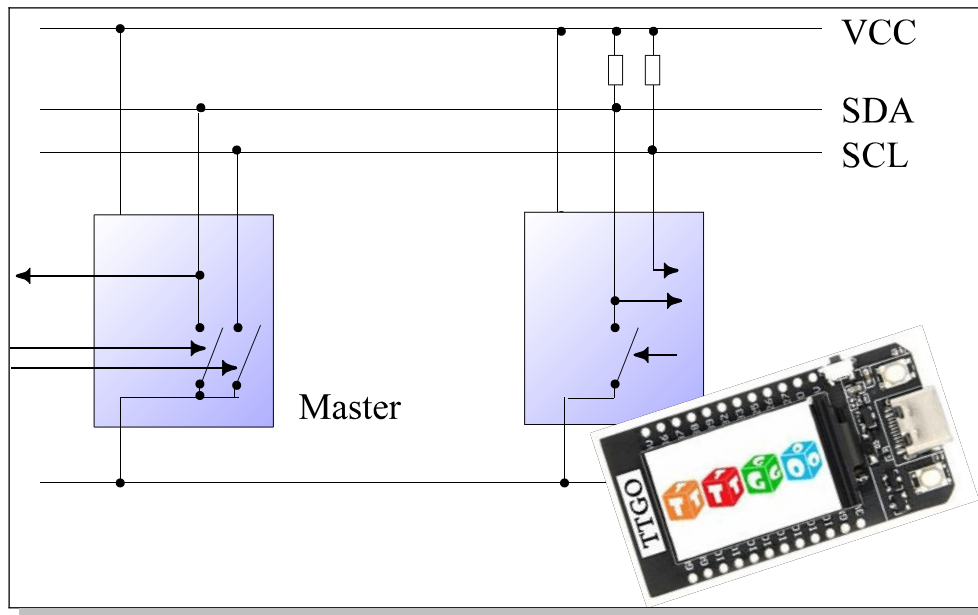


Das I²C-Bus-System



Aufbau, Funktionsweise und
Micropython-Programmierung

Eine Einführung
mit dem TTGO T-Display
und den I2C-Modulen
PCF8574 und LM75A/B

G. Heinrichs
30.12.2025

Inhaltsverzeichnis

- 1. Grundlagen**
 - 1.1 Der I2C-Datenbus
 - 1.2 Das I2C-Bus-Protokoll (Überblick)
 - 1.3 Das I2C-Bus-Protokoll genauer betrachtet
 - 1.4 Aufzeichnen und Analyse einer I2C-Datenübertragung
 - 2. I2C-Master und -Slave programmieren**
 - 2.1 Bit-Banging-Programm für einen einfachen Master
 - 2.2 Bit-Banging-Programm für einen einfachen Slave
 - 2.3 Verkabelung und Tests
 - 2.4 Bewährungsprobe: Bit-Banging-Master steuert PCF8574-Modul
 - 3. Die I2C-Klasse von Micropython (ESP32)**
 - 3.1 Umgang mit Bytes-Typen
 - 3.2 Noch einmal den PCF8574 steuern - diesmal mit der I2C-Klasse
 - 3.3 Port-Zustand beim PCF8574 lesen
 - 3.4 LM75-Modul: Temperaturwerte lesen (1 Byte, 2 Bytes)
 - 4. Noch mehr zum LM75: Mit verschiedenen Registern arbeiten**
 - 4.1 Register: schreiben und lesen
 - 4.2 Die I2C-Methoden readfrom_mem und writeto_mem
 - 4.3 Der OS Compare Mode
 - 4.4 Ein einfacher Thermostat
- Beiträge zu weiteren I2C-Modulen**
- Quellen**

Vorwort

In diesem Skript geht es nicht darum, möglichst viele verschiedene I2C-Projekte zu präsentieren. Vielmehr möchte ich die Funktionsweise des I2C-Busses an Hand einiger typischer Beispiele ausführlich erklären und auf dieser Grundlage eine Reihe von Micropython-Programmen für ESP32-Mikrocontroller (z. B. TTGO T-Display) entwickeln. Ich beginne mit einer Darstellung des I2C-Busses und des I2C-Bus-Protokolls. Auf dieser Grundlage werde ich entsprechende Programme für die Datenübertragung vorstellen. Bei diesen Programmen greifen wir zunächst nicht auf fertige Micropython-Funktionen zurück; stattdessen kontrollieren wir die Zustände der Signal-Leitungen direkt über das Setzen und Lesen der Aus- bzw. Eingänge (Bit-Banging).

