- Start (RUN-Modus) --
4                     # PIR-Pin (BCM 4)
STO 0
16                   # Buzzer-Pin (BCM 16)
STO 1
EXEC
```

# PIR: ereignisgesteuert (Kanten-Trigger), jede Bewegung -> Ausgabe

```
# -- Programm (PRGM-Modus) --
RCL 0
GPIO CREATE PIR      # PIR erzeugen (aktuell)
LBL L
GPIO WHEN MOTION     # Bewegung seit letzter Prüfung? (sonst: nächste Zeile überspring)
PRINT "Bewegung!"    # läuft nur bei Bewegung
GTO L                # (Schleife mit Strg+C beenden)
# -- Start (RUN-Modus) --
4                    # PIR-Pin (BCM 4)
STO 0
EXEC
```

**Fehler:** GPIO NOT AVAILABLE (kein gpiozero), NO DEVICE (kein aktuelles Gerät, z. B. nach CLOSEALL), INVALID DEVICE (unbekannter Typ), INVALID DEVICE TYPE (Befehl passt nicht zum Gerät, z. B. VOR bei einer LED), GPIO ERROR (Hardware-/gpiozero-Fehler), SYNTAX ERROR (falscher Unterbefehl), PROGRAM ONLY (WHEN/WARTEN im RUN-Modus).

## Bedingte Tests (wie HP 9100A)

$X=0$ ,  $X=Y$ ,  $X<Y$ ,  $X>Y$ ,  $X>0$ ,  $X<0$  prüfen die Stack-Register X (bzw. X und Y) und steuern die **nächste** Programmanweisung:

| Test  | Bedingung (wahr, wenn)   |
|-------|--------------------------|
| $X=0$ | $X = 0$                  |
| $X=Y$ | $X = Y$                  |
| $X<Y$ | $X < Y$                  |
| $X>Y$ | $X > Y$                  |
| $X>0$ | $X > 0$ (streng positiv) |
| $X<0$ | $X < 0$ (streng negativ) |

- Bedingung **wahr** → nächste Anweisung wird ausgeführt.
- Bedingung **falsch** → nächste Anweisung wird **übersprungen**, das Programm fährt mit der übernächsten Anweisung fort.

Der Stack wird dabei nicht verändert. Die Tests sind Programmanweisungen (PROGRAM ONLY – im RUN-Modus: PROGRAM ONLY) und laufen in allen Frames (EXEC/STEP/CALL/XEQ). Beispiel (Maximum von a in Reg. 0 und b in Reg. 1):

```
[PRGM] RCL 0      [PRGM] ENTER      [PRGM] RCL 1
[PRGM] X<Y        [PRGM] X<>Y       [PRGM] STO 0      [PRGM] RTN
```

Ist  $b < a$  ( $X<Y$  wahr), wird  $X<>Y$  ausgeführt und a (das Größere) liegt in X; ist  $b \geq a$ , wird  $X<>Y$  übersprungen und b bleibt in X.

$X>0/X<0$  sind die Vorzeichen-Tests (streng:  $X = 0$  ist bei beiden **falsch**). Beispiel (Betrag: bei negativem X Vorzeichen umkehren):

```
[PRGM] X<0      [PRGM] CHS      [PRGM] RTN
```

## Flag-Befehle (SF / CF / FS? / FS?C)

Ein einzelnes **Status-Flag** (wie bei HP) steuert eine konditionale Verzweigung. Das Flag ist ein globaler Zustand (nicht stack-/registerbezogen) und wird wie der Stack-Lift-Schalter **persistiert** (state.json) und von RESET gelöscht. In der Status-Anzeige (Prompt) ist der Zustand immer sichtbar als FL=0 (gelöscht) bzw. FL=1 (gesetzt), z. B. [RUN FL=1 SL].

| Befehl | Wirkung                                                                                                                                   | Modus               |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------|---------------------|
| SF     | <b>Set Flag:</b> setzt das Flag (Flag = 1, FL=1)                                                                                          | RUN und PRGM        |
| CF     | <b>Clear Flag:</b> löscht das Flag (Flag = 0, FL=0)                                                                                       | RUN und PRGM        |
| FS?    | <b>Flag Status?:</b> überspringt die nächste Programmzeile, wenn das Flag gelöscht ist; ist es gesetzt, wird die nächste Zeile ausgeführt | PRGM (PROGRAM ONLY) |
| FS?C   | wie FS? (gleiche Übersprungs-Semantik), <b>löscht das Flag zusätzlich nach dem Test</b> (das C steht für Clear)                           | PRGM (PROGRAM ONLY) |

SF und CF sind im **RUN- und PRGM-Modus** verfügbar – typischer Einsatz nach einem STOP: Das Programm hält an, man setzt/löscht im RUN-Modus das Flag (z. B. SF) und fährt mit CONT fort; ein darauffolgendes FS? im Programm sieht dann den aktuellen Flag-Zustand. FS?/FS?C sind Programmzeilen (PROGRAM ONLY), weil sie die nächste Programmzeile steuern. FS?C unterscheidet sich von FS? nur durch den Zusatzeffekt: Nach dem Test wird das Flag **gelöscht** – z. B. damit eine FS?C-Schleife beendet wird, sobald das Flag weg ist (FS? allein würde bei gesetztem Flag in eine Endlosschleife laufen). Der Stack wird von allen vier Befehlen nicht verändert.

Beispiel (Programm: bei gesetztem Flag Vorzeichen von X umkehren, sonst unverändert):

