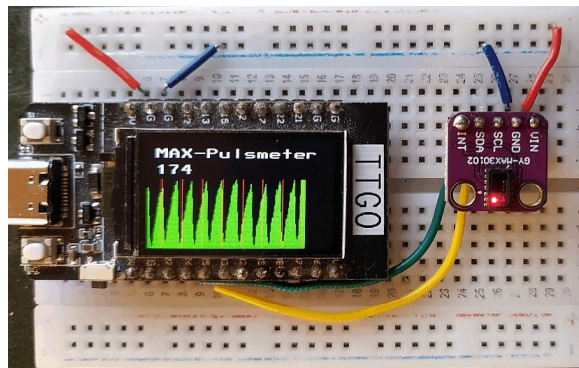




Messung von Pulsraten mit dem MAX30102 und dem TTGO T-Display



*Eine Einführung von
G. Heinrichs*

Stand: 25.03.2025

1 Einführung

Das **Modul GY-30102** (Abb. 1) dient zur Messung der **Pulsrate** und der **Sauerstoffkonzentration** des Blutes; derartige Bausteine bzw. Geräte bezeichnet man häufig auch als **Pulsoximeter**.

Wesentliche Elemente dieses Moduls sind eine rote LED ($\lambda \approx 660 \text{ nm}$), eine Infrarot-LED (kurz: IR-LED, $\lambda \approx 880 \text{ nm}$) sowie ein Lichtsensor, der auf sichtbares und infrarotes Licht anspricht.



Abb. 1

Wie funktioniert die Messung der Pulsrate? Der Proband legt die Kuppe eines Fingers auf die LED-Sensor-Kombination. Die rote LED beleuchtet die Haut des Fingers; ein Teil des Lichts wird vom Gewebe reflektiert und gelangt zum Lichtsensor. Dieser liefert ein elektrisches Signal, das von der Menge und der Sauerstoffkonzentration des Blutes in den Kapillaren abhängt. Während der Systole ist diese Blutmenge größer als bei der Diastole (vgl. Abb. 2).

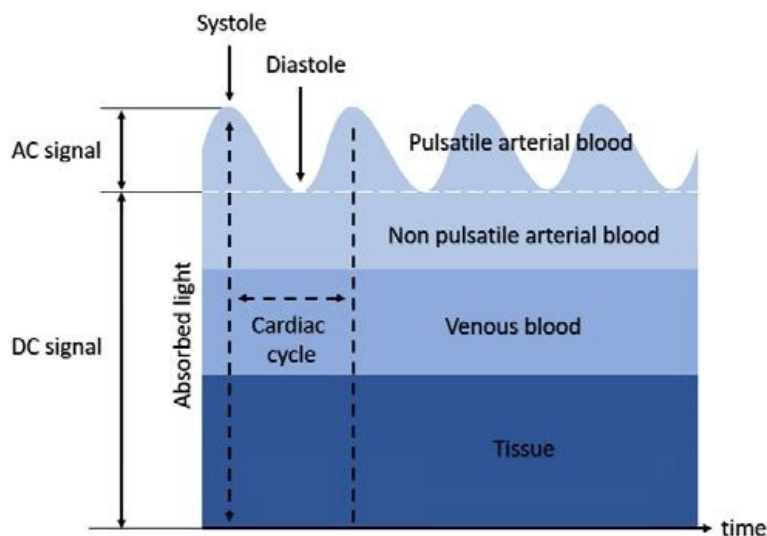


Abb. 2 (nach [2], S. 5)

Ein so gewonnenes Blutmengen-Zeit-Diagramm nennt man auch **Photoplethysmogramm** (kurz: PPT); plethys = Menge, Fülle)

Wesentliches Bauteil des GY-30102-Moduls ist der **MAX30102-IC**. Neben den oben schon erwähnten LEDs und dem Lichtsensor stellt er eine Vielfalt von Funktionen zur Verfügung: So kann man (über die Stromstärke) die Helligkeit der LEDs einstellen; zudem lässt sich damit auch die Pulsweite der LEDs einstellen. Dass die LEDs nicht permanent leuchten, sorgt auch dafür, dass der MAX30102-IC mit wenig Energie auskommt: Der IC selbst wird mit einer Spannung von 1,8 V betrieben; eine

typische Stromstärke beträgt beim Messvorgang etwa $600\ \mu\text{A}$, im Standby-Modus ist die Stromstärke nur $0,7\ \mu\text{A}$ groß (vgl. [1], S. 2f).

Für die LEDs ist eine Spannung von etwa $3,3\ \text{V}$ erforderlich. Die Stromstärke ist einstellbar; schon mit wenigen mA lassen sich gute Messergebnisse erzielen. Bei der Energiebetrachtung muss berücksichtigt werden, dass diese LEDs wegen der PWM nicht dauerhaft leuchten. (vgl. [1], S. 2f).

Schauen wir auf die Anschlüsse des GY-30102-Moduls, sehen wir, dass es mit einer einzigen Spannung (zwischen 3 und $5\ \text{V}$) betrieben wird. Neben dem MAX30102-IC befinden sich auf dem Modul zusätzliche Bauteile, die u. A. für die nötigen Spannungen von $1,8\ \text{V}$ und $3,3\ \text{V}$ sorgen. In Abb. 3 ist eine entsprechende Schaltung für den MAX30100 zu sehen.

