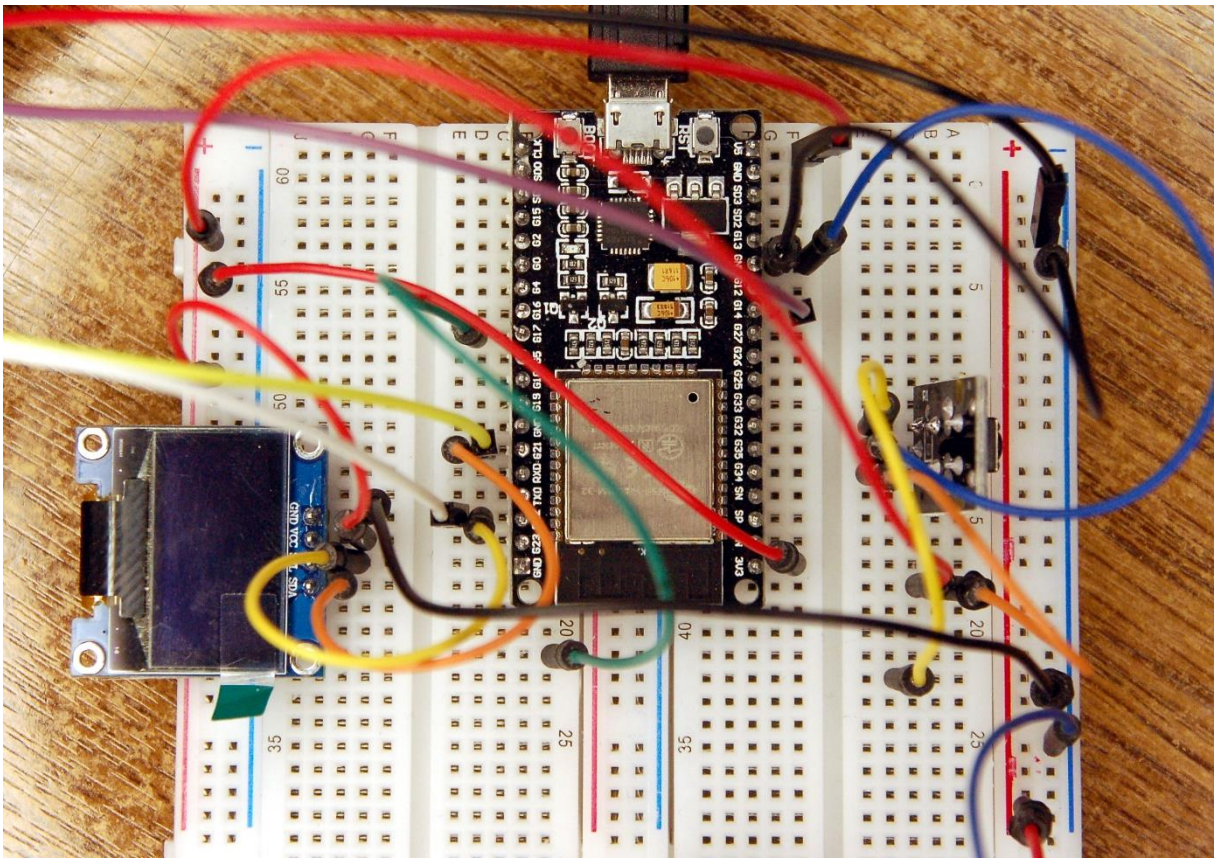
Die Schaltung zum Projekt

Der Beitrag ist auch als PDF-Dokument verfügbar.

Farberkennungssensoren haben als Wandler Fotodioden, die ihre Spannung an einen Analog-Digital-Wandler (aka ADC) weitergeben. Im Falle eines TCS3200 müssen vier ADC-Eingänge für rot, grün, blau und klar die Signale am Controller übernehmen.

Den Baustein, den ich heute unter die MicroPython-Lupe nehme, erledigt die Wandlung selbst und bietet dazu noch diverse Extras wie Verstärkungseinstellung (PGA = Programmable Gain Amplifier), Bereichsbegrenzung, Interruptausgang, der Bereichsüberschreitungen meldet, Integrationszeiteinstellung, Statemachine, Single- und Continuous Mode, schaltbare LED zur Beleuchtung und der TCS3472 bietet einen I2C-Bus, über den die Messwerte zum ESP32 gelangen. Neugierig geworden? Dann kommen Sie einfach mit auf eine Entdeckungstour mit

MicroPython auf dem ESP32 und ESP8266

heute

TCS3472 – Farberkennungssensor mit I2C

Kommen wir zur Hardware. Als Chef des Projekts setze ich einen ESP32 ein, ein ESP8266 tut es aber auch, brauche ich doch sonst nur noch ein

Farberkennungsmodul und ein OLED-Display. Aufgebaut wird die Schaltung wie immer auf einem Breadboard mit etlichen Jumperkabeln.