Danach werden wir uns intensiv mit den Modulen **PCF8574** (8-Bit-Expander) und **LM75A** bzw. **LM75B** (mehr als nur Temperatur-Sensoren) beschäftigen. Auch hier werden wir immer wieder hinter die Kulissen schauen: So werden wir uns u. A. mit der Darstellung von negativen Zahlen im Zweiersystem beschäftigen und die Funktionsweise von I2C-Micropython-Methoden mit Hilfe von Signal-Diagrammen analysieren.

1. Grundlagen

Manchmal hat ein Computer oder ein Mikrocontroller nicht genug Ein- bzw. Ausgänge, um die oft zahlreichen Steuer- und Kontrollleitungen von einem oder mehreren Peripheriegeräten zu bedienen. In solchen Situationen wird oft das **I²C-Bus-System** eingesetzt. (I²C steht für Inter-Integrated Circuit.) Der I²C-Bus besteht aus 4 Leitungen, der Leitung für die Versorgungsspannung **VCC**, der **Masseleitung (GND)**, der Datenleitung **SDA** und der Taktleitung **SCL**. Diese verbinden einen Steuercomputer, den so genannten **Master**, mit einem oder mehreren Peripheriebausteinen, den **Slaves**. (Systeme mit mehreren Mastern werden wir hier nicht betrachten.) Der Master ist meist ein PC oder ein Mikroprozessor; als Slaves findet man z. B. Datenspeicher, I/O-Portbausteine, AD- oder DA-Wandler, Uhrenbausteine, Sensoren (u. A. für Temperatur, Luftdruck, Farbwerte, Gestenerkennung) und diverse Anzeigentreiber. Häufig werden diese I²C-Bausteine auf einer Platine montiert angeboten (s. Abb. 1); solche Platinen bezeichnen wir im Folgenden als **Module**.

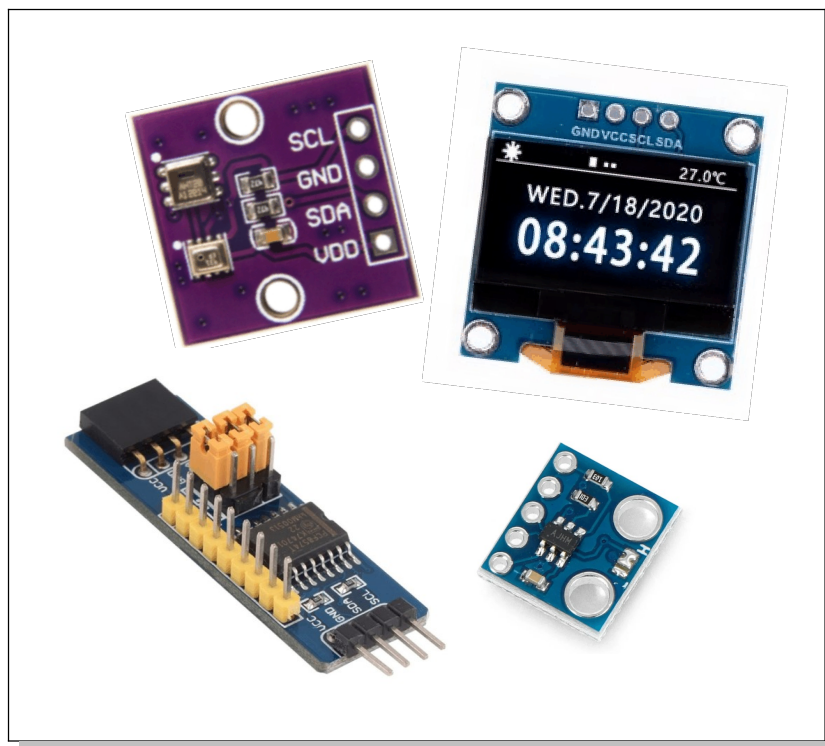


Abb. 1: Einige I²C-Module

1.1 Der I²C-Datenbus

Der Austausch der Daten zwischen dem Master und einem der Slaves geschieht über die beiden Leitungen SDA und SCL (s. Abb. 2). Da Daten vom Master sowohl gesendet als auch empfangen werden, arbeitet die SDA-Leitung bidirektional. Die entsprechenden Anschlüsse der Bausteine sind also sowohl Eingänge als auch Ausgänge. Eine Regel für Logikbausteinen besagt eigentlich, dass niemals 2 Ausgänge verbunden werden dürfen, was hier jedoch erfolgt. Daher sind eine spezielle Schaltungstechnik und ein strenges Datenprotokoll erforderlich, damit keine Fehler auftreten.