```
[PRGM] FS?      [PRGM] CHS      [PRGM] RTN
[ RUN] 5        [ RUN] SF       [ RUN] EXEC    => X = -5   (Flag gesetzt: CHS ausg
[ RUN] 5        [ RUN] CF       [ RUN] EXEC    => X = 5    (Flag gelöscht: CHS übe
```

FS?C-Beispiel (Schleife, solange das Flag gesetzt ist – FS?C löscht es nach dem Test, dadurch endet die Schleife):

```
[PRGM] LBL L      [PRGM] FS?C      [PRGM] GTO L
[PRGM] CHS        [PRGM] RTN
[ RUN] 5          [ RUN] SF       [ RUN] EXEC    => X = -5   (Schleife endet, Flag i
```

## Unterprogramme (CALL / RTN)

CALL <label> ruft ein Unterprogramm auf: Der Rechner springt zur mit LBL <label> markierten Stelle und führt die Anweisungen dort aus. Das RTN am Ende des Unterprogramms springt zurück zu der Anweisung, die dem CALL direkt folgt. Unterprogramme arbeiten auf dem **gleichen Stack** wie der Aufrufer (keine Kopie). Aufrufe können verschachtelt werden (max. **16** Stufen; darüber DEPTH ERROR). Ein RTN **ohne** vorheriges CALL beendet die Programmausführung, wie bisher.

Beispiel (Wurzel ziehen, Ergebnis anschließend verdoppeln):

```
[PRGM] 9          [PRGM] CALL SQRT  [PRGM] 2          [PRGM] *
[PRGM] RTN        [PRGM] LBL SQRT   [PRGM] SQT        [PRGM] RTN
[ RUN] EXEC       => X = 6
```

## Programm-Aufruf (XEQ <name>)

XEQ <name> ruft ein **Programm** auf – entweder ein LBL-Label im aktuellen Programmspeicher (PRGM) oder ein gespeichertes Programm aus ~/.taschenrechner/Programme/ (<name>.prg/<name>.txt; die Endung darf weggelassen werden, .prg hat Vorrang). Das aufgerufene Programm läuft in **eigenem Speicher** (eigener Frame), sodass der PRGM-Speicher beim Aufruf einer Datei **unverändert** bleibt, arbeitet aber auf dem **gleichen Stack** und den **gleichen Registern** wie der Aufrufer – dort werden Parameter und Ergebnis

ausgetauscht. Das Programm endet mit RTN oder seinem Ende; die Ausführung setzt an der Stelle **nach dem** XEQ fort.

XEQ ist im **RUN-Modus** ein Sofortbefehl (direkter Aufruf) und im **PRGM-Modus** eine normale Programmzeile (wird beim Start ausgeführt). Aufrufe sind verschachtelbar (max. **16** Stufen; darüber DEPTH ERROR). Nicht auffindbares Ziel: LABEL ERROR.

Beispiel (inc.txt in Programme/ erhöht X um 1):

Programme/inc.txt:

```
LBL INC
ENTER
1
+
RTN
```

[ RUN] 5                   => X = 5

[ RUN] XEQ inc           => X = 6           (INC: ENTER 1 +; Ergebnis in X, Y = 5)

## Programm anhalten und fortsetzen (STOP / CONT)

STOP (Programmzeile) hält ein laufendes Programm an und legt den Rechner in den **RUN-Modus**: der Benutzer kann jetzt beliebig weiterarbeiten – rechnen, Werte auf dem Stack ablegen, Register speichern oder abrufen. CONT setzt das Programm mit der Anweisung **direkt nach der STOP-Anweisung** fort.

