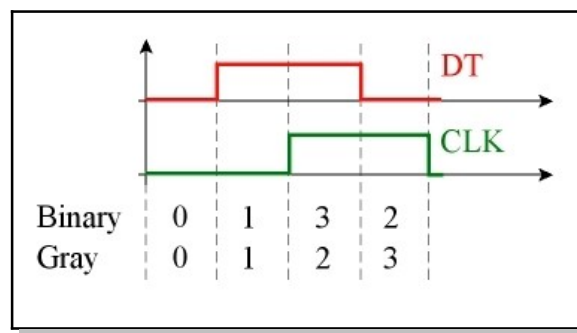


Der Drehgeber KY-040



Aufbau, Funktionsweise und
Micropython-Programmierung

Eine Einführung
mit dem TTGO T-Display

G. Heinrichs
21.03.2026

1 Übersicht

Das KY-040-Modul (Abb. 1) wandelt Drehbewegungen in digitale Signale; dabei erfasst es sowohl den Drehwinkel als auch die Drehrichtung.

Dreht man die Achse des KY-040, spürt man eine Rasterung. Eine vollständige Drehung dieser Achse hat eine bestimmte Anzahl von Rastern. Ich besitze zwei Typen mit unterschiedlichen Eigenschaften:

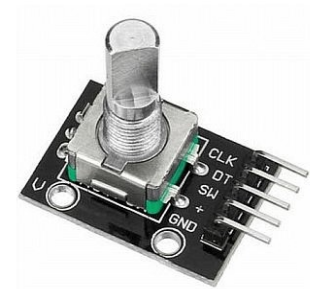


Abb. 1

Typ	Rasterpositionen/Umdrehung	Anzahl der Signale/Umdrehung
"alt"	30	15
"neu"	20	20

Beim Typ "neu" gibt es also bei jedem Raster ein Signal, beim Typ "alt" erst bei jedem zweiten. Die Raster-Anzahl beim Drehen ist damit einem bestimmten Drehwinkel zugeordnet. Zählt man die entsprechenden Signale, kann man den Drehwinkel daraus berechnen.

Das KY-040-Modul besitzt 5 Anschlüsse:

"+" (bzw. "VCC") und "GND" für die Stromversorgung (Spannung zwischen 3,3 V und 5,0 V)
 "CLK" und "DT" für die Ausgangssignale

Der Ausgang "SW" (Switch) hat den Wert High (1) oder (Low), je nachdem ob der Knopf bzw. die Drehachse nach unten gedrückt wird oder nicht. Mit dieser Zusatzfunktion werden wir uns in diesem Beitrag nicht weiter beschäftigen.

2 Aufbau

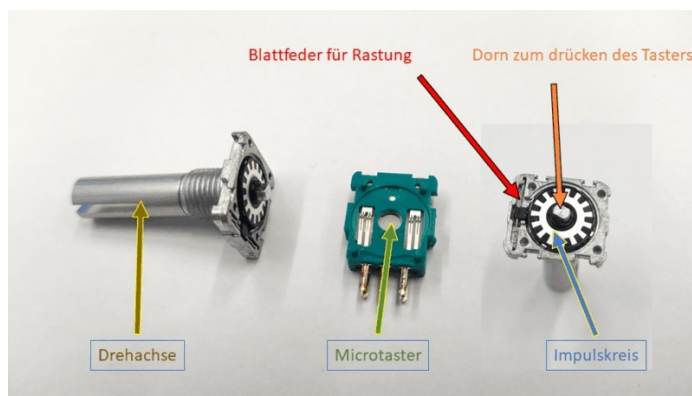


Abb. 2

Bei der Drehung der Achse bewegt sich das Zahnrad über die beiden Kontakte (roter und grüner Kreis in Abb. 3). Das Zahnrad ist mit dem Anschluss GND verbunden. Beide Kontaktpunkte sind jeweils über einen Pull-Up-Widerstand mit VCC verbunden. Sobald ein Zahn einen Kontakt berührt, wird sein Potential auf 0 (Low) gezogen, ansonsten hat er den Zustand 1 (High). Diese Kontakte liefern die Signale CLK und DT. Auf welche Weise sie sich beim Drehen der Achse ändern, wird uns die gewünschten Dreh-Informationen liefern. Wie wir im nächsten Abschnitt sehen werden, ist dafür ganz entscheidend, dass sich die beiden Kontakte NICHT auf einer Geraden durch den Mittelpunkt der Scheibe befinden. Deswegen können die Signale von CLK und DT je nach Stellung des Zahnrads unterschiedlich sein. In der Abb. 3 haben wir zur Vereinfachung die Anzahl der Zähne verringert; außerdem wurden die Kontakte so verschoben, dass sie näher beieinander liegen (die Phasen-Beziehung sich dabei im Wesentlichen aber nicht ändert).