Hardware

1	ESP32 Dev Kit C unverlötet oder ESP32 Dev Kit C V4 unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board mit CP2102 oder NodeMCU-ESP-32S-Kit oder
1	NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F WIFI oder D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F WIFI
1	0,96 Zoll OLED SSD1306 Display I2C 128 x 64 Pixel
1	TCS34725 RGB Farb Sensor mit Infrarot-Filter
1	Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord kompatibel mit Arduino und Raspberry Pi - 1x Set

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware für den ESP8266:

v1.19.1 (2022-06-18) .bin

Verwendete Firmware für den ESP32:

v1.19.1 (2022-06-18) .bin

Die MicroPython-Programme zum Projekt:

ssd1306.py Hardwaretreiber für das OLED-Display
oled.py API für das OLED-Display
tcs3472.py Hardwaretreiber für das Farbmodul
colorcheck.py Demo-Programm

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung (english version). Darin gibt es auch eine Beschreibung, wie die Micropython-Firmware (Stand 18.06.2022) auf den ESP-Chip gebrannt wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, damit der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny,

µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm kompilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Makro fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen.

Autostart

Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP-Chip hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton, oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein kompiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA oder … enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer so, wie hier beschrieben.

Signale auf dem I2C-Bus

Wie eine Übertragung auf dem I2C-Bus abläuft und wie die Signalfolge ausschaut, das können Sie in meinem Beitrag mammutmatrix_2_ger.pdf nachlesen. Ich verwende dort ein interessantes kleines Tool, mit dem Sie die I2C-Bus-Signale auf Ihren PC holen und analysieren können.

Der TCS3472 Sensor und sein Modul

Anders als der TCS3200, der seine 4 Farbkanäle rot, grün, blau und klar als Analogspannungen abgibt, liefert der TCS3472 seine Information via I2C-Bus digital ab.

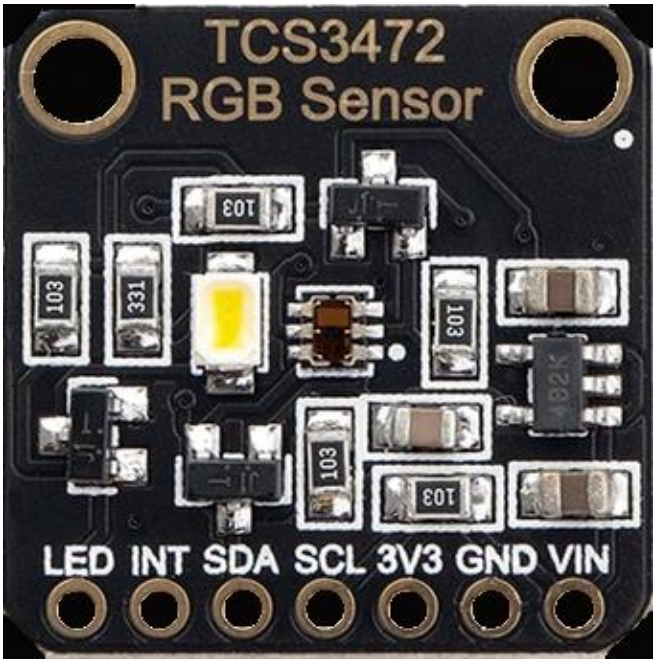

Abbildung 1: tcs3472

Außerdem lässt sich die LED über den Eingang **LED** schalten (high-aktiv). Anhand des englischen Datenblatts von OSRAM habe ich mein MicroPython-Modul zusammengestellt, dessen Methoden ich nachfolgend beschreiben werde.

Jetzt erst einmal die Schaltung für die Experimente.