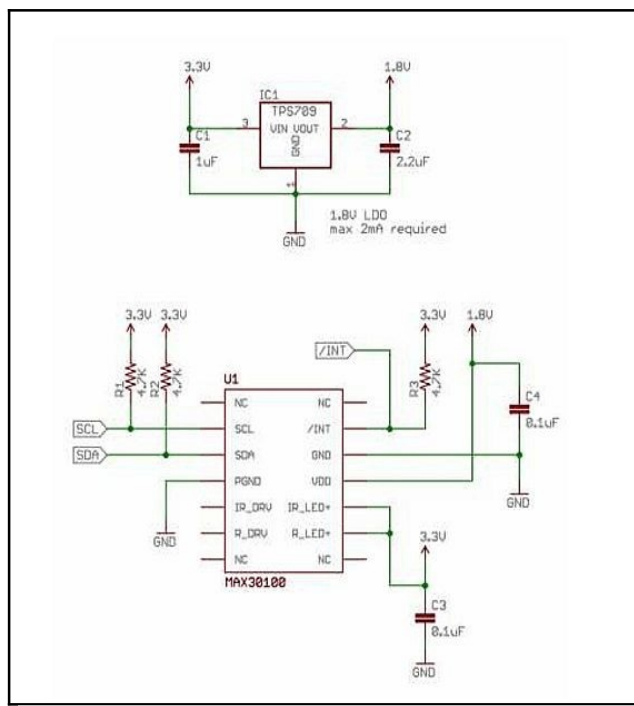


Abb. 3

Das MAX30102-Modul lässt sich dadurch leicht mit dem TTGO T-Display betreiben (s. Abb. 4). Zur Stromversorgung greifen wir auf den 3V -Anschluss des TTGO zurück. Daten zwischen dem TTGO und dem MAX30102-IC (z. B. für das Einstellen LED-Pulsweite oder zum Empfangen von Messwerten) werden per **I2C-Protokoll** übertragen. Für die I2C-Kommunikation benutzen wir hier die Pins 25 (SDA) und 26 (SCL).

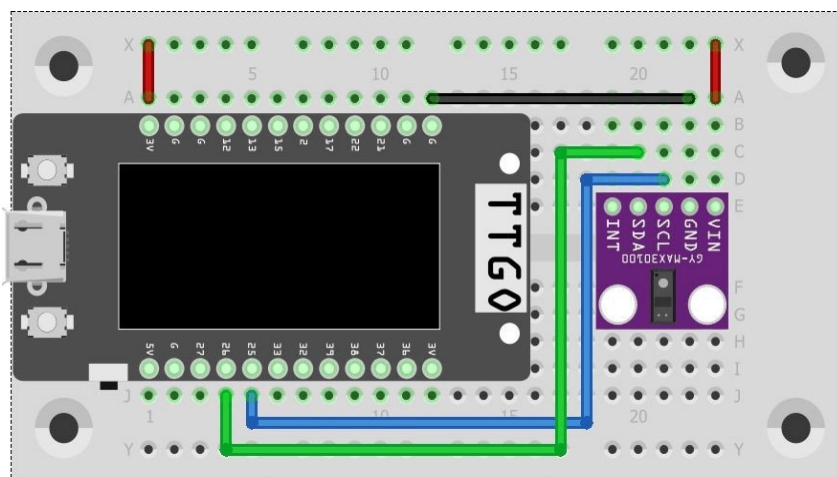


Abb. 4

2 Erste Schritte

In diesem Kapitel schauen wir uns an, wie man die LEDs des MAX30102 steuert. Die Vorgehensweise ist exemplarisch für die gesamte Kommunikation mit dem MAX30102. Dieser besitzt eine Reihe von 8-Bit-Registern. Die Inhalte dieser Register können über eine I2C-Schnittstelle beschrieben oder auch gelesen werden. Sämtliche Register werden im Datenblatt [1] ab S. 10 beschrieben.

Die I2C-Adresse wird im Datenblatt auf S. 5 angegeben:

- AE (hex) für einen Schreibvorgang
- AF (hex) für einen Lesevorgang

Die I2C-Funktionen von Micropython arbeiten mit einer einzigen 7-Bit-Adresse; je nachdem ob es um einen Schreibvorgang oder einen Lesevorgang geht, hängen sie automatisch an die 7-Bit-Adresse ein 0-Bit oder ein 1-Bit an. Die Micropython-7-Bit-Adresse erhalten wir, indem wir in der 8-Bit-Darstellung von AE (hex) = 1010 1110 (bin) die letzte Stelle streichen: 101 0111 (bin) = 57 (hex).

Das können wir leicht mit einem einfachen I2C-Scan-Programm überprüfen:

```
# I2C-Scanner mit Display
# Adressen OHNE R/W-Bit

from machine import Pin, SPI, I2C
import vga1_8x8 as font1
import st7789

spi = SPI(1, baudrate=20000000, 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)
display.init()
display.fill(0) # löschen; Bildschirm schwarz
display.text(font1, 'I2C-Bus scannen...', 10, 10)

i2c = I2C(1, scl=Pin(25), sda=Pin(26))
devices = i2c.scan() # liefert Liste mit I2C-Adressen (7 Bit, ohne R/W-Bit)
if len(devices) == 0:
    display.text(font1, 'Keine I2C-Geraete gefunden!', 10, 30)
else:
    display.text(font1, 'I2C-Geraete gefunden: ' + str(len(devices)), 10, 30)
    zeile = 50
    for adr in devices:
        display.text(font1, str(adr)+' dec ' + hex(adr) + ' hex ', 10, zeile)
        zeile = zeile + 10 # y-Koordinate der nächsten Zeile
```

Unser Programm liefert die Ergebnisse 87 (dec) bzw. 57 (hex).

Nebenbei haben wir auch gezeigt, wie das benötigte i2c-Objekt mit den im Kapitel 1 angegebenen Pin-Adressen instanziiert wird.

Mode Configuration (0x09)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
Mode Configuration	SHDN	RESET				MODE[2:0]			0x09	0x00	R/W

MODE[2:0]	MODE	ACTIVE LED CHANNELS
000	Do not use	
001	Do not use	
010	Heart Rate mode	Red only
011	SpO ₂ mode	Red and IR
100–110	Do not use	
111	Multi-LED mode	Red and IR

Abb. 5: Register 0x09 ([1], S. 18)

Kommen wir nun zum Ein- und Ausschalten einer LED, z.B. der roten LED (RED). Hierzu müssen wir auf die Register 0x09, 0x0A und 0x0C zugreifen:

Mit dem **Register 0x09 (Mode Configuration)** können wir über die Bits B2 - B0 einstellen, welche LEDs benutzt werden sollen. Da nur die rote LED benutzt werden soll, erhält [B2:B0] den binären Wert 011₂. Die restlichen Bits bleiben auf 0. Dem Register 0x09 muss also der Wert 0x02 zugewiesen werden.