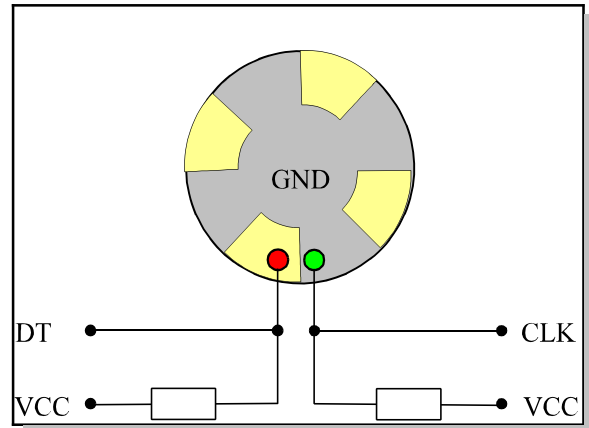


Abb. 3

3 Signalfolge bei einer Drehung

Mit Hilfe der Abb. 4 können wir erkennen, wie sich die Signale bei DT und CLK verhalten, wenn eine Drehung um 2 Raster (alter Drehgeber) bzw. um 1 Raster (neuer Drehgeber) **im Uhrzeigersinn** vorgenommen wird.

Zunächst haben beide Kontaktpunkte keine leitende Verbindung zum Zahnrad; wegen der Pull-Up-Widerstände haben DT und CLK ein 1-Potential (Phase A).

Nach einer kleinen Drehung im Uhrzeigersinn gelangen wir zur nächsten Phase (B): Jetzt liegt bei CLK ein 0-Potential vor; DT bleibt auf 1.

Im nächsten Schritt (C) gelangen wir zur Phase C: Jetzt haben sowohl der DT- als auch der CLK-Anschluss Kontakt zum Zahnrad; sie haben deswegen beide 0-Potential.

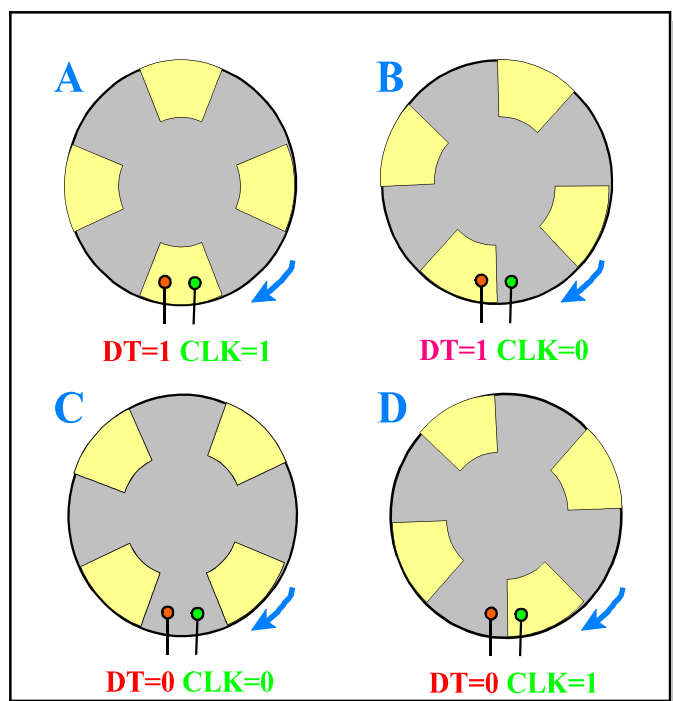


Abb. 4

In der letzten Phase (D) hat DT noch das Potential 0; CLK hat 1-Potential, weil kein Kontakt mehr zu GND vorliegt.

Bei weiterer Drehung wiederholt sich die Signalfolge. In Abb. 5 sieht man, dass die Signale für CLK und DT sich gleichen und nur um eine Phasenverschiebung von $1/4$ der Periode unterscheiden. Man beachte, dass das Diagramm aus Abb. 5 mit einer anderen Phase beginnt als in Abb. 4.

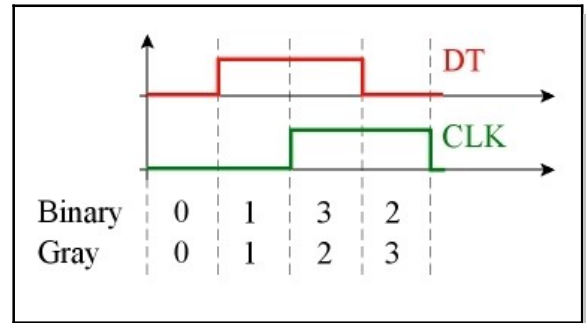


Abb. 5

Die Signalfolge für die 4 Phasen aus Abb. 4 können wir übersichtlicher darstellen, indem wir jeder Phase den 2-Bit-Wert $2 \cdot \text{CLK} + \text{DT}$ zuordnen: [3, 1, 0, 2]. Durch zyklische Vertauschung erhalten wir die Signalfolgen für eine Rechtsdrehung mit einer anderer Startposition: [1, 0, 2, 3], [0, 2, 3, 1] und [2, 3, 1, 0].