a) Schaltungstechnik

Durch Pull-Up-Widerstände (meist zwischen 5 k Ω und 10 k Ω) besitzen die SDA- und die SCL-Leitung den Zustand 1 (High), solange die internen Schalter von Master und Slave geöffnet sind; durch Schließen der Schalter können die SDA- bzw. SCL-Leitung den Zustand 0 (Low) erhalten (s. Abb. 2). Diese Schalter sind üblicherweise durch Transistoren o. Ä. realisiert. Die Pull-Up-Widerstände sind i. A. nicht Bestandteile des eigentlichen I2C-Bausteins. Die meisten I2C-Module besitzen aber schon solche Widerstände: In Abb. 1 kann man sie recht deutlich erkennen.

Wirkt ein Anschluss eines Bausteins als Ausgang, so ist er beim Senden einer 1 (High) hochohmig (Schalter offen). Zum Senden einer 0 wird über einen Schalter eine Verbindung zur Masse (GND) hergestellt, dadurch wird das Potential der entsprechenden Datenleitung auf Low gezogen. Die Taktleitung SCL wird nur vom Master angesteuert. Auf die Datenleitung SDA greifen Master und Slaves empfangend und sendend zu.

b) Datenprotokoll

Die gesamte Steuerung der Bausteine erfolgt über eine **festgelegte Reihenfolge von Signalen auf den Leitungen SDA und SCL**. Dabei ist es z. B. wichtig, dass Master und Slave **nicht gleichzeitig sendend** auf die SDA-Leitung zugreifen (mehr dazu im nächsten Abschnitt).

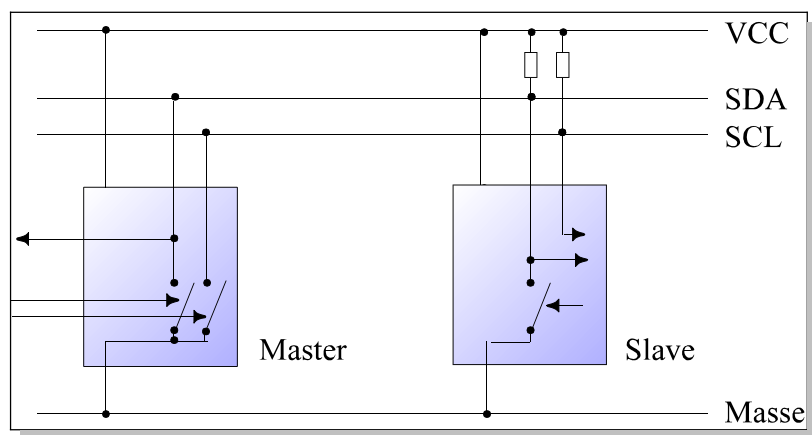


Abb. 2: Der I²C-Bus

1.2 Das I2C-Bus-Protokoll (Überblick)

Der I²C-Bus befindet sich normalerweise im **Ruhezustand**: Dabei sind SDA und SCL im 1-Zustand (VCC). Sämtliche Schalter sind geöffnet: Die Bausteine sind inaktiv. Sendet der Master nun ein **Start-Signal** (mehr dazu im nächsten Abschnitt), so wird eine **Datenübertragung** eingeleitet; dabei werden alle Slaves aktiviert, sie warten auf Signale vom Master.

Nun muss der Master eine Adresse senden, um einen der angeschlossenen Slaves auszuwählen. Eine Adresse besteht aus einem 8-Bit-Wort. Die ersten 4 Bit (beginnend mit dem höherwertigsten) werden durch den Hersteller festgelegt. Die nächsten 3 Bits können (manchmal) mit Hilfe einiger zusätzlicher Pins am Slave vom Anwender selbst bestimmt werden; sie werden auch als **individuelle Adress-Bits** bezeichnet. Durch diese lassen sich 8 gleichartige Bausteine mit unterschiedlichen Adressen ansprechen. Das niederwertigste Bit (LSB) legt fest, ob der Slave Daten senden oder empfangen soll: Ist LSB = 0, dann wird der Slave Daten empfangen, ansonsten wird er Daten senden (mehr dazu in Kapitel 3).