Für den Umgang mit solchen Registern stellt micropython die i2c-Methoden **writeto_mem** und **readfrom_mem** zur Verfügung. In unserem Fall lautet der Schreibbefehl:

```
i2c.writeto_mem(0x57, 0x09, b'\02')
```

Der erste Parameter gibt die i2c-Adresse (7 Bit) an, der zweite die Register-Nummer, der dritte den Wert, der in dieses Register geschrieben werden soll. **Beachten Sie, dass die ersten beiden Parameter als Zahl (dezimal oder hexadezimal) anzugeben sind, während der dritte Parameter vom Typ Bytes sein muss.**

Mit dem **Register 0x0A (SpO₂ Configuration)** können wir mit den Bits B1 und B0 die Pulsweite bei der LED einstellen (s. Abb. 6); die zugehörige Programmzeile für eine Pulsweite von 118 µs lautet:

```
i2c.writeto_mem(0x57, 0x0A, b'\01') # 118 us
```

SpO₂ Configuration (0x0A)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
SpO ₂ Configuration		SPO2_ADC_RGE[1:0]	SPO2_SR[2:0]			LED_PW[1:0]			0x0A	0x00	R/W

LED_PW[1:0]	PULSE WIDTH (μs)	ADC RESOLUTION (bits)
00	69 (68.95)	15
01	118 (117.78)	16
10	215 (215.44)	17
11	411 (410.75)	18

Abb.6: Register 0x0A ([1], S. 18f)

Über die Bits B1 und B0 wird auch gleichzeitig die Auflösung der Messwerte vom Lichtsensor festgelegt; diese spielt für die Steuerung der LED natürlich keine Rolle.

Nun müssen wir noch die LED-Stromstärke festlegen. Dies geschieht mit Hilfe des **Registers 0x0C (LED Pulse Amplitude)**:

LED Pulse Amplitude (0x0C–0x0D)

REGISTER	B7	B6	B5	B4	B3	B2	B1	B0	REG ADDR	POR STATE	R/W
LED Pulse Amplitude	LED1_PA[7:0]								0x0C	0x00	R/W
	LED2_PA[7:0]								0x0D	0x00	R/W

These bits set the current level of each LED as shown in [Table 8](#).

Table 8. LED Current Control

LEDx_PA [7:0], RED_PA[7:0], or IR_PA[7:0]	TYPICAL LED CURRENT (mA)*
0x00h	0.0
0x01h	0.2
0x02h	0.4
...	...
0x0Fh	3.0

Abb.7: Register 0x0C ([1], S. 18f)

Wir wählen die Stromstärke 3 mA; der zugehörige Befehl lautet:

```
i2c.writeto_mem(0x57, 0x0C, b'\0F') # 3 mA
```

Das folgende Programm lässt die rote LED im 1,5-Sekunden-Takt mit 3 mA blinken; dabei ist die Pulsweite 118 μ s. Zur besseren Lesbarkeit werden hier die Parameter zu Beginn in Variablen gespeichert.

```
# MAX_30102_RED_LED_blinken_ohne_Display.py
# zuletzt getestet am 13.03.2025: Rote LED blinkt

from machine import Pin, I2C
from time import sleep

print('RED LED beim MAX 30102 ein- und ausschalten')

i2c = I2C(1, scl=Pin(25), sda=Pin(26))
i2c_addr = 0x57 # 7-Bit-Adresse (OHNE R/W-Bit)

mode_addr = 0x09 # Mode-Configuration-Adresse (u. A. RED ONLY)
spo2_addr = 0x0A # SP02-Adresse (u. A. PW)
LED1_addr = 0x0C # LED1-Register (Stromstärke für RED LED)

red_only = b'\x02'
puls_width_118_us = b'\x03'
current_3_mA = b'\x0F'
current_0_mA = b'\x00'

i2c.writeto_mem(i2c_addr, mode_addr, red_only) # RED ONLY (unbedingt erforderlich, sonst bleiben LEDs aus!!!)
i2c.writeto_mem(i2c_addr, spo2_addr, puls_width_118_us) # Pulsweite 118 us (deutlich heller als bei 69 us)

print('Anhalten mit Strg-C')
print()

print('Stromstärke (Rohwert):')
pause = 1.5 # Dauer der einzelnen Phasen (in s)
while True:
    i2c.writeto_mem(i2c_addr, LED1_addr, current_3_mA)
    print('I = ', current_3_mA)
    sleep(pause)
    i2c.writeto_mem(i2c_addr, LED1_addr, current_0_mA)
    print('I = ', current_0_mA)
    sleep(pause)
    print()
```

Dieses Programm finden Sie im Ordner `MAX30102_Materialien` unter dem Dateinamen `MAX_30102_RED_LED_blinken_ohne_Display.py`. Dort finden Sie auch eine Version dieses Programm, bei der die Stromstärkewerte auf dem Display des TTGO angezeigt werden.

Dieses Programm kann man jetzt recht einfach auch so erweitern, dass die Stromstärke in Schritten von 0,2 mA langsam ansteigt.

Kann man die IR-LED auf die gleiche Weise testen? Auch wenn das menschliche Auge kein IR-Licht wahrnehmen kann, lässt es sich aber häufig mittels einer Handy- oder auch einer Digitalkamera sichtbar machen. Bei meinen Experimenten habe ich damit leider kein Leuchten bei der IR-LED feststellen können – selbst bei hohen Stromstärken und großen Pulsweiten. Bei IR-Fernbedienungen war das hingegen möglich gewesen. Zunächst dachte ich, dass die IR-LED vielleicht defekt wäre oder nicht mit Strom versorgt würde. Versuche zeigten aber, dass das MAX30102-Modul mehr Strom aufnimmt, wenn die IR-LED eingeschaltet wird. In einem weiteren Versuch habe ich die rote LED ausgeschalt, die IR-LED eingeschaltet und das Umgebungslicht ausgeblendet. Mit dem Sensor des MAX30102 konnte dann das von einem Blatt Papier reflektierte IR-Licht nachgewiesen werden. Die Wellenlänge des IR-Lichts bei RC-5-Fernbedienungen liegt bei etwa 940 – 950 nm, unterscheidet sich also kaum von der Wellenlänge der IR-LED des MAX30102. Vermutlich ist wegen der PWM die mittlere Leistung beim MAX30102 doch zu klein für einen Nachweis mit Digitalkameras.