Bei einer Drehung **gegen den Uhrzeigersinn** werden die Phasen in Abb. 4 in umgekehrter Reihenfolge durchlaufen. Eine zugehörige Signalfolge ist dann z. B. [2, 0, 1, 3].

Im nächsten Abschnitt stellen wir ein Programm vor, welches diese Folgen von 2-Bit-Werten ausnutzt, um die Anzahl der Dreh-Schritte sowie die Drehrichtung zu ermitteln.

4 Drehgeber-Programm mit 4-Zyklus

Das folgende Programm zeigt die aktuelle "Position" des Drehgebers an: Der Positionswert wird bei einem Zyklus um 1 erhöht, wenn die Drehung im Uhrzeigersinn erfolgt, und um 1 verringert, wenn die Drehung gegen den Uhrzeigersinn erfolgt. Das Programm beginnt mit dem Positionswert 0. Der aktuelle Positionswert wird auf dem Display angezeigt; weitere Informationen werden im Terminal angezeigt.

```
# KY040_2c.py, s. Materialiensammlung
# Drehgeber
# www.g-heinrichs.de

from time import sleep_ms
import vga1_bold_16x32 as font1
import st7789
from machine import SPI, Pin

# Display instanziiieren:
spi = SPI(1, baudrate=200000000, polarity=1, sck=Pin(18), mosi=Pin(19))
display = st7789.ST7789(spi, 135, 240, reset=Pin(23, Pin.OUT),
                        cs=Pin(5, Pin.OUT), dc=Pin(16, Pin.OUT),
```

```
        backlight=Pin(4, Pin.OUT), rotation=3)

# Landscape
display.init()
display.fill(st7789.BLUE)
display.text(font1, 'KY040', 80, 10, st7789.WHITE, st7789.BLUE)

# Pins CLK u. DT instanziiieren:
CLK = Pin(12, Pin.IN) # Clock, PullUp auf Modul
DT = Pin(13, Pin.IN)  # Data, PullUp auf Modul

print('Start des Programms')
old_phase = CLK.value()*2 + DT.value()
steps = []
n = 0 # Phasenzähler
position = 0

while True:
    new_phase = CLK.value()*2 + DT.value()
    if new_phase != old_phase:
        n += 1
        steps.append(new_phase)
        sleep_ms(2) # wichtig!
        old_phase = new_phase
        if n == 4:
            print(steps)
            if (steps == [2,3,1,0] or steps == [3,1,0,2]) or
                (steps == [1,0,2,3] or steps == [0,2,3,1]) :
                position = position + 1 # Drehung im Uhrzeigersinn
                display.text(font1, str(position) + '    ', 80, 60,
                             st7789.WHITE, st7789.BLUE)

                print('rechts', position)
            elif (steps == [2,0,1,3] or steps == [1,3,2,0]) or
                (steps == [0,1,3,2] or steps == [3,2,0,1]) :
                position = position - 1 # Drehung gegen den Uhrzeigersinn
                display.text(font1, str(position) + '    ', 80, 60,
                             st7789.WHITE, st7789.BLUE)

                print('links', position)
            else: # Fehler
                print('Falscher Code!')
            n = 0
            steps = []
```

Hier als Beispiel ein kurzes Protokoll von einer Drehung im Uhrzeigersinn und gegen den Uhrzeigersinn.

```
[2, 0, 1, 3]
links 1
[1, 0, 2, 3]
rechts 2
```

```
[1, 0, 2, 3]
rechts 3
[1, 0, 1, 0]
Falscher Code!
[2, 3, 1, 0]
rechts 4
[2, 3, 1, 0]
rechts 5
```

Auffällig sind hier die Fehlermeldungen 'Falscher Code'. Warum können solche Fehlermeldungen überhaupt auftreten?

Die Antwort liefert eine Signal-Analyse mit einem Digital-Analysator. Abb. 6 zeigt (rot markiert) eine Situation, welche eine solche Fehlermeldung veranlassen kann: Die Signalfolge entspricht hier nicht den in Abschnitt 3 angegebenen Zyklen. Es handelt sich hier um Störungen, die durch **Prellen** während des Umschaltens beim CLK/DT Kontakt auftreten können. Derartige Störungen können durch so genanntes **Entprellen (debouncing)** unterdrückt werden. Dies kann hardware- oder software-mäßig geschehen.