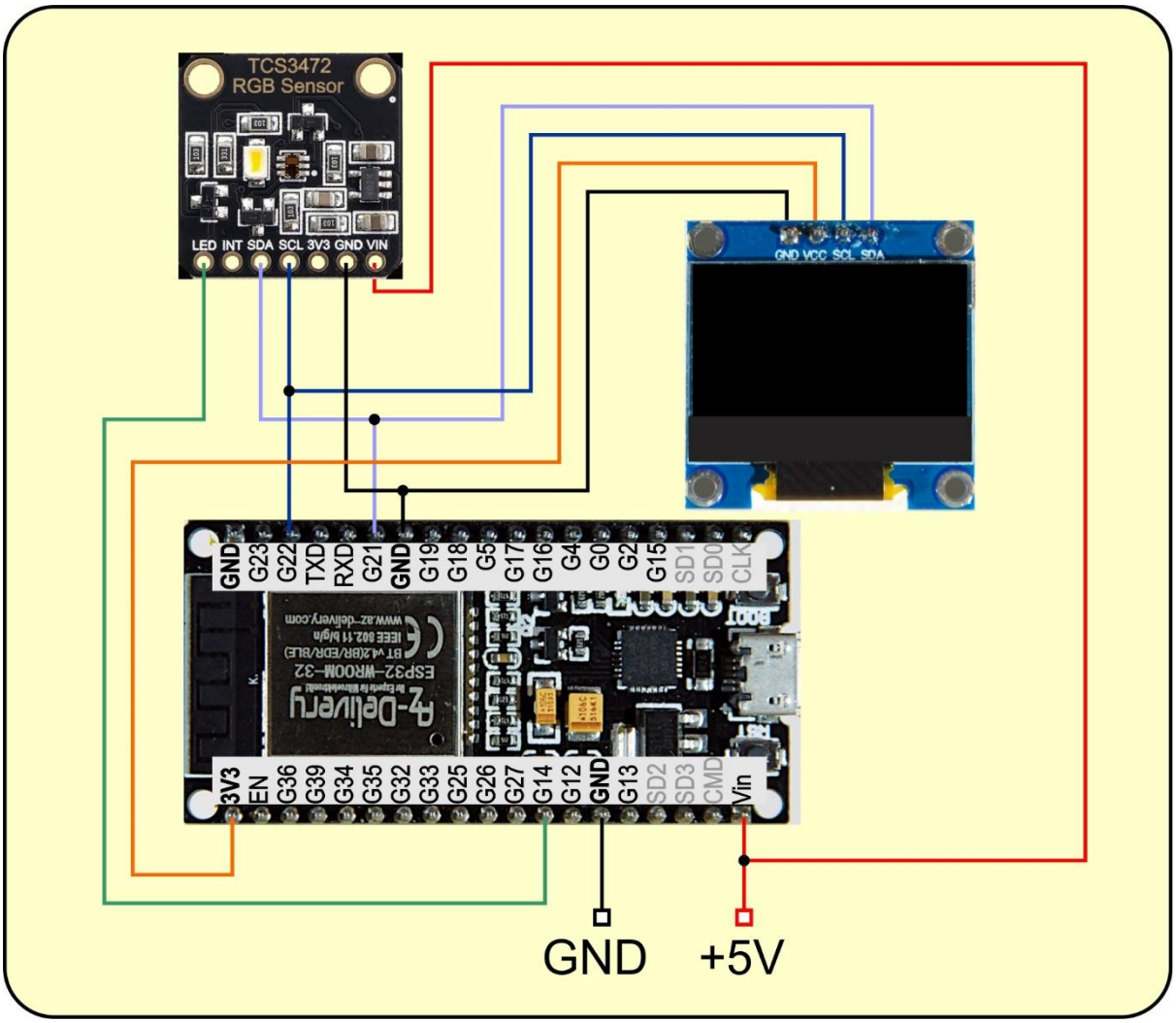

Abbildung 2: Schaltung mit ESP32

Im TCS3472 arbeitet eine Statemachine, die den Ablauf der einzelnen Arbeitsschritte steuert. Sie kann von außen an einigen Stellen beeinflusst werden. Die Signale der vier Farbkanäle gelangen über Verstärkerstufen an integrierende ADC-Wandler. Die in vier Stufen einstellbare Verstärkung sowie die Integrationszeit beeinflussen beide den Wert, der schließlich in den einzelnen Farbregistern landet. Das Berechnen der Werte für die Integrationszeit ist etwas exotisch, so wie mit der rechten Hand hinten um den Kopf am linken Ohr gekratzt. Ähnlich verhält es sich mit der Berechnung der Wartezeit zwischen einzelnen Messungen im Dauerlauf -Modus. Die folgende Abbildung der Statemachine findet sich im Datenblatt auf Seite 17.