Jetzt sendet der Master die 8 Adressbits seriell an den oder die Slaves. Danach erzeugt der Master auf der Leitung SCL ein weiteres Taktsignal, bei dem der Baustein mit der passenden Adresse den Empfang der Adresse bestätigt. Insgesamt besteht dieser Adressierungsvorgang also aus insgesamt 9 Takten.

Nur der adressierte Baustein bleibt aktiv und kommuniziert weiter mit dem Master. Alle anderen Slaves gehen in den Ruhezustand und warten nun auf ein neues Start-Signal.

Je nach LSB des Adressbytes ist der adressierte Slave nun auf *Senden* oder *Empfangen* geschaltet. Soll ein Slave z. B. ein oder mehrere Daten empfangen, sendet der Master diese Daten byteweise und beendet die Übertragung mit einem Stop-Befehl. Genauer dazu finden Sie im nächsten Abschnitt.

Wie der Master Datenbytes von einem Slave empfängt, wird später im Zusammenhang mit den Sensor-Bausteinen noch genauer dargelegt.

1.3 Das I²C-Bus-Protokoll genauer betrachtet

Master und Slave tauschen Informationen byteweise aus. Als Beispiel für einen solchen Informationsaustausch soll hier nun ausführlich dargelegt werden, wie der Master einem Slave mit der binären Adresse 01001100 ein Datenbyte übermittelt. Dies geschieht in zwei Schritten: Zunächst sendet der Master diese Adresse aus; nur der Slave mit dieser Adresse bleibt nach dem Empfang der Adresse aktiv, die anderen gehen in den Ruhezustand. Anschließend sendet der Master das Datenbyte; es wird vom Slave empfangen und kann von ihm z. B. zur Steuerung eines Aktors dienen.

Die einzelnen Schritte im Detail:

0. Zunächst befinden sich alle Bausteine im **Ruhezustand**; er ist durch SDA = 1 und SCL = 1 gekennzeichnet.
1. Nun sendet der Master ein **Start**-Signal, indem er durch Schließen der beiden Schalter **zuerst SDA und dann SCL auf 0** setzt. Dadurch werden alle Slaves des Bus-Systems in Bereitschaft versetzt: Sie warten jetzt auf das Adressbyte (s. Schritt 2).
2. Der Master **sendet** nun bitweise die Adresse 01001100. Dabei beginnt er bei dem höchstwertigen Bit (Bit 7), in unserem Beispiel also mit der 0, die am weitesten links steht.

- 2.1 Der Master legt Bit 7 an SDA; in unserem Fall wird/bleibt dazu der SDA-Schalter geschlossen. Kurz darauf öffnet der Master den SCL-Schalter, so dass die SCL-Leitung vom Zustand 0 in den Zustand 1 übergeht. Dies ist für die Slaves das Signal, das Bit 7 von der Datenleitung SDA entgegenzunehmen und in einem Puffer zwischenspeichern. Kurze Zeit später setzt der Master die Clockleitung SCL wieder auf 0.
- 2.2 Der Master legt nun Bit 6 an SDA; in unserem Fall wird dazu der SDA-Schalter geöffnet; durch den Pull-Up-Widerstand erhält die SDA-Leitung jetzt den Zustand 1. Kurz darauf öffnet der Master wieder den SCL-Schalter, so dass die SCL-Leitung erneut vom Zustand 0 in den Zustand 1 übergeht. Dies ist für die Slaves das Signal, das nächste Bit (Bit 6) von der Datenleitung SDA entgegenzunehmen und in einem Puffer zwischenspeichern. Kurze Zeit später setzt der Master die Clockleitung SCL wieder auf 0.
- 2.3 Bit 5 bis Bit 0 werden auf gleiche Weise übertragen.
-2.8
- 2.9 Nachdem das Bit 0 übertragen worden ist, geschieht folgendes: Der Master öffnet seinen SDA-Schalter; dadurch geht die SDA-Leitung zunächst auf 1. Sämtliche Slaves vergleichen, ob das empfangene (und zwischengespeicherte) Adress-Byte mit ihrer eigenen Adresse übereinstimmt. Derjenige Baustein, welcher Übereinstimmung feststellt, legt nun SDA auf 0; dieses Signal wird **Acknowledge-Signal (ACK)** genannt. Der Master legt jetzt SCL auf 1 und prüft dann, ob die SDA-Leitung auf 0 oder 1 liegt. Liegt sie auf 1, geht der Master davon aus, dass sich kein Slave mit der gesendeten Adresse im Bus-System befindet. Liegt SDA nun aber auf 0, zeigt dies dem Master, dass der gewünschte Slave gefunden wurde. Der Master setzt nun seinerseits SCL wieder auf 0 und zeigt damit dem Slave, dass er dessen Acknowledge-Signal erhalten hat. Daraufhin öffnet der Slave wieder seinen SDA-Schalter, die SDA-Leitung wird dadurch auf 1 gelegt, so dass der Master im Folgenden die Datenleitung wieder benutzen kann. Der Slave ist jetzt zum Empfang eines weiteren Bytes (Datenbyte) bereit. Alle anderen Slaves gehen wieder in den Ruhezustand über, bis wieder ein Start-Signal erfolgt.