Zum Abschluss dieses Kapitels soll noch gezeigt werden, wie man die Register des MAX30102 auslesen kann. Für den aktuellen Wert der Stromstärke bei der roten LED benutzen wir den Befehl

```
value = i2c.readfrom_mem(i2c_addr, LED1_addr, 1)
```

Dabei gibt der dritte Parameter an, wie viele Bytes aus dem Register gelesen werden sollen (beginnend mit der aktuellen Adresse). Bitte beachten Sie, dass der Rückgabewert vom Typ Bytes ist. Für ein weiteres numerisches Bearbeiten muss er entsprechend umgewandelt werden (Siehe meinen Beitrag "Zahlen, Zeichenketten und Byte-Strings"; Sie finden ihn in meinem Forum unter <https://www.forum.g-heinrichs.de/viewtopic.php?f=18&t=204>.)

3 Sensorwerte auslesen

Der Sensor des MAX30102 kann sowohl sichtbares als auch IR-Licht messen. Wie viele Messungen pro Sekunde (**Sample Rate**) durchgeführt werden, kann über das Register **REG_SP02_CONFIG (0x0A)** eingestellt werden. Über dieses Register werden zusätzlich auch die Pulsweite der LEDs, die Auflösung des Messwertes als auch der Bereich des Analog-Digital-Wandlers festgelegt: Wir wählen als Konfigurationswert **spo2_config = b'\x27'**; dies entspricht dem Bitmuster 0b 0 01 001 11. Das bedeutet:

- **SP02_ADC range** = 4096 nA (Analog-Digital-Wandler-Bereich)
- **SP02 sample rate** = 100 Hz
- **Auflösung** = 18 bits und **Pulsweite** = 411 us

Der zugehörige Befehl lautet:

```
i2c.writeto_mem(i2c_addr, REG_SP02_CONFIG, spo2_config)
```

Die Messwerte (Samples) werden automatisch in einem **FIFO-Speicher** abgelegt. Jedes Sample besteht aus 3 Bytes für den RED Channel und 3 Bytes für den IR Channel; die getrennte Messung wird dadurch ermöglicht, dass die rote LED und die IR-LED nicht gleichzeitig gepulst werden. Insgesamt kann der FIFO-Speicher bis zu 32 Samples speichern. Für den Zugriff auf den FIFO kommen zwei Pointer zum Einsatz: ein **Write Pointer** (Register REG_FIFO_WR_PTR = 0x04) und ein **Read Pointer** (Register REG_FIFO_RD_PTR = 0x06). Diese werden bei einem Schreib- bzw. Lesezugriff jeweils automatisch inkrementiert. Diese Pointer sollten gelöscht werden, wenn man den SpO₂- oder HR-Modus aktiviert; dadurch liegen keine alten Daten mehr im FIFO vor:

```
i2c.writeto_mem(i2c_addr, REG_FIFO_WR_PTR, b'\x00')  
i2c.writeto_mem(i2c_addr, REG_FIFO_RD_PTR, b'\x00')
```

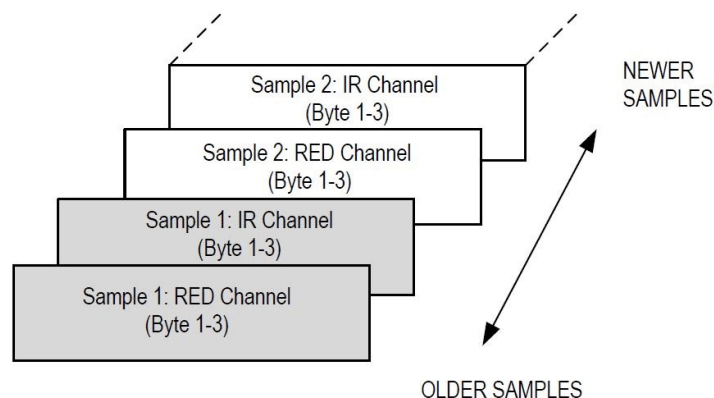


Abb. 8: FIFO ([1], Abb. 2)

Wenn der FIFO voll ist, dann werden keine weiteren Samples mehr in den FIFO geschoben. Die Anzahl der dadurch verlorenen Samples wird in dem **Over-Flow-Counter-Register** (REG_OVF_COUNTER = 0x05) gespeichert. Auch dieser Wert sollte bei der Aktivierung des SpO₂- oder HR-Modus auf 0 gesetzt werden:

```
i2c.writeto_mem(i2c_addr, REG_OVF_COUNTER, b'\x00')
```

Damit der FIFO nicht zu schnell voll wird, kann man den Datendurchfluss verringern, indem man benachbarte Samples eines Kanals (Channel) mittelt. Wie viele Daten jeweils gemittelt werden, kann man mit den Bits B7 - B5 des **FIFO Configuration Registers** (REG_FIFO_CONFIG = 0x08) einstellen. Wir stellen diesen Wert auf 0b010 ein; dadurch werden jeweils vier Messwerte zu einem einzigen gemittelt (vgl. Tabelle 3 auf S. 17 von [1]). Mit dem Bit B4 desselben Registers sorgen wir dafür, dass es zu keinem **FIFO-Rollover** kommt, indem wir dieses Bit auf 0 setzen. Über die nächsten 4 Bits (B3 - B0) können wir die Anzahl der Samples bestimmen, bei der ein Interrupt ausgelöst wird. Wir setzen diesen Wert auf 15 = 0b1111. Damit erhält das REG_FIFO_REGISTER den Wert

```
fifo_config = 0b010 0 1111 = b'\x4F'
```

Die zugehörige Programmzeile lautet dann:

```
i2c.writeto_mem(i2c_addr, REG_FIFO_CONFIG, fifo_config)
```