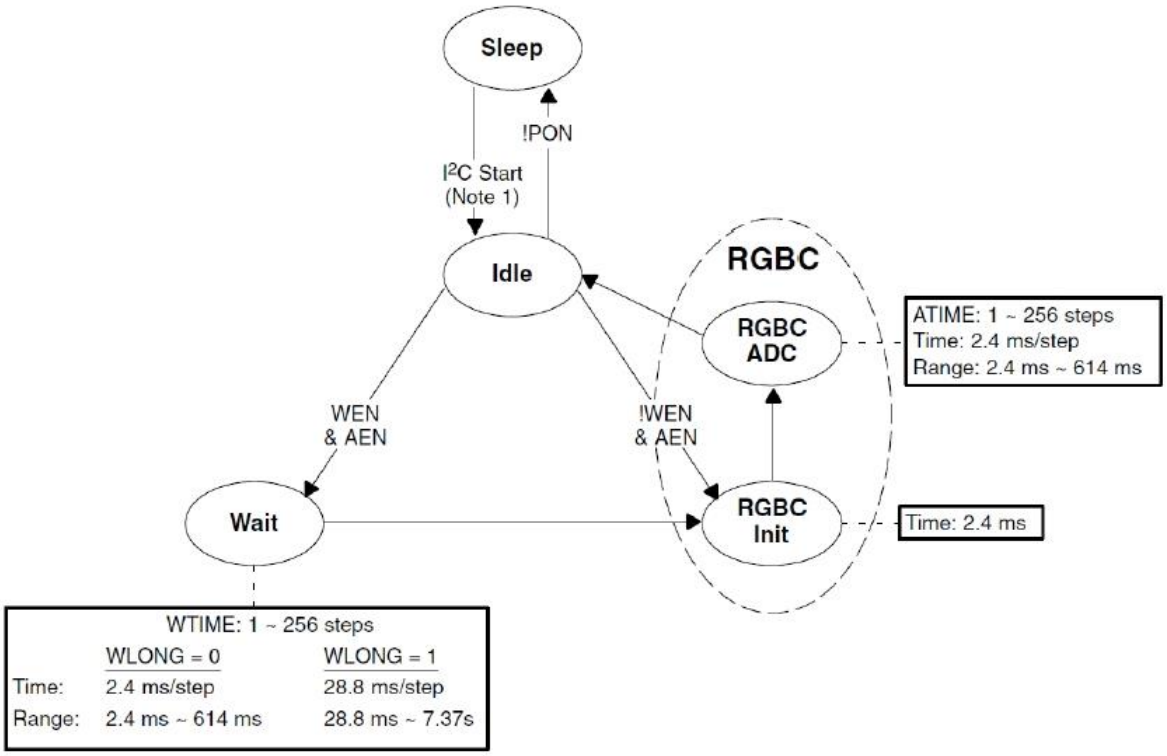

Abbildung 3: Diagramm der Statemachine

Auf die Berechnung der Werte für das **ATIME**-Register, durch welches die Integrationszeit eingestellt wird, gehe ich weiter unten ein.

Das Modul beginnt mit zwei Importen und einer langen Liste von Deklarationen der in der Klasse TCS3472 benutzten Namen für Register, Masken und so weiter.

```
# tcs3472.py
from time import sleep_ms
from machine import Pin

class TCS3472:
    HWADR  = const(0x29)

    ENABLE = const(0x00)
    ATIME  = const(0x01) # 256 - Integration Time / 2.4 ms
    WTIME  = const(0x03)
    AILTL  = const(0x04)
    AILTH  = const(0x05)
    AIHTL  = const(0x06)
    AIHTH  = const(0x07)
    PERS   = const(0x0c)
    CONFIG = const(0x0d)
    CONTROL= const(0x0f)
    ID     = const(0x12)
    STATUS = const(0x13)
    CLEARL = const(0x14)
```

```
    CLEARH = const(0x15)
    REDL   = const(0x16)
    REDH   = const(0x17)
    GREENL = const(0x18)
    GREENH = const(0x19)
    BLUEL  = const(0x1a)
    BLUEH  = const(0x1b)

    CMD    = const(0x80) # 1 SFunc; 0 RegAdr
    AUTOINC= const(0x20)
    TYPEMASK=const(0x60)
    SFUNC  = const(0x60)
    ADRMASK= const(0x1F)
    CCICLR = const(0x06)

    AIEN   = const(0x10) # ADC-IRQ enable
    WEN    = const(0x08) # Wait enabble
    AEN    = const(0x02) # ADC enable
    PON    = const(0x01) # Power on
    WLONG  = const(0x01) # 256 - Wait Time / 2.4ms

    GAINx1 = const(0x00)
    GAINx4 = const(0x01)
    GAINx16= const(0x02)
    GAINx60= const(0x03)
    GAIN=["x1","x4","x16","x60"]

    AINTMASK=const(1<<4)
    AVALIDMASK=const(0x01)

    PART = {0x44: "TCS34725",
            0x4d: "TCS34727" }

    PERSIST= {0:"every",
            1: 1,
            2: 2,
            3: 3,
            4: 5,
            5: 10,
            6: 15,
            7: 20,
            8: 25,
            9: 30,
            10: 35,
            11: 40,
            12: 45,
            13: 50,
            14: 55,
            15: 60}
```

Gleich am Anfang befindet sich die Sieben-Bit-Hardwareadresse des TCS3472, die 0x29. Einige Dictionaries (kurz Dict), werden für Klartext-Rückmeldungen von Registerinhalten eingesetzt.

Die Methode **__init__**() stellt den Konstruktor dar, der durch TCS3472() im übergeordneten Programm aufgerufen wird, um eine TCS3472-Instanz (aka -Objekt) zu erzeugen.

```
    def __init__(self, i2c, itime=100,
                 wtime=3, gain=GAINx1, led=14):
        self.i2c=i2c
        self.Itime=itime
        self.Atime=255
        self.Wtime=255
        self.Again=gain
        self.Wlong=0
        self.LED=Pin(led,Pin.OUT,value=0)
        self.LED.value(0)
        self.running=False
        self.wakeUp()
        print("Constructor of TCS3472-class")
        print("Integration Time",self.setAtime(itime),"ms")
        self.setWtime(wtime)
        print("Wait Time",self.getWait(),"ms")
        self.setGain(gain)
        print("Gain is",self.GAIN[self.getGain()])
        print("Chip is a ",self.getID())
```

Beim Aufruf muss ein I2C-Objekt an den Positionsparameter **i2c** übergeben werden. Dieses Objekt wird im Hauptprogramm entsprechend dem eingesetzten Controller erzeugt. Mehr dazu im Beispielprogramm. Alle weiteren Parameter sind optional und mit Defaultwerten vorbelegt.

Ich erzeuge einige Attribute auf die ich bei der Besprechung der jeweiligen Routinen eingehen werde. Das Pin-Objekt **LED** wird unter Verwendung der in **led** übergebenen Pinnummer erzeugt und auf GND-Potenzial gebracht. Das löscht die LED auf dem BOB (aka Break Out Board) des TCS3472.