Diese in den Punkten 0 bis 2.9 dargestellte Signalfolge kann graphisch in einem so genannten **Timing-Diagramm** dargestellt werden. Unser Fall ist in Abb. 3 wiedergegeben.

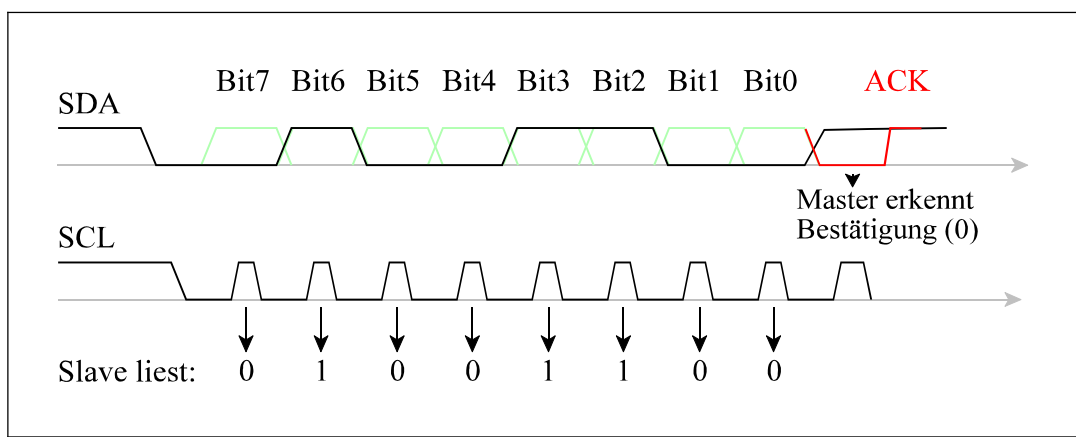


Abb. 3: Die Signale des Masters sind schwarz, die des Slaves rot dargestellt. Grün sind alternative Bitwerte des Masters angedeutet.

3. Nachdem durch entsprechende Schritte wie bei 2.1 - 2.8 auch das **Datenbyte** übertragen worden ist, sorgt der Slave für ein ACK-Signal. Es signalisiert diesmal dem Master, dass das Datenbyte übertragen worden ist. (Auf diese Weise können auch weitere Datenbytes übertragen werden.) Der Master legt am Ende zunächst SDA und dann auch SCL auf 1. Durch dieses **Stop-Signal** wird der ursprüngliche Ruhezustand wiederhergestellt.

1.4 Aufzeichnen und Analyse einer I²C-Datenübertragung

Diagramme wie in der Abb. 3 findet man auch häufig in den Datenblättern von I²C-Bausteinen. Zeichnet man das SDA- und das SCL-Signal mit einem Digital-Analysator (z. B. Logic, vgl. [RC5]) auf, so sieht das Diagramm in Wirklichkeit ein klein wenig anders aus (Abb. 4).