```
[PRGM] RCL 1           [PRGM] STO 0           [PRGM] STOP       <- hält hier an
[PRGM] RCL 0           [PRGM] SQRT           [PRGM] RTN
[ RUN] EXEC           => Programm läuft bis STOP und hält an
[ RUN] 5              [ RUN] ENTER       [ RUN] 3           [ RUN] +
                     => X = 8            (Benutzerarbeit während des Anhaltens)
[ RUN] CONT           => Programm setzt nach STOP fort
```

- STOP wirkt in jeder Programmzeile (auch in CALL/XEQ-Frames) und ist nur als Programmzeile verfügbar (im RUN-Modus: PROGRAM ONLY).
- CONT ist ein Sofortbefehl (RUN- und PRGM-Modus); ohne angehaltenes Programm: NOT STOPPED.
- Das Anhalten überlebt **keine** Neustart-/Zustandsänderung: ein neuer EXEC/XEQ-Lauf oder RESET verwirft das angehaltene Programm.

## Steuerung:

- Strg+C während eines laufenden Programms: **Programm stoppen** (Rechner läuft weiter).
- STOP im Programm: **Programm anhalten** (Übergabe an den RUN-Modus); CONT setzt fort (siehe oben).
- PSE im Programm: zeigt den **Stack** an und pausiert **1 Sekunde** (kein Enter nötig).
- INPUT <n> im Programm: bittet um eine Zahl (Eingabe), speichert in Register n.
- PRINT [Text] im Programm: gibt den Text aus; **ohne Text** wird eine leere Zeile ausgegeben (nur Zeilenvorschub). PRINT ist nur im PRGM-Modus verfügbar (im RUN-Modus: PROGRAM ONLY).

**PRINT-Platzhalter** (Stack- und Registerwerte im Text anzeigen):

| Platzhalter | Bedeutung                                                     |
|-------------|---------------------------------------------------------------|
| %X-%W       | Wert des Stack-Registers X/Y/Z/W (groß/klein egal)            |
| %<n>        | Wert des Speicherregisters <n> (0–99, eine oder zwei Ziffern) |
| %%          | ein wörtliches %                                              |

Die Werte werden im **aktuellen Anzeigeformat** formatiert (NORM/FIX n/ SCI n/ ENG n) – die Ausgabe entspricht exakt dem, was die Anzeige zeigt. Nicht erkannte %-Sequenzen (z. B. %q) bleiben unverändert.

```
[PRGM] PRINT "Die Temperatur ist %x Grad."
[ RUN] 21.5
[ RUN] FIX 1
[ RUN] EXEC
      Die Temperatur ist 21.5 Grad.
```

## Pause (PAUSE)

PAUSE wartet die in X stehende Zeit – wie sleep in der Shell, aber **ohne jegliche Ausgabe** (kein Stack, kein Text; im Gegensatz zu PSE). Die Zeit ist in **Dezimalstunden**: vor dem Komma die Stunden, nach dem Komma der Bruchteil einer Stunde (1.5 = 1 h 30 min). Auch **Bruchteile einer Sekunde** sind gültig (z. B. 0.25 ENTER 3600 / PAUSE = 0,25 s warten). Der Stack bleibt unverändert.

| Punkt   | Detail                                                                              |
|---------|-------------------------------------------------------------------------------------|
| Zeit    | in X (Dezimalstunden; berechenbar, z. B. 1.3020 HR PAUSE)                           |
| Wirkung | wartet X × 3600 Sekunden; ohne Ausgabe; Stack bleibt unverändert                    |
| Fehler  | negative bzw. nicht-endliche Zeit in X → TIME ERROR                                 |
| Modi    | RUN (Sofortbefehl) und PRGM (Programmzeile); argumentlos (PAUSE 1.5 → SYNTAX ERROR) |