Die Methode **wakeUp**() bringt den TCS3472 aus dem Sleepmodus in den Modus Idle, aus dem heraus Messungen gestartet werden. Die Argumente für die Integrationszeit, die Wartezeit und den Verstärkungsfaktor gehen an den TCS3472 und werden im Terminal ausgegeben. Die Methode **getID**() liefert den Typ des Chips. Ein TCS34725 verträgt bis zu 3,6V Betriebsspannung, der vorliegende TCS34727 dagegen nur 3,3V. Das muss uns aber nicht interessieren, weil auf dem Board ein Spannungsregler wohnt, der mit 3,3V bis 5V beschickt werden kann.

Es folgen vier Routinen, die zum Senden und Empfangen von Daten dienen und dem, im Datenblatt (Seite 19) dargestellten, I2C-Protokoll des TCS3472 gehorchen. Danach ist einem Transfer ein Commando-Byte voranzustellen, das unter anderem Die Registeradresse enthält. Bei einem Registerzugriff muss auch das Command-Bit (MSB = Bit 7) gesetzt werden, das geschieht durch oderieren mit der Konstanten **CMD**. Die Payload ist an ein Objekt gebunden, das dem Buffer Protokoll genügt.

Deshalb erzeuge ich das Bytearray **buf** und fülle die Elemente mit meinen Daten. **data** wird auf den Umfang eines Bytes beschränkt. **writeto**() sendet das array zum TCS3472. Das Autoincrement-Bit wird verdeckt ausgeschaltet, weil es nicht explizit gesetzt wird.

```
    def writeByte(self, adr, data):
        buf=bytearray(2)
        buf[1]=data & 0xff #
        buf[0]= adr | CMD
        self.i2c.writeto(HWADR,buf)
```

Ähnlich funktioniert **writeword**(), nur dass hier zwei Datenbytes zu senden sind. Das 16-Bit-Wort wird durch Undieren mit 0xFF und Rechtsschieben um 8 Bitpositionen in ein Low-Byte und ein High-Byte zerlegt. Im Kommando-Byte wird außerdem das Autoincrement (AUTOINC = 0x20) eingeschaltet, damit die Bytes in aufeinander folgende Register geschoben werden. Fehlt AUTOINC beim Oderieren, dann stehen in **buf**[0] in den Bits 5 und 6 zwei Nullen. Das bedeutet, dass bei wiederholten Lesezugriffen stets dasselbe Register angesprochen wird.

```
    def writeWord(self, adr, data):
        buf=bytearray(3)
        buf[1]=data & 0xff #
        buf[2]=data >> 8   # HIGH-Byte first (big endian)
        buf[0]= AUTOINC | adr | CMD
        self.i2c.writeto(HWADR,buf)
```

Die Methode **readByte**() liest ein Byte von dem in **adr** übergebenen Register ein. Das Bytearray **buf** mit nur einem Element erfüllt seinen Zweck sowohl beim Senden der Registeradresse im Kommando-Byte, als auch beim Einlesen des Werts. Zurückgegeben wird nicht der Buffer, sondern der Inhalt des nullten Elements, also ein Zahlenwert.

```
    def readByte(self, adr): #Register adresse
        buf=bytearray(1)
        buf[0]= adr | CMD
        self.i2c.writeto(HWADR,buf)
        buf=self.i2c.readfrom(HWADR,1)
        return buf[0]
```

readWord() liest einen 16-Bit-Wert ab der Adresse in **adr** ein und arbeitet ein bisschen tricky. Für das Senden der Adresse brauche ich nur ein Byte, für den Empfang aber zwei. Damit ich mit nur einem Bytearray auskomme, verfrachte ich CMD-Bit, AUTOINC und die Adresse in **buf**[0] und sende dann auch nur diesen Teil des Arrays buf[0:1]. Würde ich **buf**[0] senden, bekäme ich eine Fehlermeldung, weil dann ein Zahlenwert gesendet würde, dessen Typ **int** nicht auf dem Bufferprotokoll aufbaut. Folgende Eingaben in REPL machen das deutlich.

>>> buf=bytearray(2)
>>> buf[0] = 0b10111010
>>> buf[0]

>>> buf[0:1]
bytearray(b'\xba')
>>> type(buf[0])
<class 'int'>

```
    def readWord(self, adr): #Register adresse
        buf=bytearray(2)
        buf[0]=AUTOINC | adr | CMD
        self.i2c.writeto(HWADR,buf[0:1])
        buf=self.i2c.readfrom(HWADR,2)
        return buf[0] | buf[1] << 8
```

Über die folgenden neun Methoden gibt es nicht viel zu sagen. Sie benutzen die Methoden readByte() und readWord(), um die, dem Namen entsprechenden, Registerinhalte einzulesen. Exemplarisch greife ich **getItime**() und **getWait**() heraus.

```
    def getStatus(self):
        return self.readByte(STATUS)

    def getEnable(self):
        return self.readByte(ENABLE)

    def getAtime(self):
        return self.readByte(ATIME)

    def getItime(self):
        self.Itime=int(2.4*(256-self.getAtime()))
        return self.Itime

    def getWait(self):
        return 2.4*(256-self.readByte(WTIME))

    def getGain(self):
        return self.readByte(CONTROL)

    def getID(self):
        return self.PART[self.readByte(ID)]

    def getPersistance(self):
        self.fieldVal=self.readByte(PERS)
        return self.PERSIST[self.fieldVal]

    def getThresholds(self):
        low=self.readWord(AILTL)
        high=self.readWord(AIHTL)
        return low,high
```

Das Register ATIME enthält einen Bytewert, der die Dauer des Zeitfensters enthält, in welchem die Signale von den Fotodioden aufkumuliert werden. Die Berechnung des ATIME-Werts geschieht nach folgender Formel.