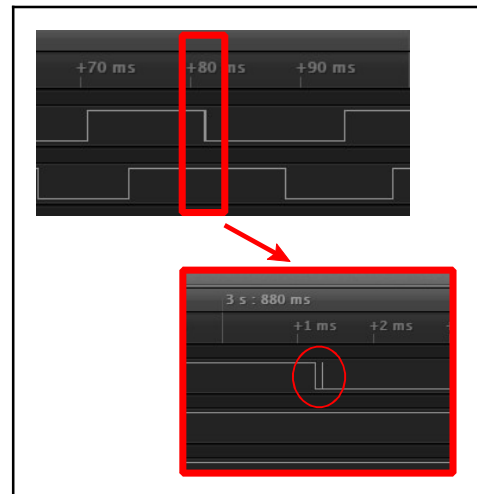


Abb. 6

Ein hardware-mäßiges Entprellen kann z. B. mit einem so genannten **Tiefpassfilter** erfolgen; dieser besteht aus einem Kondensator und einem Widerstand (s. Abb. 7).

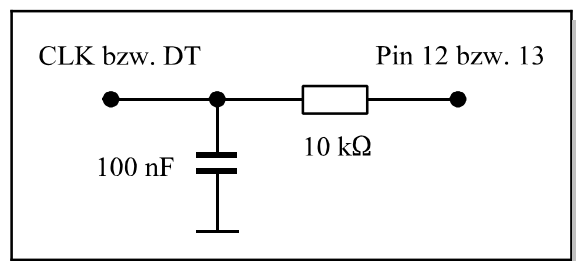


Abb. 7

Zuletzt noch eine Anregung zur Erweiterung dieses Programms: Beim Drehen soll das Programm dafür sorgen, dass der Positionswert vorgegebene Maximal- und Minimalwerte nicht über- bzw. unterschreitet.

Intermezzo: KY-040 mit optischem Schalter?

The optical encoder, like the KY-040 works on a disc with marks or holes and an optical transmitter/receiver system (photodiode or LED). As the shaft rotates, the interruptions produced by these marks generate electrical pulses that the microcontroller can count, thus determining the angular displacement. Its internal construction typically includes a static part (the disc) and a rotating part (the shaft attached to the element being measured). Quelle: [OMR]

Tatsächlich gibt es optische Drehgeber, aber das KY-040-Modul ist definitiv ein mechanischer Drehgeber. Optische Drehgeber sind übrigens deutlich teurer als das KY-040-Modul.

5 Drehgeber-Programm mit verschachtelten Verzweigungen

Das hier vorgestellte Programm findet man an einigen Stellen im Internet (z. B. [SP]). Das Programm ist kürzer als das aus dem letzten Abschnitt, aber vielleicht nicht so einfach nachvollziehbar wie das Programm aus dem letzten Abschnitt. Bei dem folgenden Programm habe ich mich auf das Wesentliche konzentriert, nämlich die Detektierung des Drehsinns (links/rechts). Wer will kann das Programm in ähnlicher Weise ausbauen wie im letzten Abschnitt dargestellt.

Die einzelnen Programmzeilen sind ausführlich kommentiert; es empfiehlt sich, das Programm an Hand der 4 Phasen aus Abb. 4 Schritt für Schritt durchzugehen.

```
# KY_040_simple.py, s. Materialiensammlung
# Das Programm funktioniert auch recht gut ohne Pausen
# UND OHNE Tiefpassfilter

from machine import Pin
from time import sleep

DT = Pin(13, Pin.IN)
CLK = Pin(12, Pin.IN)

delta_t = 0.5
previous_value = True

print('Programm gestartet...')

while True:
    if previous_value != DT.value():
        # wenn DT seinen vorigen Wert ändert, dann... [A]
        if DT.value() == 0:
            # wenn DT = 0 ist, dann...
            if CLK.value() == 0:
                # wenn CLK = 0 ist, dann
                print('rechts')
                # Rechtsdrehung ("im Uhrzeigersinn") anzeigen
                # sleep(delta_t)
                # und ggf. etwas warten (s. o.)
            else:
                # sonst (wenn CLK = 1 ist):
                print('links')
                # Linksdrehung ("im Gegenuhrzeigersinn") anzeigen
                # sleep(delta_t)
                # und ggf. etwas warten (s. o.)
        previous_value = DT.value()
        # den >geänderten< Wert von DT merken (s. [A])
        # nächster Schleifendurchlauf
```

Aufgabe: Erweitern Sie das Programm `KY_040_simple.py` um eine Positionsangabe.