```
[ RUN] 1.5 PAUSE           => wartet 1 h 30 min, danach X = 1.5
[ RUN] 1.3020 HR PAUSE    => wartet 1 h 30 min 20 s (X = 1.5055...)
[ RUN] 0.25 ENTER 3600 / PAUSE => wartet 0,25 s (Bruchteil einer Sekunde)
```

## Step (STEP [n])

STEP [n] lässt das Programm **anweisungsweise** (Schritt für Schritt) ablaufen: Jeder STEP-Befehl führt **genau eine** Anweisung aus und zeigt dabei – im selben Tabellen-Aufbau wie TRACE (siehe unten) – den Befehl (zeilennummeriert) sowie die Stack-Register X, Y, Z, W an, und hält anschließend an. Damit kann man ein Programm ohne TRACE-Durchlauf Schritt für Schritt steuern und den Zustand nach jedem Schritt prüfen.

| Punkt        | Detail                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| STEP         | führt einen Schritt aus – <b>setzt die STEP-Durchlauf-Fortsetzung fort</b> , wenn eine aktiv ist (siehe „Fortsetzung“); ist keine aktiv, beginnt die Durchlauf-Steuerung bei <b>Zeile 1</b>                                                                                                                                                                                                |
| STEP n       | beginnt die Schritt-Steuerung bei <b>Zeile n</b> (1-basiert)                                                                                                                                                                                                                                                                                                                               |
| Fortsetzung  | <b>Konsequente STEP-Befehle ohne Nummer</b> (STEP STEP STEP ...) setzen die Ausführung gemäß der programmierten Befehlsfolge fort – auch über CALL/RTN-Frames und bedingte Übersprünge (X=0, FS?, ...)                                                                                                                                                                                     |
| Abbruch      | <b>Jeder andere Befehl</b> (RUN-Befehl, Zahleneingabe, EXEC/GO/RUNP, CONT, XEQ, LIST, COUNT, TRACE, ...), ein <b>Moduswechsel</b> (MODE/PRGM/RUN), eine <b>Programm-Änderung</b> (INS/DEL/CLR/CLEAR/neue Programmzeile) oder ein <b>Rechnerfehler</b> während des Steps beendet die Fortsetzung – der nächste STEP ohne Nummer beginnt dann wieder bei <b>Zeile 1</b> , STEP n bei Zeile n |
| Programmende | Ist das Programm beim STEP zu Ende, beginnt ein STEP ohne Nummer wieder bei <b>Zeile 1</b> (statt NO PROGRAM)                                                                                                                                                                                                                                                                              |
| Neustart     | Die STEP-Position ist <b>nicht persistent</b> (kein Teil von state.json); RESET setzt sie zurück – beim Starten des Rechners zeigt STEP immer auf <b>Schritt 1</b> (sofern das Programm nicht leer ist)                                                                                                                                                                                    |

| Punkt | Detail                                                                                                                                                                                                     |
|-------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| STOP  | Ein STEP auf einer STOP-Anweisung hält wie EXEC an (Übergabe an den RUN-Modus, CONT setzt fort); der STEP-Modus bleibt dabei <b>aktiv</b> , ein folgender STEP ohne Nummer setzt <b>nach dem STOP</b> fort |
| Modi  | RUN <b>und</b> PRGM – ein Sofortbefehl in beiden (keine Programmzeile, wie EXEC/CONT)                                                                                                                      |

**Fehler:** NO PROGRAM (leeres Programm), SYNTAX ERROR (z. B. STEP 0, STEP 1.5, STEP abc, STEP 2 3), INDEX ERROR (Zeile n > Programm-Länge).

Beispiel (Programm 1 ENTER 2 +; jede STEP-Zeile zeigt Befehl + Stack an):

```
[PRGM] 1          [PRGM] ENTER          [PRGM] 2          [PRGM] +
[ RUN] STEP 1      => Zeile 1: 1          (X wird zu 1)
[ RUN] STEP        => Zeile 2: ENTER      (X nach Y abgesetzt)
[ RUN] STEP        => Zeile 3: 2          (X wird zu 2)
[ RUN] STEP        => Zeile 4: +          (X = 1 + 2 = 3)
[ RUN] STEP        => Programmende -> beginnt wieder bei Zeile 1
```

## Trace (TRACE)

TRACE schaltet die schrittweise Anzeige der Programmausführung ein (TRACE off schaltet sie wieder aus). Zu Beginn eines getraceten Laufs steht eine **Überschrift**, die die Stack-Register benennt, darunter eine Trennlinie; jede weitere Zeile zeigt den Befehl (Zeilennummer + Anweisung) und die aktuellen Werte der Register X, Y, Z, W (der Zustand **vor** der ausgeführten Anweisung):

| Command     | X | Y | Z |  |
|-------------|---|---|---|--|
| >> 1: RCL 0 | 0 | 0 | 0 |  |
| >> 2: ENTER | 5 | 0 | 0 |  |
| >> 3: RCL 1 | 0 | 5 | 0 |  |
| >> 4: +     | 3 | 5 | 0 |  |
| >> 5: RTN   | 8 | 0 | 0 |  |

Die Spaltenbreiten werden **aus der Bildschirmbreite berechnet** (die Stack-Spalten beginnen ab ca. Spalte 30, der Rest wird auf die vier Spalten verteilt), sodass alle Trennstriche | genau übereinanderstehen – unabhängig davon, ob eine Zahl kürzer oder länger ist. Die Breite ist nach oben durch die größte darstellbare Zahl begrenzt und nach unten abgegrenzt, damit die Spalten auch auf sehr schmalen oder sehr breiten Fenstern sinnvoll bleiben.

- Ist eine **Zahl** breiter als ihre Spalte (inkl. 2 Zeichen Rand), wird sie gekürzt; drei Punkte markieren den Wegfall (z. B. wird 4711081523,55 gekürzt dargestellt).
- Ist ein **Befehl** (inkl. Zeilennummer) breiter als die Befehlsspalte (ca. 30 Zeichen), wird er auf dieselbe Weise gekürzt (z. B. PSE "Berechne die Spitzengeschwindigkeit").

STEP zeigt denselben Aufbau (Überschrift + eine Zeile) für einen einzelnen Schritt an – auch ohne TRACE.

## Step-Limit (LIMIT ON / LIMIT OFF)

Die Engine begrenzt jeden Programm-Lauf (EXEC/XEQ/CONT) auf **1 000 000 Schritte** (Konstante `_MAX_STEPS` in `calc.py`), damit fehlerhafte Programme (z. B. versehentliche Endlosschleifen) die Maschine nicht hängen – daranstoßen liefert LIMIT ERROR. Die Schranke ist **schaltbar** (persistenter Modus):

| Punkt   | Detail                                                                       |
|---------|------------------------------------------------------------------------------|
| Befehl  | LIMIT ON (an, <b>Default</b> ) bzw. LIMIT OFF (aus)                          |
| Modus   | RUN und PRGM (Sofortbefehl, keine Programmzeile)                             |
| Zustand | persistent (state.json); RESET stellt LIMIT ON (Default) wieder her          |
| Anzeige | im Prompt nur wenn an: [RUN FL=0 LIMIT]; bei LIMIT OFF kein Eintrag (wie SL) |

LIMIT OFF hebt die Schranke auf – sinnvoll für Programme, die **bewusst endlos laufen** (z. B. eine Sensorüberwachung mit WHEN/GTO-Schleife); solche Programme beendet Strg+C:

```
[ RUN] LIMIT OFF      => keine Step-Schranke mehr (kein LIMIT im Prompt)
[ RUN] EXEC           => endlose Schleife läuft, bis Strg+C
[ RUN] LIMIT ON       => Schranke wieder aktiv (Prompt zeigt LIMIT)
```

## Alarmer (ALMSET / ALMCAT / ALMDEL / ALMON / ALMOFF)

Ein Alarm ist ein Zeit-/Datumstermin, zu dem eine **Aktion** ausgeführt wird – Text-Anzeige oder Programm-Aufruf. Alarmer bleiben über Neustarts bestehen (Katalog in ~/.taschenrechner/alarms.json); ausgelöst werden sie **solange der Rechner (REPL) geöffnet ist**. Alarmer, deren Zeitpunkt während der Auszeit verstrichen ist, **verfallen** (einmalige) bzw. wiederkehrende werden auf den nächsten künftigen Termin vorgerollt.

| Punkt     | Detail                                                                                                                                                                                                                         |
|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Anlegen   | ALMSET TEXT "Meldung" · ALMSET XEQ <Ziel> – Werte vom Stack: <b>X = Zeit</b> (HH.MMSS), <b>Y = Datum</b> (DD.MMYYYY, 0 = heute), <b>Z = Intervall</b> in Dezimalstunden (0 = einmalig, 24 = täglich); Stack bleibt unverändert |
| Auflisten | ALMCAT (ID, Zeit, Datum, Intervall, Aktion, Status [inaktiv]/[abgelaufen])                                                                                                                                                     |
| Löschen   | ALMDEL <id> · ALMDEL <a>-<b> (Bereich) · ALMDEL (alle)                                                                                                                                                                         |
| Aktiv     | ALMON <id> / ALMOFF <id> (deaktivierte Alarmer bleiben erhalten, werden übersprungen)                                                                                                                                          |
| Modi      | RUN und PRGM (Sofortbefehl, keine Programmzeile)                                                                                                                                                                               |
| Fehler    | ungültige Zeit/Datum/Intervall → INVALID ALARM; unbekannter Alarm → NO ALARM                                                                                                                                                   |