Wie kann man im Programm auf die Interrupts reagieren? Interrupts können durch Flags in den Interrupt-Registern "**Interrupt Status 1**" (REG_INTR_STATUS_1 = 0x00) und "**Interrupt Status 2**" (REG_INTR_STATUS_2 = 0x01) festgehalten werden. Dazu müssen die Interrupt-Flags zuvor freigeschaltet werden. Dies geschieht mit Hilfe der Register **REG_INTR_ENABLE_1** (0x02) und **REG_INTR_ENABLE_2** (0x03) durch die Befehle

```
i2c.writeto_mem(i2c_addr, REG_INTR_ENABLE_1, int_enable_1)
i2c.writeto_mem(i2c_addr, REG_INTR_ENABLE_2, int_enable_2)
```

Dabei sind nach [1], S. 13:

```
int_enable_1 = b'\xc0' # 0xC0 = 0b1100 0000 (A_FULL_EN u. PPG_RDY_EN)
int_enable_2 = b'\x02' # 0x02 = 0b0000 0010 (für Temperaturmessungen)
```

Mit dem Bit B6 des Interrupt-Status-Registers-1 kann z. B. überprüft werden, ob ein neues Sample im FIFO-Data-Register vorliegt (vgl. [1], S. 12).

Um den FIFO auszulesen, bestimmen wir zunächst die Anzahl der zur Verfügung stehenden Sample-Paare (3 Bytes für RED und 3 Bytes für IR) bestimmen. Dies geschieht mit der folgenden Funktion `get_data_present()`:

```
def get_data_present():
    read_ptr = i2c.readfrom_mem(i2c_addr, REG_FIFO_RD_PTR, 1)
    read_ptr = int.from_bytes(read_ptr, 'big') # bytes -> num
    # print('RD_PTR', read_ptr)
    # read_ptr wird bei jedem Lese-Vorgang inkrementiert,
    # dabei kann ggf. ein Überlauf erfolgen
    write_ptr = i2c.readfrom_mem(i2c_addr, REG_FIFO_WR_PTR, 1)
    write_ptr = int.from_bytes(write_ptr, 'big')
    # print('WR_PTR', write_ptr)
    # write_ptr wird bei jedem Schreib-Vorgang inkrementiert,
    # dabei kann ggf. ein Überlauf erfolgen
    if read_ptr == write_ptr:
        return 0
    else:
        num_samples = write_ptr - read_ptr
        # pointer wrap around berücksichtigen:
        if num_samples < 0:
            num_samples += 32
        return num_samples
```

Die Funktionsweise wird durch entsprechende Kommentare erläutert. Zwischenergebnisse können ggf. mit Hilfe der (hier auskommentierten) print-Funktionen angezeigt werden.

Mit der folgenden Funktion **read_fifo()** können wir ein einziges Sample-Paar lesen:

```
def read_fifo(): # ein einziges Sample-Paar aus dem FIFO-Puffer lesen
    red_led = None
    ir_led = None

    # jeweils 1 Byte aus REG_INTR_STATUS_1 und REG_INTR_STATUS_2 lesen,
    # dadurch werden diese Interrupt-Register gelöscht
    reg_INTR1 = i2c.readfrom_mem(i2c_addr, REG_INTR_STATUS_1, 1)
    reg_INTR2 = i2c.readfrom_mem(i2c_addr, REG_INTR_STATUS_2, 1)

    # 6 Bytes (3 für RED-LED und 3 für IR-LED) aus dem FIFO lesen
    d = list(i2c.readfrom_mem(i2c_addr, REG_FIFO_DATA, 6))
    # print('Byte-Liste für 1 Sample', d)

    # aus jeweils 3 Bytes die zugehörige 3-Byte-Zahl formen;
    # nur die unteren 18 Bits benutzen (MSB ist in Bit 17),
    # also mit 0x03FFFF maskieren (vgl. Datasheet S. 15)
    # Die Listenelemente d[0], d[1], ... sind Zahlen!
    red_led = (d[0] << 16 | d[1] << 8 | d[2]) & 0x03FFFF
    ir_led = (d[3] << 16 | d[4] << 8 | d[5]) & 0x03FFFF

    return red_led, ir_led
```

4 Die Klasse MAX30102

Es bietet sich an, die Befehle und Funktionen aus den Kapiteln 2 und 3 in einer Klasse MAX30102 zusammenzufassen. Man findet die Definition dieser Klasse mitsamt den erforderlichen Importen in der Datei **max30102.py**.

Wir wollen diese Datei hier nicht im einzelnen erörtern, sondern nur auf einige Details näher eingehen.

Instanziierung

```
def __init__(self, address=0x57, scl_Pin_nr=25, sda_Pin_nr=26):
    print('I2C-Adresse:', address)
    self.address = address
    self.i2c = I2C(1, scl=Pin(scl_Pin_nr), sda=Pin(sda_Pin_nr))

    self.reset()
    sleep(1) # warte 1 s
    reg_data = self.i2c.readfrom_mem(self.address, REG_INTR_STATUS_1, 1)
    print('Interrupt Status 1:', bin(int.from_bytes(reg_data, 'big')))

    print('Der Wert ist 0b1, wenn zuvor ein Power-Up stattgefunden hat,
                                                sonst 0b0.')
```

Import der MAX30102-Klasse, Instanziierung und Festlegung aller im Kapitel 3 beschriebenen **Konfigurationen** erfolgen dann mit den folgenden Zeilen:

```
from max30102 import MAX30102
max30102 = MAX30102()
max30102.setup(b'\x02')
```

Beachten Sie, dass die I2C-Adresse nunmehr eine Eigenschaft der Instanz **max30102** ist. Der Setup-Parameter **b'\x02'** sorgt dafür, dass nur die rote LED zum Einsatz kommen soll (s. o., S. 6).

Die Instanz **max30102** besitzt noch **weitere Methoden**, mit denen verschiedene Einstellungen einzeln vorgenommen werden können. Außerdem stellt sie auch die Methode **get_temp** zur Verfügung; damit kann die Temperatur des MAX30102-IC ermittelt werden (s. Datasheet S. 22).