6 Helligkeits-Regler

In diesem Abschnitt wollen wir zeigen, wie man mit unserem Drehgeber die Helligkeit einer LED

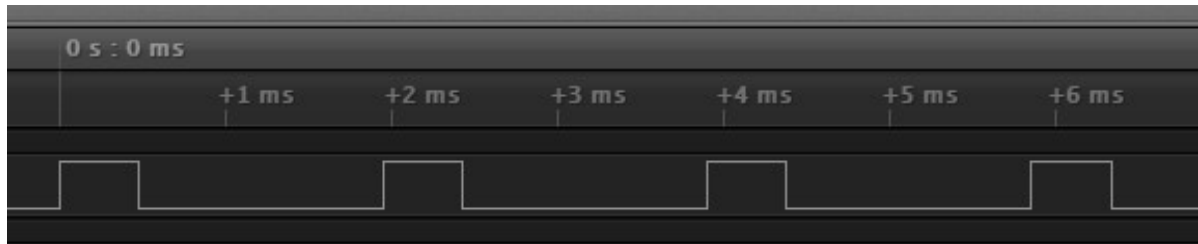


Abb. 8

steuern kann. Unterschiedliche Helligkeiten erreicht man auf einfache Weise mit Hilfe einer **Pulsweitenmodulation (PWM)**: Die LED wird dabei nicht über verschieden große Spannungen gesteuert; vielmehr wird die LED (bei konstanter Spannung) periodisch ein- und ausgeschaltet. Die Frequenz ist dabei so hoch, dass das menschliche Auge kein Flackern mehr wahrnimmt; je nach Pulsweite nimmt es eine größere oder kleinere Leuchtkraft wahr.

Eine solche Pulsweitenmodulation kann mit wenigen Befehlen erzeugt werden. Zum Testen schließen wir zunächst eine LED (über einen Widerstand von 100 Ω) an den Pin 25 an. Das Testprogramm besteht nur aus wenigen Zeilen:

```
from machine import Pin, PWM
frequency = 512                # in Hz
led = Pin(25)                  # led-Objekt erzeugen
pwm = PWM(led, frequency)      # pwm-Objekt zum led-Objekt erzeugen
pwm.duty(250)                  # duty-Wert festlegen
```

In diesem Fall wird ein PWM-Signal mit einer Frequenz von 512 Hz erzeugt; die Periode ist dann $T = 1/512 \text{ s} = 1,953 \text{ ms}$. In dieser Zeit wird die LED einmal ein- und ausgeschaltet. Wie sich dabei die Ein- und Ausphase zueinander verhalten, hängt von dem `duty`-Wert ab: Dieser liegt zwischen 0 und 1023. Ist der `duty`-Wert gleich 0, dann ist die LED während der gesamten Periode ausgeschaltet, ist der Wert gleich 1, dann ist er nur für $1,953/1024 \text{ ms}$ eingeschaltet. Bei einem `duty`-Wert von 1023 ist die LED während der gesamten Periode eingeschaltet.

Für den `duty`-Wert 250 ist das zugehörige Signal in Abb. 8 zu sehen. Man erkennt mit bloßem Auge, dass die 1-Phase ungefähr $1/4$ mal so lang ist wie die Periode.

Für den Helligkeitsregler gehen wir von unserem Programm `KY040_2c.py` aus: hier fügen wir die obigen Zeilen vor dem Hauptprogramm ein (Der Import des Pin-Moduls kann jetzt allerdings wegfallen, weil er schon in dem Programm `KY040_2c.py` steht.) Zudem legen wir den `duty`-

Startwert auf 0 fest.

Im Hauptprogramm fügen wir vor der while-Schleife die folgenden Zeilen ein:

```
p_min = 0
p_max = 5
duty_list = [0, 50, 100, 200, 300, 500]
# Die wahrgenommene Helligkeit nimmt nicht proportional zum duty-Wert zu
```

Mit den ersten beiden Zeilen legen wir den minimalen und den maximalen Positionswert fest; die Inkrementierung der Variablen `position` innerhalb der Schleife erfolgt jetzt nur noch, wenn der aktuelle Positionswert kleiner als `p_max` ist:

```
if position < p_max:
    position = position + 1
```

Analog verfahren wir bei der Dekrementierung:

```
if position > p_min:
    position = position - 1
```

Am Ende des if-Blocks aktualisieren wir den `duty`-Wert mit

```
pwm.duty(duty_list[position])
```

Ist der Wert von der Variablen `position` z. B. 2, dann wird der `duty`-Wert auf 200 gesetzt. (Mit Hilfe der Listenwerte können wir unterschiedliche Profile für den Anstieg der Helligkeit festlegen.) Genauso verfahren wir auch am Ende des else-Blocks.

Das gesamte Programm finden Sie auch unter dem Dateinamen `KY040_pwm_1.py` in der Materialiensammlung [MAT].