ATIME = 256 − Integration Time / 2.4 ms (Formel 1)

Integration Time gibt die Zeitdauer in Millisekunden vor. Die kürzeste Integrationszeit ist 2,4ms die längste 256 • 2,4ms = 614ms. Der ATIME-Wert wird also wohl in Einerschritten hochgezählt bis zum Überlauf des 8-Bitregisters. Nach 255 kommt 0 und damit stoppt der Integrationsprozess.

Um die Integrationszeit zu erhalten, muss (Formel 1) nach **Integration Time** aufgelöst werden, genau das verwendet die Methode **getItime**().

Integration Time = 2.4 ms × (256 − ATIME) (Formel 2)

Im Wert in WTIME steckt die Wartezeit zwischen zwei Messungen. Er wird ähnlich berechnet wie bei ATIME, sofern der Wert von WLONG = 0 ist. Ist WLONG = 1, dann ist die Wartezeit von 2,4ms bis 614ms noch mit dem Faktor 12 zu multiplizieren und liegt dann im Bereich zwischen 28,8ms und 7,37 Sekunden.

Zu den getXXXX()-Methoden gibt es als Gegenstück natürlich entsprechende setXXXX()-Methoden.

```
    def setThresholds(self,low=0,high=0):
        self.writeWord(AILTL,low)
        self.writeWord(AIHTL,high)
```

Mit **setThresholds**() kann ich Grenzwerte setzen, bei deren Unter- oder Überschreitung ein Pegelwechsel am Anschluss **Int** ausgelöst wird, sofern das durch Setzen des Bits AIEN im ENABLE-Register freigegeben ist.

```
    def setGain(self,val):
        assert val in range(4)
        self.gain=val
        self.writeByte(CONTROL,val)
```

Mit **assert** prüfe ich, ob der Wert in **val** im korrekten Bereich 0..3 liegt. Die Obergrenze ist in MicroPython üblicherweise nicht im Intervall. Die Feinheiten der Berechnung von **Atime** kennen Sie ja schon. Wenn an **intTime** nicht ein Vielfaches von 2,4 übergeben wurde, weicht der Rückgabewert der Funktion **setAtime**() vom Übergebenen Argument ab.

```
    def setAtime(self,intTime):
        self.Atime=256-int(intTime/2.4)
        self.writeByte(ATIME, self.Atime)
        self.Itime=int(2.4*(256-self.getAtime()))
        return self.Itime
```

Bei **setWaitTime**() wird natürlich auch zuerst der Bereich geprüft. **WLONG** muss den Wert 1 erhalten, falls die Zeitdauer in **waitTime** größer als 614 ist, andernfalls wird eine 0 in das Register geschrieben.

```
    def setWtime(self,waitTime):
        assert 2 < waitTime <= 7370
        if waitTime > 614:
            self.setWlong(1)
            wt = waitTime / 12
        else:
            self.setWlong(0)
            wt = waitTime
        self.Wtime=256-int(wt/12*5)
        self.writeByte(WTIME, self.Wtime)

    def setWlong(self, wlong):
        assert wlong in range(2)
        self.Wlong=wlong
        self.writeByte(CONFIG,wlong)
```

Der Wert im Register PERS (0 … 15) legt fest, wie viele aufeinanderfolgende Messungen außerhalb des Thresholdbereichs liegen müssen, damit ein Interrupt ausgelöst wird. Welcher Anzahl der Wert in PERS entspricht, weiß das Dict PERSIST.

```
    def setPersistance(self,val):
        assert val in self.PERSIST.values()
        for k,v in self.PERSIST.items():
            if val == v:
                self.fieldVal= k
                self.writeByte(PERS,k)
                return
```

Das **ENABLE**-Register 0x00 enthält drei Bit-Flags, welche die Statemachine beeinflussen, **PON**, **WEN** und **AEN**. Das Diagramm derselben, in Abbildung 3, sagt uns, dass der TCS3472 nach dem Empfang einer Startcondition auf dem I2C-Bus in den **Idle**-Status wechselt. Ist dabei **PON** = 1, dann hängt das weitere Verhalten von **AEN** und **WEN** ab. Ist zu diesem Zeitpunkt aber **PON** = 0, dann kehrt der TCS3472 in den **Sleep**-Status zurück. Zum Aufwecken muss also **PON** auf 1 gesetzt werden, das macht die Methode **wakeUp**(), die ihrerseits **setEnable**(pon=1) aufruft.

```
    def wakeUp(self):
        self.setEnable(pon=1) # Idle State
        sleep_ms(3)
        self.running=True
```

```
    def setEnable(self, aien=None,
                        wen=None,
                        aen=None,
                        pon=None):
        enable=self.readByte(ENABLE)
        if aien is not None:
            assert aien in [0,1]
            enable &= (255-AIEN)
```

```
            enable |= (aien << 4)
        if wen is not None:
            assert wen in [0,1]
            enable &= (255-WEN)
            enable |= (wen <<3)
        if aen is not None:
            assert aen in [0,1]
            enable &= (255-AEN)
            enable |= (aen << 1)
        if pon is not None:
            assert pon in [0,1]
            enable &= (255-PON)
            enable |= pon
        self.writeByte(ENABLE,enable)
        return enable
```