An dem einfachen Beispiel der Temperaturmessung wollen wir zeigen, wie man mit der MAX30102-Klasse umgeht. Das Programm besteht im Wesentlichen aus

- den Importen für die Klasse MAX30102 und die Funktion sleep,
- der Instanziierung eines max30102-Objekts,
- einer Endlos-Schleife, in welcher die Temperatur mit Hilfe der get_temp-Methode ermittelt und ausgegeben wird.

Beachten Sie: Das Modul max30102.py für die Klasse MAX30102 muss im Flash des Mikrocontrollers gespeichert sein.

Das Programm MAX30102_Temperatur.py:

```
from time import sleep
from max30102 import MAX30102

# Instanziierung...
max30102 = MAX30102()
print()

while True:
    t = max30102.get_temp()
    print('Temperatur des MAX30102:', t, '°C')
    sleep(2)
```

Wir stellen den Aufbau aus Abb. 2 für ein paar Minuten in einen Kühlschrank und schließen ihn anschließend an unseren PC an. Dann starten wir das Programm; im Terminal-Bereich von Thonny wird nun folgendes angezeigt:

```
I2C-Adresse: 87
MAX30102 resettet
Interrupt Status 1: 0b0
Der Wert ist 0b1, wenn zuvor ein Power-Up stattgefunden hat, sonst 0b0.
```

```
Temperatur des MAX30102: 16.1875 °C
Temperatur des MAX30102: 16.5625 °C
Temperatur des MAX30102: 16.625 °C
Temperatur des MAX30102: 16.75 °C
Temperatur des MAX30102: 17.0625 °C
Temperatur des MAX30102: 17.25 °C
Temperatur des MAX30102: 17.4375 °C
```

Bitte beachten Sie: Auch wenn der MAX30102-IC vier Bits für die Nachkommastellen bereitstellt, bedeutet dies nicht, dass eine entsprechend hohe Genauigkeit garantiert wird. Diese beträgt nach den Angaben auf des Datenblatts ([1], S. 4) 1 °C.

5 Ein erstes Programm zur Pulsrate

Unser erstes Programm wird die vom Sensor aufgenommenen Signale auf dem Display des TTGO anzeigen. Wenn der Graph an den rechten Rand kommt, wird die Anzeige gelöscht und die neuen Messwerte werden ausgegeben. Der Vorgang kann mit dem Taster 0 (in der Abb. 9 links oben) beendet werden. Neben dem Signalverlauf werden auch Zeit-Markierungen in Form von roten senkrechten Linien eingezeichnet; diese kennzeichnen einen Zeitabstand von 1 s. Damit kann man recht gut den Zeitraum für eine bestimmte Anzahl von Maxima bestimmen.

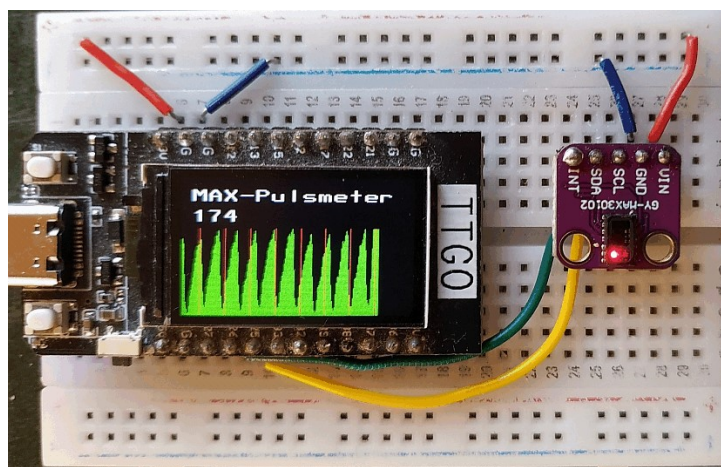


Abb. 9

Importe und Instanziierungen

```
mw_skal = 5 # Vorgabewert: 5, je nach Haut, Blutdruck in den Kapillaren
               und Anpressdruck des Fingers zwischen 2 und 20

from machine import SPI, Pin
import vga1_16x16 as font1
import st7789
from time import sleep, ticks_ms
from max30102 import MAX30102

taster = Pin(0, Pin.IN, Pin.PULL_UP)

# Instanziierung von display...
spi = SPI(1, baudrate=20000000, 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)
display.init()
display.fill(0) # loeschen; Bildschirm schwarz
display.text(font1, 'MAX-Pulsmeter', 10, 10)
```

```
# Instanziierung und setup von max30102
max30102 = MAX30102()
max30102.setup(b'\x02') # Nur Red-LED; Default ist b'\x03' (Red & IR)
```

Erster Mittelwert

Das uns interessierende AC-Signal in Abb. 2 ist sehr gering gegenüber dem DC-Signal. Das DC-Signal eliminieren wir folgendermaßen: Bevor Messwerte zur Anzeige kommen, wird das Gesamtsignal 50 (samp_width) mal gemessen; die Rohwerte werden in einer Liste samp_FIFO gespeichert. Von diesen wird schließlich ein Mittelwert (mean_value) berechnet und in der Variablen mean_value gespeichert.

```
samp_width = 50
samp_FIFO = []
mean_value = 0
i = 0
while i < samp_width:
    num_samples = max30102.get_data_present()
    if num_samples > 0:
        red_led, ir_led = max30102.read_fifo()
        samp_FIFO.append(red_led)
        i += 1
    mean_value = mean_value + red_led
mean_value = mean_value / samp_width
```