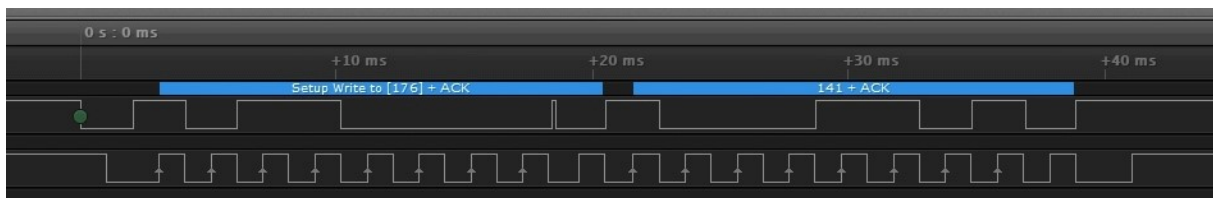


Abb. 4

In dem Diagramm von Abb. 4 wurde die Zahl 141 an einen Slave mit der 8-Bit-Adresse 176 gesendet. (Man beachte, dass bei dieser Adresse das niederwertigste Bit den Wert 0 hat!)

Es fällt auf: Die Flanken sind in Abb. 4 steiler als in der schematischen Darstellung von Abb. 3 dargestellt. Das erstaunt nicht: Der Digitalanalysator zeigt ja nur die Zustände 0 und 1 an und keine Potentiale. Eine Messung mit einem Oszilloskop macht indes deutlich, dass sich die Potentiale tatsächlich sehr rasch ändern (s. Abb. 5).

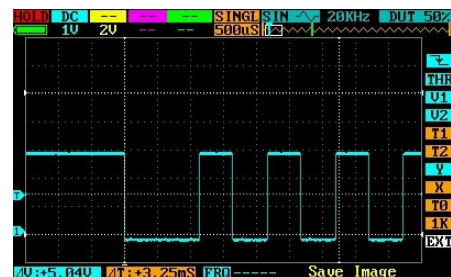


Abb. 5: SCL-Oszillogramm

Die SCL-Pulse liegen nicht mittig zu den SDA-Pulsen; dies wird schon bei dem ersten übertragenen Bit deutlich. Der Slave kontrolliert den Zustand an der SDA-Leitung, wenn das SCL-Signal gerade den 1-Zustand erreicht hat, also am Ende der aufsteigenden Flanke vom SCL-Signal.

Kontrolliert man nun die Zustände von SDA der Reihe nach bei diesen Markierungen, so erhält man die Bitfolge 1011000; diese stellt die Zahl 176 (also die Adresse unseres Slaves) dar. Der Digital-Analysator kommt zu demselben Ergebnis und gibt diese Adresse (blau unterlegt) an. Zusätzlich meldet er, dass beim 9. SCL-Signal die Leitung SDA auf 0/Low liegt; dies bedeutet: Die vom Master gesendete Adresse stimmt mit der Adresse des Slaves überein.

In der nächsten Phase wird die Zahl 141 übertragen. Der Slave quittiert den Empfang mit einem ACK-Signal.

Zum Abschluss sendet der Master das bereits bekannte Stop-Signal (zuerst SDA auf High und danach auch SCL auf High).

2. I2C-Master und -Slave programmieren

In diesem Kapitel werden wir zwei TTGO-Mikrocontroller als Master bzw. Slave einsetzen. Dazu erstellen wir sowohl für den Master als auch für den Slave Micropython-Programme. Dabei verzichten wir in diesem Kapitel auf den Einsatz der von Micropython bereitgestellten I2C-Klassen. Vielmehr werden wir die Signale direkt über die Pin-Instanzen

```
SDA = Pin(21, Pin.OUT) bzw. SDA = Pin(21, Pin.IN)
SCL = Pin(22, Pin.OUT)
```

erzeugen (**Bit-Banging-Methode**). Dabei greifen wir auf die detaillierte Beschreibung der I2C-Signale aus dem Abschnitt 1.3 zurück. Wer direkt mit den von Micropython zur Verfügung gestellten I2C-Klassen arbeiten möchte, kann dieses Kapitel überfliegen oder auslassen und zum nächsten Kapitel 3 übergehen.

2.1 Bit-Banging-Programm für einen einfachen Master

Der Einfachheit halber sollen sich an unserem I2C-Bus nur ein Master und ein einziger Slave befinden. Das Master-Programm soll lediglich ein einziges Daten-Byte an diesen Slave senden. Falls die vom Master gesendete Adresse mit der vom Slave übereinstimmt, soll das Datenbyte übertragen werden; andernfalls soll der Master eine entsprechende Fehler-Meldung ausgeben.