setEnable() ist durch die übergebenen Parameter in der Lage alle vier Flags zu setzen (Parameter = 1) oder rückzusetzen (Parameter = 0). In allen Fällen wird das Flag zuerst rückgesetzt und dann das Argument, 0 oder 1, auf die entsprechende Bitposition geschoben. Weil MicroPython keinen Operator zur Bildung des Einerkomplements besitzt, muss die Bildung der Differenz 255 – Flagbyte den Mangel ausgleichen. Parameter für die kein Wert übergeben wird, bleiben außen vor, weil sie den Defaultwert None haben.

Die Methode **toSleep**() ist das Gegenstück zum Erwecken. Ich lasse den Wert des ENABLE-Registers einlesen, lösche PON und AEN in der eben beschriebenen Art und Weise und schreibe das Byte zurück zum TCS3472.

```
    def toSleep(self):
        h=self.readByte(ENABLE)
        self.writeByte(ENABLE,h & (255-(PON+AEN)))
        self.running=False
```

Der Name ist Programm, die Routine **startSingle**() startet eine Einzelmessung. Der TCS3472 wird sicherheitshalber aufgeweckt, dann schalte ich die ADCs ein, indem ich **AEN** auf 1 setze. Der TCS3472 braucht 3ms für diesen Vorgang, außerdem lasse ich den ESP32 weiterschlafen, bis die Integrationszeit abgelaufen ist.

Danach hole ich die Farbwerte mit **getRawValues**() ab und bringe den TCS3472 in den Idle-Status, bevor ich das Tuple zurückgeben lasse.

```
    def startSingle(self):
        self.wakeUp()
        self.setEnable(aen=1) # Idle and ADC on
        sleep_ms(3) # wait for initialisation
        sleep_ms(int((256-self.Atime)*12/5)+1) # conversion
time
        vals=self.getRawValues()
        self.setEnable(pon=1,aen=0)
        return vals
```

Eine Dauerlaufmessung wird mit **start**() angestoßen. Nach dem obligatorischen Aufwecken setze ich die Wartezeit in Millisekunden zwischen den Messungen auf den übergebenen Argumentwert in **wtime**. Nach dem Setzen der drei Steuerflags auf 1 lasse ich den ESP32 die Wartezeit und Wandlungsdauer verpennen. Für das Abholen der Werte ist **getRawValues**() zuständig, für das Einhalten der Latencies weiterer Messungen ist der Anwender verantworlich.

```
    def start(self,wtime):
        self.wakeUp()
        self.setWtime(wtime)
        self.writeByte(ENABLE, WEN | PON | AEN) # loop
        sleep_ms(int(self.getWait()))
        sleep_ms(self.getItime())
```

```
    def getRawValues(self):
        self.clear= self.readWord(CLEARL)
        self.red  = self.readWord(REDL)
        self.green= self.readWord(GREENL)
        self.blue = self.readWord(BLUEL)
        return self.clear,self.red,self.green,self.blue
```

Das Datenblatt empfiehlt, die Farbwerte nicht byteweise einzulesen, sondern als Word. Das macht die Methode **readWord**(), die auch gleich den 16-Bit-Wert zurückgibt. Sobald nämlich das Low-Byte gelesen wird, wird das High-Byte blockiert, sodass es nicht durch einen zwischenzeitlich angekommenen neuen Messwert überschrieben werden kann.

Den Dauerlauf kann ich mit **stop**() beenden. Die Routine setzt einfach die Flags AEN und WEN zurück. Der TCS3472 bleibt dann im IDLE-Mode.

```
    def stop(self):
        self.setEnable(aen=0, wen=0)
```

Die Farbrohwerte sind **counts** von den ADCs und lassen einen groben Überblick zu. Besser ist es, man bezieht die Werte auf den Gesamtpegel den die klaren Fotodioden liefern. Darin müssen ja die anderen Farben anteilsmäßig vorkommen. Bis auf kleine Abweichungen ist R + G + B = CLEAR, 177 + 56 + 38 = 271 ≈ 268. Getestet wurde ein oranges Papier. Die Methode **getRGB**() erledigt die Umrechnung von counts in RGB-Werte und normiert diese auf den Bereich von 0 bis 255. Die if-Konstruktion verhindert einen **ZeroDivisionError**. Zurückgegeben wird ein RGB-Tupel.

```
    def getRGB(self):
        c,r,g,b=self.startSingle()
        if c==0:
            r,g,b=0,0,0
        else:
            r = int(r/c*255+0.5)
            g = int(g/c*255+0.5)
```

```
            b = int(b/c*255+0.5)
        return r,g,b
```