```
[ RUN] 0 ENTER 28.092026 ENTER 13.4530
[ RUN] ALMSET TEXT "Wecker"
Alarm 1: 13.4530 Uhr am 28.092026, einmalig
Aktion: TEXT 'Wecker' – fällig 28.09.2026 13:45:30
[ RUN] ALMCAT
ID  Zeit      Datum      Intervall  Aktion
1  13.4530    28.092026  einmalig  TEXT Wecker
```

- **Aktionen:** TEXT zeigt die Meldung an; XEQ <Ziel> ruft ein Programm auf (gemeinsamer Stack, gemeinsame Register) – ein STOP darin übergibt an den RUN-Modus (CONT setzt fort), der Alarm gilt als ausgeführt.
- **Wiederkehrende Alarmer** (Intervall > 0) rollen nach jeder Auslösung auf den nächsten Termin vor; verpasste Termine werden **nicht** nachgeholt. **Einmalige** (Intervall 0) gelten nach der Auslösung als verbraucht.
- **Verfall:** beim Start wird der Katalog geladen und abgearbeitet – einmalige Alarmer mit vergangenem Termin verfallen (der Rechner war aus), wiederkehrende rollen auf den ersten künftigen Termin vor; inaktive bleiben unangetastet.

## Flowchart (FLOWCHART)

Zeigt ein Programm als **ASCII-Flowchart** an – in der Konsole oder in einer .flow-Datei. Globaler Sofortbefehl (RUN **und** PRGM), Stack bleibt unverändert.

| Befehl                | Wirkung                                         |
|-----------------------|-------------------------------------------------|
| FLOWCHART             | Aktuelles PRGM-Programm in der Konsole          |
| FLOWCHART <name>      | Gespeichertes Programm in der Konsole           |
| FLOWCHART SAVE <name> | Aktuelles Programm in Programme/<name>.flow     |
| FLOWCHART <name> SAVE | Gespeichertes Programm in Programme/<name>.flow |

#### Formen-Mapping:

| Form                  | Opcodes                                           |
|-----------------------|---------------------------------------------------|
| Stadium (Start/Ende)  | LBL, RTN, STOP                                    |
| Parallelogramm (E/A)  | INPUT, PRINT, PSE                                 |
| Raute (Entscheidung)  | X=0, X=Y, X<Y, X>Y, X>0, X<0, FS?, FS?C, ISG, DSG |
| Rechteck (Prozess)    | Arithmetik, RCL, STO, Konstanten, ...             |
| Doppelrahmen (Aufruf) | CALL, XEQ                                         |

#### Kanten:

| Kante              | Bedeutung                                                         |
|--------------------|-------------------------------------------------------------------|
| Spine (vertikal)   | Normaler Fluss (Knoten $i \rightarrow i+1$ ), sauber ohne Brüche  |
| YES (unten)        | Entscheidung wahr $\rightarrow$ nächste Zeile wird ausgeführt     |
| NO (Bypass rechts) | Entscheidung falsch $\rightarrow$ nächste Zeile wird übersprungen |
| GTO (rechts)       | Sprung zum Ziel-Label (eine durchgehende Linie)                   |

CALL/XEQ werden als **Doppelrahmen** gezeichnet; das Ziel ist im Kontrollflussgraphen (build\_flow) verknüpft, wird aber nicht als eigene Kante gezeichnet (XEQ kann ein externes, gespeichertes Programm aufrufen). Aufeinanderfolgende einfache Prozesse (Arithmetik, RCL/STO, ...) werden zu einer **Mehrzeilen-Box** gruppiert.

Reines Standardbibliotheken-Modul flowchart.py (analog zeit.py/alarm.py): build\_flow(program) baut den Kontrollflussgraphen, render\_flow(flow) liefert den ASCII-String. Die Engine bleibt UI-frei.

#### Grafik (PT / PT? / LINE / BOX / CIRC / CLG / TEXT / PLOT / GRID / PEN / BG / GRAPH)

Zeichnet in ein **separates Fenster** (Tkinter): Grundformen, Funktionsgraphen mit Achsen/Ticks und Text. Die Koordinaten kommen **über den Stack** (1. Wert = X, 2. = Y, 3. = Z, 4. = W), der Stack bleibt danach **unverändert** (wie PRINT/PAUSE). PLOT nutzt ein **Programm** als Funktionsdefinition – es erhält x in X und hinterlässt y in X. GRID legt das Raster der Achsen fest (Abschnitte + optionales Y-Fenster als Zoom); ohne GRID arbeitet PLOT wie bisher mit automatischen Werten.

| Befehl              | Stack (X, Y, Z, W)       | Wirkung                                                               |
|---------------------|--------------------------|-----------------------------------------------------------------------|
| PT                  | x, y                     | Punkt setzen (Eingabe: y ENTER x PT, wie S+-Paar)                     |
| PT?                 | x, y (werden konsumiert) | 1 (Tinte) bzw. 0 (Hintergrund) in X, Roll-up wie binäre Op            |
| LINE                | x1, y1, x2, y2           | Linie                                                                 |
| BOX                 | x, y, breite, höhe       | Box ab Ecke                                                           |
| CIRC                | cx, cy, r                | Kreis (Mittelpunkt + Radius, $r < 0 \rightarrow$ GRAPH ERROR)         |
| CLG                 | –                        | Grafik leeren (Stift/Hintergrund bleiben)                             |
| TEXT [alpha] <text> | x, y                     | Text an (x, y); alpha optional in Grad (90 = senkrecht, 180 = auf dem |

| Befehl                                      | Stack (X, Y, Z, W) | Wirkung                                                                                                                                                                                                                                                                |
|---------------------------------------------|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                             |                    | Kopf); Font: monospace (TkFixedFont, Konsolenfont)                                                                                                                                                                                                                     |
| PLOT <name> <x0> <x1><br><n>                | –                  | Funktionsgraph: Programm <name> wird n-mal für $x \in [x0, x1]$ aufgerufen; MATH ERROR → Lücke in der Kurve                                                                                                                                                            |
| GRID <n> <m>                                | –                  | Raster für PLOT: X-Achse in n, Y-Achse in m Abschnitte (Teilern = Abschnitte + 1); Ansichts-Bounds = exakte Daten-Bounds (ohne Padding); z. B. GRID 4 4 bei Sinus $0..2\pi \rightarrow$ X-Teiler 0, $\pi/2$ , $\pi$ , $3\pi/2$ , $2\pi$ ; Y-Teiler -1, -0.5, 0, 0.5, 1 |
| GRID <n> <m> <k> <s>                        | –                  | wie GRID <n> <m> + Y-Fenster [k, s] (Zoom/Crop): nur Y-Werte in [k, s] werden angezeigt, größere/kleinere Werte werden gecropt; die m Abschnitte unterteilen [k, s] (erfordert $k < s$ )                                                                               |
| GRID OFF                                    | –                  | Raster auf Auto zurücksetzen – PLOT arbeitet dann wie ohne GRID mit automatischen Werten                                                                                                                                                                               |
| PEN <dicke><br><strichfarbe><br>[füllfarbe] | –                  | Stift setzen: Dicke (ganze Zahl $\geq 1$ ), Farben #rrggbb; 2 Argumente = ohne Füllung (Füllung greift bei BOX/CIRC)                                                                                                                                                   |
| BG <farbe>                                  | –                  | Hintergrundfarbe setzen (Default #ffffff; PT? vergleicht dagegen)                                                                                                                                                                                                      |
| GRAPH                                       | –                  | Fenster öffnen bzw. aktualisieren (Achsen + Tick-Beschriftung im aktuellen Anzeigeformat)                                                                                                                                                                              |
| GRAPH CLOSE                                 | –                  | Fenster schließen (die Grafik bleibt erhalten)                                                                                                                                                                                                                         |
| GRAPH SAVE <name>                           | –                  | Grafik als Programme/<name>.png speichern – nur gezeichnete Grafik + TEXT, <b>ohne</b> Gitter/Achsen/Beschriftung (das GRAPH-Fenster zeigt diese weiterhin); reines Rendering ohne Tkinter; PNG für Inkscape/GIMP & Co.                                                |

Alle Zeichenbefehle laufen in RUN **und** PRGM (in PRGM als Programmzeilen). Die Grafik ist Teil von state.json (überlebt Neustarts); RESET leert sie. Das

GRID-Raster (inkl. Zoom-Fenster) ist Teil des Canvas-Zustands und wird ebenfalls in state.json gespeichert; GRID OFF bzw. RESET stellen den Auto-Modus wieder her.

**Beispiel** (Sinus, Cosinus und Tangens):