Das Programm ist nicht darauf ausgerichtet, die Daten möglichst rasch zu übertragen; vielmehr war es mir wichtig, die in Abschnitt 1.3 dargestellten Schritte auf möglichst transparente Weise umzusetzen. So habe ich z. B. die zur Adresse bzw. zum Datenbyte gehörige Bit-Folge in Form einer Liste dargestellt; damit kann man sehr übersichtlich auf die einzelnen Bits zugreifen.

```
# i2c_master_1.py
# 15.11.2025
# www.g-heinrichs.de

from sys import exit
from utime import sleep_ms, sleep_us
from machine import Pin

SDA = Pin(21, Pin.OUT, value = 1) # in get_ACK_signal(): Pin.IN
SCL = Pin(22, Pin.OUT, value = 1) # unverändert

# clock_time = 1000 # in us; Bit-Länge = 2 * clock_time
clock_time = 500 # funktioniert, gesamte Übertragungsdauer ca. 20 ms
# clock_time = 200 # funktioniert auch, aber Diagramm nicht mehr so schön
# clock_time = 100 # funktioniert auch, aber Diagramm nicht mehr so schön

address = 176 # 8-Bit_Adresse
# address_list = [1, 0, 1, 1, 0, 0, 0, 0]
# Adresse ist 176, RW-Bit = 0 (WRITE: Master sendet Datenbyte an Slave)
```

```
##### Funktionen #####
```

```
def num_to_byte_list(num):
    pot_2_list = [128, 64, 32, 16, 8, 4, 2, 1]
    byte_list = []
    for i in range(8):
        byte_list.append((num & pot_2_list[i])//pot_2_list[i])
    return byte_list

def start():
    SCL.value(1)
    SDA.value(1)
    sleep_us(clock_time)
    SDA.value(0)
    sleep_us(clock_time)
    SCL.value(0)
    sleep_us(clock_time)

def send_bit(bit):
    SCL.value(0)
    SDA.value(bit)
    sleep_us(clock_time)
    SCL.value(1)
    sleep_us(clock_time)
    SCL.value(0)

def send_byte(byte_list):
    for bit in byte_list:
        send_bit(bit)

def get_ACK_signal():
    global SDA
    SDA = Pin(21, Pin.IN)
    SCL.value(0)
    sleep_us(clock_time)
    SCL.value(1)
    ack_bit = SDA.value()
    sleep_us(clock_time)
    SCL.value(0)
    SDA = Pin(21, Pin.OUT)
    SDA.value(1)
    return ack_bit

def stop():
    # Stop-Signal: erst SCL und dann SDA "loslassen"
    sleep_us(clock_time)
    SDA = Pin(21, Pin.IN)
    sleep_us(clock_time)
    SCL = Pin(22, Pin.IN)
```

```
##### Hauptprogramm #####

data = int(input('data:'))

address_list = num_to_byte_list(address)
# print('address_list =', address_list)
data_list = num_to_byte_list(data)
# print('data_list =', data_list)

start()
send_byte(address_list)
# print('gesendet: data =', data_list, '; empfangen: ACK =',
      get_ACK_signal())

ACK_signal = get_ACK_signal()
if ACK_signal == 1:
    print('address: NACK')
    stop()
    print('Stop-Signal gesendet ')
    exit()
else:
    # print('address: ACK')
    send_byte(data_list)

ACK_signal = get_ACK_signal()

if ACK_signal == 0:
    print('Data_ACK erhalten')
else:
    print('Kein Data_ACK erhalten')

stop() # Stop-Signal
print('I2C_Master fertig')
```