7 Drehgeber-Programm mit Interrupts

Bei unseren Programmen befand sich der Code für die Steuerung des Drehgebers immer im Hauptprogramm. In manchen Fällen - insbesondere bei größeren oder komplexeren Aufgaben - kann es sinnvoll, den KY-040-Code in Interrupt-Routinen zu verlagern.

In diesem Abschnitt stellen wir ein entsprechendes Programm vor; dabei greifen wir für die Auswertung der Signale auf die Vorgehensweise aus dem Abschnitt 4 zurück. Die Interrupt-Routine für den Eingang DT sieht so aus:

```
# IR-Routine für DT
def callback1(pin): # wenn EIN Signal wechselt, ist das ANDERE konstant
```

```
pin.irq(trigger = Pin.IRQ_FALLING | Pin.IRQ_RISING, handler = None)
global CLK_val
global DT_val
global phase
global steps
global nr
global position
DT_val = pin.value()
phase = 2*CLK_val + DT_val
steps.append(phase)
LED.value(1) # zum Testen
sleep_ms(1)
LED.value(0)
if nr == 3:
    if (steps == [2,3,1,0] or steps == [3,1,0,2])
        or (steps == [1,0,2,3] or steps == [0,2,3,1]) :
        position = position + 1
        # print('position =', position) # ggf. (nur) im Hauptprogramm
        display.text(font1, str(position) + ' ', 80, 60,
            st7789.WHITE, st7789.BLUE) # ggf. im Hauptprogramm
    elif (steps == [2,0,1,3] or steps == [1,3,2,0])
        or (steps == [0,1,3,2] or steps == [3,2,0,1]) :
        position = position - 1
        # print('position =', position) # ggf. (nur) im Hauptprogramm
        display.text(font1, str(position) + ' ', 80, 60,
            st7789.WHITE, st7789.BLUE) # ggf. im Hauptprogramm
    nr = 0
    steps = []
else:
    nr += 1
pin.irq(trigger = Pin.IRQ_FALLING | Pin.IRQ_RISING, handler = callback1)
```

Die Interrupt-Routine `callback2` für CLK sieht fast genauso aus: Es werden nur "`DT_val = pin.value()`" gegen "`CLK_val = pin.value()`" und "`callback1`" gegen "`callback2`" getauscht.

Die Interrupts werden zu Beginn des Hauptprogramms "aktiviert":

```
DT.irq(trigger = Pin.IRQ_FALLING | Pin.IRQ_RISING, handler = callback1)
CLK.irq(trigger = Pin.IRQ_FALLING | Pin.IRQ_RISING, handler = callback2)
```

Die Interrupt-Routinen werden aufgerufen, wenn an den jeweiligen Pins das Signal fällt oder steigt.

Schauen wir uns nun die Interrupt-Routine für DT an: Auffällig ist, dass direkt nach dem Aufruf der Interrupt mit `handler = None` gesperrt wird. Der Grund ist folgender: Micropython arbeitet Interrupt-Routinen nicht unbedingt sofort ab; vielmehr kommen sie in eine "Warteschlange" (**queue** genannt). Zu gegebener Zeit werden dann alle Interrupts dieser queue hintereinander abgearbeitet. Durch unsere Sperrung wird nun erst gar keine queue gebildet; vielmehr werden

sofort die einzelnen Befehle der Interrupt-Routine ausgeführt. Am Ende dieser Routine aktivieren wir schließlich den Interrupt wieder. (Mehr zu diesem Thema finden Sie in [QUE]).

Die restlichen Befehle sind uns fast alle aus Abschnitt 4 bekannt. Lediglich die Benutzung des Schlüsselworts `global` müssen wir noch erläutern: Mit dem Befehl `global clk_value` steht die Variable `clk_value` aus der `callback1`-Routine jetzt zusätzlich nicht nur im Hauptprogramm, sondern auch in der `callback2`-Routine zur Verfügung.

Aus diesem Grund kann man auch im Hauptprogramm (am Ende des Programms) im Rahmen einer Schleife den aktuellen Wert von **position** anzeigen lassen:

```
print('Programm gestartet...')
old_position = position
print('Startposition =', old_position)
print()

while True:
    new_position = position
    if new_position != old_position:
        print('position =', new_position)
        old_position = new_position
```

Das gesamte Programm finden Sie unter dem Dateinamen `KY040_irq_0.py` in der Materialiensammlung [MAT].

Zum Testen habe ich hier keinen realen Drehgeber, sondern ein entsprechendes Simulationsprogramm benutzt. Das zugehörige Programm `KY040_simulator1.py` finden Sie ebenfalls in der Materialiensammlung [MAT].