Messwerte aufzeichnen und graphisch darstellen

Dieser Mittelwert wird während der nun folgenden Messungen stets aktualisiert (gleitender Mittelwert, vgl. [4]).

```
# Parameter für die Grafik und Startwerte
x_min = 0
x_max = 239
x = x_min
y_alt = 134
t0 = ticks_ms() # für vertikale Zeitmarken

# Messen und darstellen
do_again = True # wird ggf. durch Taster T0 auf False gesetzt
while do_again:
    num_samples = max30102.get_data_present()
    if num_samples > 0:
        red_led, ir_led = max30102.read_fifo()
        value_in = red_led
        value_out = samp_FIFO[samp_width-1]
        for i in range(1, samp_width):
            samp_FIFO[samp_width-i] = samp_FIFO[samp_width-i-1]
```

```
samp_FIFO[0] = value_in
mean_value = mean_value + (value_in - value_out) / samp_width
# Mittelwert subtrahieren, Skalierung für Display...
mw = int((value_in - mean_value)/mw_skal) + 50
display.text(font1, str(mw) + '          ', 10, 30)
y = max(134-mw, 50)
display.line(x,y_alt,x,y,st7789.GREEN)

# Zeit-Raster...
t1 = ticks_ms()
if t1 >= t0 + 1000 : # 1,0 s später
    t0 = t1
    display.line(x,50,x,134,st7789.RED)

# nächste Messung vorbereiten, ggf. Display löschen...
x += 1
if x > x_max:
    x = x_min
    display.fill(0) # loeschen; Bildschirm schwarz
    display.text(font1, 'MAX-Pulsmeter', 10, 10)
do_again = taster.value() # False, wenn Taster 0 gedrückt

print('Ende der Messung...')
```

Das gesamte Programm finden Sie in der Datei **HR_1.py**.

Die Messung zum Foto aus Abb. 9 wurde mit `mw_skal = 10` durchgeführt; hier war bei mir nach einem strammen Spaziergang die Durchblutung etwas größer als normal. Von dem Maximum bei der ersten roten Linie bis zum Maximum direkt links von der 7. roten Linie verstreichen etwa 5, 8 s. In dieser Zeitspanne gibt es 8 Pulse; das entspricht etwa 1,38 Pulsen pro s bzw. 82,8 bpm.

Im nächsten Kapitel werde ich zeigen, wie das Programm so erweitert werden kann, dass es aus den gesammelten Daten die Pulsrate selbst bestimmt.

6 Programm zur Bestimmung der Pulsrate

Bei diesem Programm werden zunächst auf dieselbe Weise wie bei dem Programm **HR_1.py** die Messwerte bestimmt und auf dem Display angezeigt. Allerdings werden jetzt diese Messwerte mit den zugehörigen Zeiten in zwei Listen (`signal_data` bzw. `time_data`) gespeichert. Diese Listen werden direkt vor der Messwertschleife zunächst als leere Listen definiert:

```
...
do_again = True
signal_data = []
time_data = []

while do_again:
    num_samples = max30102.get_data_present()
    ...
```

Nach der Bestimmung eines y -Wertes für das HR-Diagramm

```
y = max(134-mw, 50)
display.line(x,y_alt,x,y,st7789.GREEN)
```

wird dieser mit seinem Zeitwert `t1` an die Liste `signal_data` bzw. `time_data` angehängt:

```
signal_data.append(y)

# Zeit-Raster...
t1 = ticks_ms()
time_data.append(t1)
```

Wenn der Graph (fast) vollständig dargestellt ist, betätigen wir den Taster 0; die Messwerterfassung wird dadurch beendet. Nun werden die Signal-Maxima des Graphen detektiert und die Zeitintervalle zwischen den Maximalstellen bestimmt. Der Mittelwert d dieser Zeitintervalle (gemessen in ms) liefert die **Herzfrequenz in bpm** mit der Rechnung

```
bpm = 60/(d/1000)
```

Dabei ist die Bestimmung der Maximalstellen allerdings etwas aufwendiger, als es auf den ersten Blick erscheint. Das liegt im Wesentlichen daran, dass unsere Messwerte "verrauscht" sind. Das idealisierte Signal (in Abb. 10 mit schwarz umrandeten Balken dargestellt) hat offensichtlich nur ein einziges Maximum (nämlich bei M_1). Hier führt das Kriterium

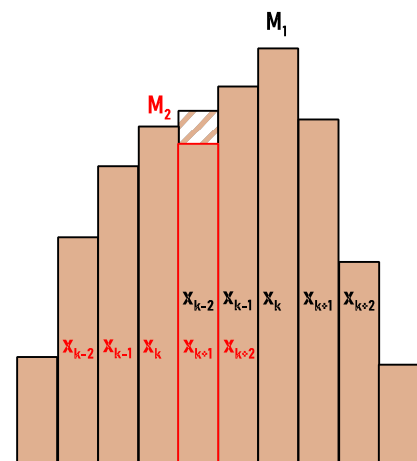


Abb. 10

$$f(x_{k-1}) < f(x_k) \text{ und } f(x_{k+1}) < f(x_k)$$

korrekt zur gesuchten Maximalstelle. Alle anderen Stellen erfüllen das Kriterium nicht. Ist der 5. Messwert allerdings durch Rauschen bedingt ein wenig kleiner (rot umrandeter Balken), so liefert unser Kriterium jetzt zwei Maxima, nämlich bei M_1 und M_2 .

Solche durch Ausreißer bedingte Maximalstellen müssen wir nach Möglichkeit ausschließen. Dass x_k eine Maximalstelle ist, ist schon deutlich sicherer, wenn wir fordern:

$$f(x_{k-2}) < f(x_k) \text{ und } f(x_{k-1}) < f(x_k) \text{ und } f(x_{k+1}) < f(x_k) \text{ und } f(x_{k+2}) < f(x_k).$$

Nach diesem Kriterium würde jetzt das Maximum bei M_2 verworfen werden. Noch sicherer ist es, wenn wir nicht nur 2, sondern 3, 4 oder auch mehr Stellen links und rechts von x_k in derselben Weise untersuchen. Allerdings kann diese Forderung ggf. so stark sein, dass kein einziger Wert x_k sie erfüllt. Es ist deswegen sinnvoll, diese Forderung abzuschwächen: Die gesuchten Stellen x_k sollen "fast alle" der obigen Bedingungen erfüllen.