```
PRGM
GRID "8 4 -1 1"
PEN 1 "#ff0000"
PLOT SINUS "0 6.28 200"
PEN 1 "#00ff00"
PLOT COSINUS "0 6.28 200"
PEN 1 "#0000ff"
PLOT TANGENS "0 6.28 200"
RTN
LBL SINUS
SIN
RTN
LBL COSINUS
COS
RTN
LBL TANGENS
TAN
RTN

RUN
GRAPH
```

**Fehlerstrings:** GRAPH NOT AVAILABLE (kein Tkinter/Display), GRAPH ERROR (nicht-endliche Koordinaten; PLOT:  $n < 2$  oder nicht-endliche  $x_0/x_1$ ), SYNTAX ERROR (Parse; bei GRID: falsche Argumentanzahl, nicht numerisch,  $n/m < 1$  oder nicht ganzzahlig,  $k \geq s$ ), INVALID NAME (GRAPH SAVE).

**Optional wie gpiozero:** Ohne python3-tk läuft der Rechner unverändert weiter; GRAPH liefert dann GRAPH NOT AVAILABLE. Auf dem Raspberry Pi: `sudo apt install python3-tk`.

## Programme & Persistenz

- SAVE <name> speichert Programme als JSON in ~/.taschenrechner/Programme/<name>.prg (Pfad überschreibbar via Umgebungsvariable TASCHENRECHNER\_HOME).
- LOAD <name> lädt Programme aus .prg (JSON) **oder** .txt (Befehle zeilenweise, Text-Argumente in "Anführungszeichen"; leere Zeilen und #-Kommentare werden ignoriert). Die Endung darf weggelassen werden (.prg hat Vorrang). Das Programm wird nur geladen, nicht ausgeführt.
- PROGRAMS listet alle gespeicherten Programme.
- ~/.taschenrechner/seed.bin – Fortuna-Seed (64 Bytes): beim Start wird der Zufallsgenerator (RND) damit reseedet und 64 neue Zufallsbytes geschrieben (Schneier Kap. 9.6) – dadurch bleibt der Generator über Neustarts hinweg fortlaufend (Forward-Sekret).
- ~/.taschenrechner/alarms.json – Alarm-Katalog (ALMSET/ALMCAT/..., siehe [Alarmer](#)): eigene Datei (kein Teil von state.json), wird nach jeder Änderung und nach jeder Alarm-Auslösung gespeichert; beim Start geladen, verfallene Alarmer dabei abgearbeitet.
- ~/.taschenrechner/Programme/<name>.flow – ASCII-Flowchart eines Programms (siehe [FLOWCHART](#)); wird von FLOWCHART SAVE <name> bzw. FLOWCHART <name> SAVE geschrieben.
- Gespeicherte Programme lassen sich mit XEQ <name> aufrufen (siehe [XEQ](#)) – auch direkt aus dem RUN-Modus.

- **Persistenz:** Beim Beenden (Quit) wird der komplette Zustand (Stack, Speicher 0–99, aktuelles Programm, Modus RUN/PRGM, Winkel RAD/DEG, Stack-Lift SLON/SLOFF, Status-Flag SF/CF, Step-Limit LIMIT ON/LIMIT OFF, Format NORM/FIX/SCI/ENG) in ~/.taschenrechner/state.json geschrieben und beim nächsten Start automatisch wieder geladen.

Details: HELP im Rechner.

## Entwicklung

Entwicklung, Lint, Typen und Tests laufen in der **venv** (.venv):