Beispiel:

Zunächst wird der Simulator-TTGO (Sender) an den Signal-Analysator-TTGO (Empfänger) angeschlossen: Dazu werden die beiden GND-Anschlüsse und die Daten-Pins (Pin12 an Pin12, Pin13 an Pin13) miteinander verbunden.

Nun starten wir zunächst das Programm `KY040_irq_0.py` auf dem Signal-Analysator-TTGO; es erscheint die Meldung:

```
Programm gestartet...
Startposition = 0
```

Danach starten wir das Simulator-Programm; dieses sendet zunächst **die Signale für 3 steps im Uhrzeigersinn**, anschließend **die Signale für 4 steps im Gegenuhrzeigersinn** und zuletzt **die Signale für 5 steps im Uhrzeigersinn**. Im Terminal vom Simulator-TTGO wird dies verkürzt so protokolliert:

Start des Programms

```
3 * [2, 3, 1, 0]
4 * [2, 0, 1, 3]
5 * [2, 3, 1, 0]
```

Auf dem Terminal des Signal-Analysators erscheint die folgende Anzeige:

```
position = 1
position = 2
position = 3
position = 2
position = 1
position = 0
position = -1
position = 0
position = 1
position = 2
position = 3
position = 4
```

8 Eine Micropython-Klasse für den KY-040

Im Internet findet man eine Reihe von Micropython-Klassen, mit deren Hilfe man die Signale eines KY-040-Drehgebers auswerten kann. Üblicherweise kann man damit über eine Klassen-Methode (häufig mit **value** bezeichnet) die aktuelle Position des Drehgebers erhalten. Der Vorteil ist: Der Anwender muss sich nicht mehr um die Funktionsweise dieser Klassen kümmern und kann sich darauf konzentrieren, wie die erhaltenen Positionswerte des Drehegebers genutzt werden können.

Bei der Auswahl solcher Klassen muss man etwas aufpassen: Nicht alle Micropython-Programme mit solchen KY-040-Klassen funktionieren auf unserem TTGO. Der Grund dafür ist, dass sie häufig nicht für den ESP32, sondern für andere Hardware (z. B. den Pico) geschrieben worden sind.

Hier möchte ich ein Software-Paket vorstellen, welches auch für den TTGO (mit dem ESP32) funktioniert. Es stammt von Mike Teachman [MT].

Das Paket besteht aus mehreren Dateien:

- `ky_040_main_1.py` (Hauptprogramm)
- `rotary.py` (Basis-Klasse Rotary)
- `rotary_irq_esp.py` (davon abgeleitete Klasse RotaryIRQ)

Die beiden letzten Dateien müssen in den Speicher des ESP32 geladen werden. Sie stellen die Klassen **Rotary** und **RotaryIRQ** zur Verfügung. Wie im Abschnitt 7 werden auch hier die CLK- und DT-Signale mit Hilfe von Interrupts erfasst.

Eine Instanz von `RotaryIRQ` wird folgendermaßen erzeugt:

```
r = RotaryIRQ(pin_num_clk=12,
               pin_num_dt=13,
               min_val=-10,
               max_val=10,
               reverse=False,
               range_mode=RotaryIRQ.RANGE_WRAP)
```

Hierbei legen `min_val` und `max_val` den minimalen bzw. maximalen Wert für die Position des Drehgebers fest. Mit dem Parameter `range_mode` wird festgelegt, ob nach dem Erreichen des minimalen bzw. maximalen Positionswertes dieser Positionswert im nächsten Schritt beibehalten wird (`RANGE_BOUNDED`) oder ob er zum maximalen bzw. minimalen Positionswert übergeht (`RANGE_WRAP`).

Den aktuellen Positionswert des Drehgebers liefert die Methode `value`:

```
pos_val = r.value()
```

Dieser Wert wird ähnlich wie im Abschnitt 7 durch Interrupts automatisch (sozusagen im Hintergrund) aktualisiert.

Ein einfaches Programm, welches in einer Schleife den aktuellen Positionswert anzeigt, befindet sich in der Datei `ky_040_main_1.py` (s. Materialiensammlung [MAT]).

Testen Sie das Programm aus: Vergrößern Sie dabei auch einmal die Warte-Zeit zwischen den Abfragen des Positionswertes (z. B. von 50 ms auf 300 ms). Wenn Sie nun den Drehgeber zügig drehen, werden innerhalb dieser Warte-Zeit mehrere Interrupts erfolgen; die nacheinander angezeigten Positionswerte werden sich nun nicht mehr nur um 1 unterscheiden.