Die RGB-Werte benutzt die Routine **calcCCT**(), um die Correlated Color Temperature nach den Formeln von McCamy zu berechnen.

```
    def calcCCT(self,r,g,b):
        if r==0 and g==0 and b==0:
            return 0
        else:
            X = -0.14282 * r + 1.54924 * g - 0.95641 * b
            Y = -0.32466 * r + 1.57837 * g - 0.73191 * b
            Z = -0.68202 * r + 0.77073 * g + 0.56332 * b
            xc = X/(X+Y+Z)
            yc = Y/(X+Y+Z)
            n = (xc-0.3320)/(0.1858-yc)
            cct=449.0*(n**3) + 3525.0*(n**2) + \
                6823.3*n + 5520.33
            return int(cct)
```

Liegt der Klarwert außerhalb des mit Threshold eingestellten Bereichs, kann am ESP32 ein Interrupt ausgelöst werden, wenn die Leitung INT am TCS3472 mit einem interruptfähigen Eingang des Controllers verbunden wird und das Flag AIEN im ENABLE-Register gesetzt ist. Die Methode **clearClearChannel**() setzt den IRQ zurück.

```
    def clearClearChannelIRQ(self):
        buf=bytearray(1)
        buf[0]=SFUNC | CCICLR | CMD
        self.i2c.writeto(HWADR,buf)
```

Eine Methode mit Tripple-Effekt kommt zum Schluss. Die Methode **led**() schaltet die LED auf dem TCS3472-BOB ein oder aus, je nachdem als Argument eine 1 oder eine 0 übergeben wird. Wird kein Argument übergeben, dann wird **val** auf den Defaultwert None gesetzt und daraufhin der Zustand der LED zurückgegeben.

```
    def led(self,val=None):
        if val is not None:
            self.LED.value(val)
        else:
            return self.LED.value()
```

Eine ganze Reihe von get- und set-Prozeduren lassen sich auf diese Weise zusammenfassen. Wollen Sie es ausprobieren? Hausaufgabe bis zum nächsten Mal?

Natürlich wollen wir das neue Modul auch noch am Objekt testen. Haben Sie die Schaltung schon aufgebaut? Hier ist das Testprogramm.!

```
# colorcheck.py
from tcs3472 import TCS3472
from machine import Pin, SoftI2C
from oled import OLED
import sys
from time import sleep

if sys.platform == "esp8266":
    i2c=SoftI2C(scl=Pin(5),sda=Pin(4))
elif sys.platform == "esp32":
    i2c=SoftI2C(scl=Pin(22),sda=Pin(21))
else:
    raise RuntimeError("Unknown Port")

led=14
tcs=TCS3472(i2c,led=led)

d=OLED(i2c)
d.clearAll()
d.contrast(255)
d.writeAt("COLOR CHECK",0,0)
sleep(3)

tcs.led(1)
taste=Pin(0,Pin.IN,Pin.PULL_UP)

while 1:
    clear,red,green,blue=tcs.startSingle()
    d.writeAt("CLEAR: {} cnts   ".format(clear),0,1)
    d.writeAt("RED:   {} cnts   ".format(red),0,2)
    d.writeAt("GREEN: {} cnts   ".format(green),0,3)
    d.writeAt("BLUE:  {} cnts   ".format(blue),0,4)
    r,g,b=tcs.getRGB()
    cct=tcs.calcCCT(r,g,b)
    d.writeAt("CCT :  {} K   ".format(cct),0,5)
    sleep(1)

    if taste.value() == 0:
        d.clearFT(0,5,15,5)
        d.writeAt("CANCELLED",0,5)
        sys.exit()
```

Nach den Importen stellt das Programm den Controllertyp fest und richtet den I2C-Bus an den entsprechenden Pins ein. Ein TCS3472-Objekt wird instanziiert, ebenso das OLED-Display. In der while-Schleife hole ich die Farbrohwerte und gebe sie am Display aus. Dann werden die RGB-Werte berechnet und danach die Farbtemperatur. Zum Schluss wird die Flash-Taste abgefragt und das Programm sauber beendet, falls diese gedrückt ist.

Auch nach dem Programmende sind die Methoden des Moduls TCS3472 dem Controller noch bekannt. Sie können daher jetzt alle Routinen von Hand im Terminalbereich von Thonny ausprobieren.

LED aus:
>>> tcs.led(0)

LED abfragen
>>> tcs.led()
0

Verstärkung auf 4-fach
>>> tcs.setGain(tcs.GAINx4)

Dauerlauf starten mit kleinster Wartezeit
>>> tcs.start(2)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "tcs3472.py", line 230, in start
File "tcs3472.py", line 164, in setWtime
AssertionError:

>>> tcs.start(3)

>>> tcs.getRawValues()
(184, 85, 62, 37)
>>> tcs.getRawValues()
(53, 29, 14, 3)
>>> tcs.getRawValues()
(187, 87, 63, 37)

>>> tcs.stop()
>>> tcs.getRawValues()
(185, 85, 62, 37)
>>> tcs.getRawValues()
(185, 85, 62, 37)
>>> tcs.getRawValues()
(185, 85, 62, 37)

In der nächsten Folge werden wir das Modul TCS3472 compilieren. Das geht in Windows und natürlich auch in Linux. Bis dann!