```
source .venv/bin/activate
pip install -e .[dev]
pytest           # Tests
ruff check .     # Lint
ruff format .    # Format
mypy             # Typen
```

**Wichtig (venv ≠ Laufzeit):** Die laufende App wird über das installierte .deb gestartet und nutzt den **System-Python**, nicht die venv. taschenrechner führt die **installierte** Kopie aus, nicht den Quellbaum – Quell-Änderungen erscheinen in der laufenden App erst nach neuem Build + Installation (./build-deb.sh → sudo dpkg -i dist/...deb).

**Entwicklungsrechner:** Die GPIO-Entwicklung läuft auf dem **Raspberry Pi 500** (ARM64, echte Hardware + gpiozero). Ein zweiter **X86-Entwicklungsrechner** dient der allgemeinen Entwicklung; Quell-Dateien werden manuell getauscht (dort kein GPIO).

## Paket (.deb) bauen & installieren

Für die Installation auf Debian/Ubuntu/LMDE-Systemen (auch Raspberry Pi OS, nur dpkg-deb nötig) – das Paket ist architekturunabhängig (Architecture: all) und läuft sowohl auf X86 als auch auf ARM:

```
./build-deb.sh           # -> dist/taschenrechner_<ve
dpkg-deb --info dist/taschenrechner_*.deb    # Info prüfen
dpkg-deb --contents dist/taschenrechner_*.deb # Inhalt prüfen
```

Auf dem Ziel-System:

```
sudo dpkg -i taschenrechner_0.1.0_all.deb    # installieren
taschenrechner                               # starten
sudo dpkg -r taschenrechner                  # deinstallieren
```

Das Paket legt die Dateien nach /usr/lib/python3/dist-packages/ und einen Launcher /usr/bin/taschenrechner ab (nutzt /usr/bin/python3); einzige Abhängigkeit: python3 (>= 3.10).

## Projektlayout

```
taschenrechner/
  __init__.py    # Version (__version__)
  __main__.py    # Start via python -m taschenrechner
  stack.py       # 4-stufiger Register-Stack
  instruction.py # Opcodes + Parser
  display.py     # Anzeige-Formate (NORM/FIX/SCI/ENG)
```

|              |                                                                      |
|--------------|----------------------------------------------------------------------|
| storage.py   | # Save/LOAD + Persistenz (Programme/, state.json, seed.bin, alarms.j |
| fortuna.py   | # Zufallszahlen: Schneiers Fortuna (AES-256, Kap. 9) für RND         |
| zeit.py      | # Zeit/Datum: TIME/DATE (HH.MMSShh, DD.MMYYYY, lokale Zeit)          |
|              | # + Datumsarithmetik DATE+/DATE-/DDAYS (add_days/sub_days/diff_day   |
| alarm.py     | # Alarm-Katalog: ALMSET/ALMCAT/ALMDEL/ALMON/ALMOFF (Katalog,         |
|              | # Alarm, parse_time, fmt_zeit/fmt_datum, due/after_fire/expire)      |
| flowchart.py | # FLOWCHART: Programme als ASCII-Flowchart rendern (build_flow,      |
|              | # render_flow, Formen: Stadium/Parallelogramm/Raute/Rechteck/Doppe   |
| graph.py     | # Grafik-Modell: Primitiven (Punkt/Linie/Box/Kreis/Polyline/Text),   |
|              | # Canvas, View (Bounding-Box + gleiche Skalierung), Ticks,           |
|              | # Hit-Test, Sampling, Farben + PNG-Rendering (render_png,            |
|              | # 5x7-Bitmap-Font, zlib/struct) für GRAPH SAVE (rein, UI-frei, Std   |
| gpio.py      | # GPIO: BOARD/BCM-Pin-Zuordnung + Geräte (LED, PWMLED, BUZZER,       |
|              | # TONALBUZZER, MOTOR, RGBLED, PIR, LDR, HCSR04, BUTTON per           |
|              | # gpiozero, lazy Import)                                             |
| errors.py    | # CalcError                                                          |
| calc.py      | # Engine: Modi, Speicher, Programme, Serialisierung                  |
| cli.py       | # REPL / Kommandozeile                                               |
| tests/       | # pytest-Suite                                                       |