Zwei weitere Anwendungen dieser Klasse möchte ich hier noch kurz vorstellen:

rect_color_1.py

Hier wird mit dem Drehgeber die Farbe eines Quadrats gesteuert, welches auf dem Display des TTGO angezeigt wird. Ich gehe hier nur auf die wichtigsten Teile des Programms ein:

```
# Farb-Palette:
color_r_g_b_list = [0b11101_00000000_000000, 0b11111_00000000_000000, ...]

# Größe und Position des Quadrats
d = 50
```

```
x_pos = 105
y_pos = 60

# Funktionen
def set_rect(x, y, d, c): # x, y: Position; d: Seitenlaenge; c: Farbwert aus
                           color_r_g_b_list
    display.fill_rect(x, y, d, d, c)

##### Hauptprogramm #####

print('pixel_regen_0.py gestartet')

old_val = r.value()
print('color =', old_val)
while True:
    set_rect(x_pos, y_pos, d, color_r_g_b_list[old_val])
    sleep_ms(100)
    new_val = r.value()
    if old_val != new_val:
        old_val = new_val
        print('color =', new_val)
```

Das vollständige Programm `rect_color_1.py` finden Sie in der Materialiensammlung [MAT].

pixel_regen_1.py

Bei diesem Programm wird mit dem Drehgeber die Farbe eines "Pixel-Regens" gesteuert. Auch hier gehe ich nur auf die wichtigsten Teile ein:

```
# Farb-Palette:
color_r_g_b_list = [0b11101_00000000_000000, 0b11111_00000000_000000, ...

# Grafik:
x_max = 234    # max. x-Position eines "Pixels"
y_max = 129    # max. y-Position eines "Pixels"
d = 6          # Größe eines "Pixels"

# Funktionen:
def set_pixel(x, y, d, c): # x, y: Position, d: Pixel-Größe;
                           c: Farbwert aus color_r_g_b_list
    display.fill_rect(x, y, d, d, c)

##### Hauptprogramm #####

display.text(font1, 'KY-040', 80, 10, color_r_g_b_list[0], st7789.BLACK)
print('pixel_regen_0.py gestartet')
```

```
old_val = r.value()
print('color =', old_val)
while True:
    pixel_x = randint(0, x_max)
    pixel_y = randint(0, y_max)
    set_pixel(pixel_x, pixel_y, d, color_r_g_b_list[old_val])
    sleep_ms(100)
    new_val = r.value()
    if old_val != new_val:
        print('color =', new_val)
        display.text(font1, 'KY-040', 80, 10, color_r_g_b_list[new_val],
                      st7789.BLACK)

    old_val = new_val
```

Nach dem Starten des Programms erscheinen auf dem Display des TTGO an zufälligen Positionen "Pixel" (Quadrate mit der Seitenlänge 6); diese haben zunächst die Farbe mit dem Index 0 aus der Farb-Palette. Betätigen wir den Drehgeber, ändert sich die Farbe der neu erscheinenden "Pixel" entsprechend der zur Position des Drehgebers passenden Farbe aus der Farb-Palette `color_r_g_b_list`.

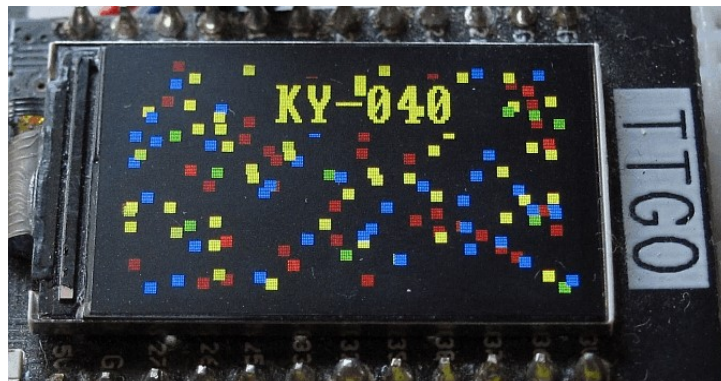


Abb. 9

Das vollständige Programm `pixel_regen_1.py` sowie ein dazu gehöriges kurzes Video (`pixel_regen_1.mp4`) finden Sie in der Materialiensammlung [MAT].

Quellen und Materialien:

Abb.2: https://wiki.hshl.de/wiki/index.php/Drehimpulsgeber_KY-040

[SP]: <https://www.youtube.com/watch?v=3fAFNwA-aEY>

[OMR]: <https://en.hwlibre.com/Optical-and-magnetic-rotary-encoder%3A-differences-in-operation-and-examples-with-KY-040-and-AS5600/>

[QUE] <https://www.i-programmer.info/programming/148-hardware/17088-esp32-in-micropython-interrupts.html?start=1>

[MT]: <https://github.com/miketeachman/micropython-rotary>

[MAT]: Materialiensammlung: <https://forum.g-heinrichs.de/viewtopic.php?f=18&t=221&p=295#p295>