Das folgende Programmteil erstellt auf der Grundlage dieser Strategie eine Liste von Kandidaten für die gesuchten Maximalstellen: Dabei benutzen wir der Anschaulichkeit halber im Folgenden statt der Zeitwerte die entsprechende x-Werte des Diagramms.

```
# Rohliste der Maximalstellen (x_max_list)...
w = 5 # Unschärfe bei der ersten Bestimmung der Maximalstelle
last_x = x # letzter x-Wert des Diagramms (entspricht letztem Zeitwert)
x_max_list = [] # Kandidaten für Maximalstellen (x-Werte im Diagramm)
for x in range(w, last_x-w):
    # zählen, wie häufig...
    count = 0
    for x_c in range(x-w, x+w+1):
        # x_c geht von x-w bis x+w; in der Mitte x; insgesamt 2*w+1 Werte
        if signal_data[x_c] < signal_data[x]: # < oder <= ???
            count += 1
    if count > 2*w - 2: # fast alle y-Werte unter dem Wert an der Stelle x_c
        x_max_list.append(x)
        t_max_list.append(time_data[x])
print('x_max_list ggf. mit mehrfachen x-Werten:', x_max_list,
      len(x_max_list))
```

Das Ergebnis einer solchen Rohliste kann z. B. so aussehen:

```
[14, 15, 33, 34, 51, 52, 70, 71, 89, 91, 108, 109, 127, 128, 146, 147, 165,
182, 183, 202, 203]
```

Auffällig ist, dass manche Elemente sehr nah beieinander liegen: Sie repräsentieren aber nur eine einzige Maximalstelle. Dass hier mehrere Werte in der Liste auftauchen, ist darauf zurückzuführen, dass wir das "weiche" Kriterium "fast alle..." benutzen. Diese nah beieinander liegenden Werte reduzieren wir mit den folgenden Programmzeilen sukzessive zu einem einzigen Wert.

```
# jetzt dicht beieinander liegende Werte zu einem Mittelwert zusammenfassen
delta_x = 10 # Unschärfe eines x-Werts bei Max-Stellen-Reduzierung
for i in range(3): # ggf. noch häufiger reduzieren
    x_max_list_reduced = []
    x = 0 # x-Wert im Diagramm (entspricht Zeit)
    while x <= len(x_max_list)-1:
        if x == len(x_max_list) - 1: # letztes Element
            x_max_list_reduced.append(x_max_list[x])
            x += 1
        elif abs(x_max_list[x] - x_max_list[x+1]) <= delta_x:
            x_max_list_reduced.append(int(round(x_max_list[x]+
                                                x_max_list[x+1])/2))
            x += 2
        else:
            x_max_list_reduced.append(x_max_list[x])
            x += 1
    x_max_list = x_max_list_reduced
```

Die reduzierte Liste unsere obigen Rohliste sieht dann so aus:

```
[14, 33, 51, 70, 90, 108, 127, 146, 165, 182, 202]
```

Mit dieser Liste `x_max_list` werden jetzt zunächst die Längen der einzelnen Zeitintervalle zwischen den Maxima berechnet:

```
diff_T = []
for i in range(len(x_max_list)-1):
    diff_T.append(time_data[x_max_list[i+1]] - time_data[x_max_list[i]])
```

Davon wird nun der Mittelwert berechnet:

```
m = 0
if len(diff_T) == 0:
    print('Kein Zeitintervall vorhanden - Abbruch')
    exit()
for i in range(len(diff_T)):
    m = m + diff_T[i]
d = m / len(diff_T) # mittlere Zeitspanne (in ms)
```

Zur Anzeige auf dem Display stellen wir die Herzfrequenz (in bpm) mit einer Nachkommastelle als Zeichenkette dar:

```
bpm = 60/(d/1000)
print('bpm:', bpm)
bpm = round(bpm,1)
bpm_str = str(bpm)
```

Und so berechnen wir den **Standardfehler** aus der Liste `diff_T`:

```
s = 0
n = len(diff_T)
for i in range(n):
    d = 60/(diff_T[i]/1000) - bpm
    print('d:', d)
    s = s + d*d
sigma_bpm = sqrt(s/((n-1)*n)) # in s
sigma_bpm = round(sigma_bpm,1) # auf 1 Stelle hinter dem Komma gerundet
print('sigma_bpm =', round(sigma_bpm), ' bpm')
sigma_bpm_str = str(sigma_bpm)
```

Mit den Zeichenketten bpm_str und sigma_bpm_str können wir schließlich das Ergebnis im Display oberhalb des Diagramms anzeigen lassen:

```
display.text(font1, bpm_str, 10, 30, st7789.RED) # 4*16 = 64 Pixel breit
display.text(font1, b'\xF1', 82, 30, st7789.RED) # 16 Pixel breit
display.text(font1, sigma_bpm_str + ' bpm', 104, 30, st7789.RED)
```

Dabei steht der Code b'\xF1' für das Zeichen “±”.

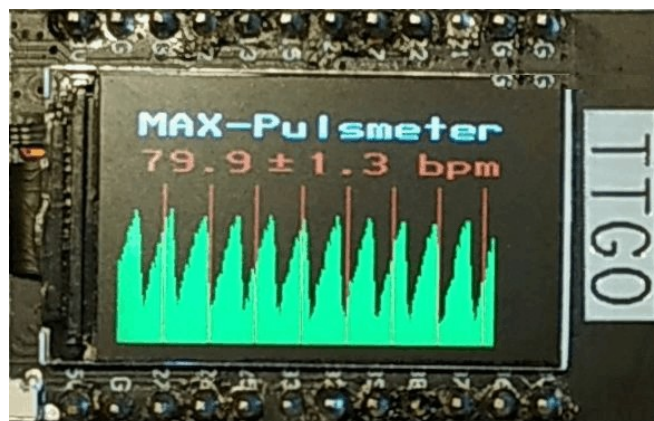


Abb. 11

Das zugehörige befindet sich in der Datei **HR_2d.py**. Beachten Sie: Die im Programm angegebenen print-Kommandos dienen nur zur Kontrolle von (Zwischen-) Ergebnissen; Sie können sie bei Bedarf aus dem Programm löschen.

7 Quellen

- [1] MAX30102 Datasheet
- [2] max3010x-ev-kits-recommended-configurations-and-operating-profiles.pdf
- [3] https://www.researchgate.net/figure/Schematic-sensor-oximetry-module-MAX30100_fig2_341731508
- [4] https://en.wikipedia.org/wiki/Moving_average
- [5] <https://github.com/doug-burrell/max30102>