2.2 Bit-Banging-Programm für einen einfachen Slave

```
# i2c_slave_1.py
# 15.11.2025
# www.g-heinrichs.de

# I2C-Slave zum Lesen eines Bytes
# zuerst Slave starten, dann Master

from machine import Pin
from utime import sleep_us
from sys import exit

SDA = Pin(21, Pin.IN)
SCL = Pin(22, Pin.IN)
slave_address = 176 # (8-Bit-Adresse)
pause = 2 # in us (ggf. abh. von Frequenz des Masters)
```

```
##### Funktionen #####
```

```
def start():
    global SDA
    while SDA.value() or SCL.value(): # warten, bis BEIDE Leitungen
                                        auf 0 liegen
        pass

def get_bit():
    global SDA
    while SCL.value() == 0: # auf Anfang eines Bit-Signals vom
                            Master warten
        pass
    # (ggf.) ein paar Mikrosekunden warten; m. E. nicht nötig
    # sleep_us(pause)
    bit = SDA.value()
    while SCL.value() == 1: # auf Ende des Bit-Signals vom Master warten
        pass
    sleep_us(pause)
    return bit

def get_byte():
    byte = 0
    for i in range(8):
        bit = get_bit()
        byte = 2*byte + bit
    return byte

def send_ack_bit(bit):
    # zieht SDA für 1 Bit-Zeit auf 0 (ACK), wenn bit = True ist,
    # d. h. die empfangene Adresse (rec_addr) und Slave-Adresse
    # (slave_address) gleich sind
    # print(bit)
    global SDA
    SDA = Pin(21, Pin.OUT)
    if bit:
        SDA.value(0) # ACK
    else:
        SDA.value(1) # NACK
    while SCL.value() == 0: # auf Anfang des SCL-Signals vom Master warten
        pass
    # sleep_us(pause)
    while SCL.value() == 1: # auf Ende des SCL-Signals vom Master warten
        pass
    # sleep_us(pause)
    SDA = Pin(21, Pin.IN)
```

```
##### Hauptprogramm #####
```

```
print('Slave gestartet...')
```

```
start()
```

```
rec_addr = get_byte() # gesendete Adresse empfangen und  
                      in received_addr speichern
```

```
print('Empfangene Adresse:', rec_addr)
```

```
if rec_addr == slave_address:
```

```
    send_ack_bit(True) # ACK senden
```

```
else:
```

```
    send_ack_bit(False) # NACK senden
```

```
    print('Adresse falsch : NACK gesendet -> Abbruch')
```

```
    exit()
```

```
# Datenbits empfangen und anzeigen:
```

```
data = get_byte()
```

```
send_ack_bit(True) # ACK senden
```

```
print('Empfangenes Byte:', data)
```

2.3 Verkabelung und Tests

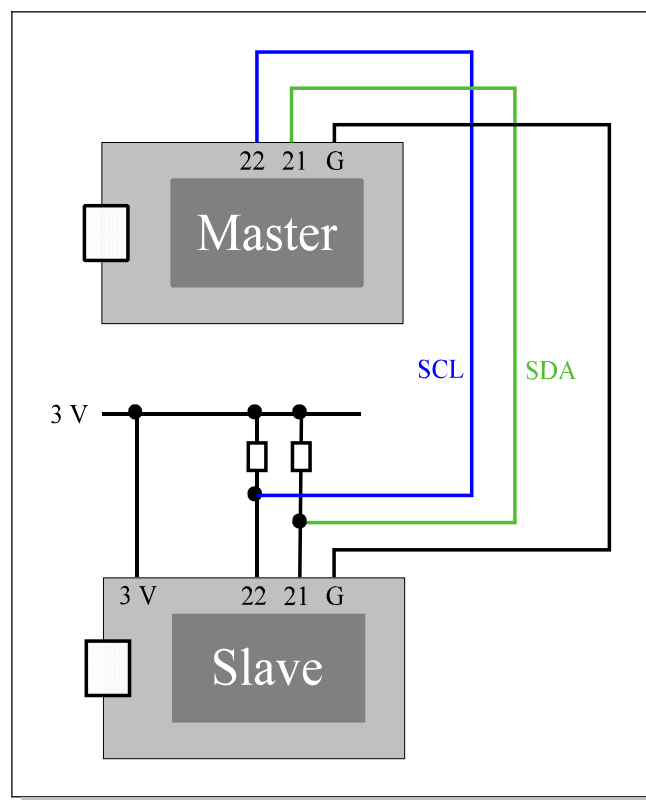


Abb. 6: Verbindung von Master und Slave.

In Abb. 6 ist die Verbindung von Master und Slave dargestellt. Die Position der Anschlüsse bezieht sich auf das Modul TTGO T-Display. Die Widerstände betragen jeweils 10 k Ω .

Die TTGO-Module werden an zwei PCs angeschlossen. Alternativ kann man den Slave standalone betreiben; dazu muss die Ausgabe des übertragenen Wertes auf dem Display angezeigt werden (s. `i2c_slave_1_display.py` in `Materialien.zip`).

Starten Sie zuerst das Slave-Programm `i2c_slave_1.py` für das Slave-Modul; im Terminal erscheint die Meldung:

```
I2C-Slave gestartet
```

Starten Sie dann das Master-Programm `i2c_master_1.py` für das Master-Modul; im Terminal erscheint:

```
I2C-Master gestartet  
Data:
```

Geben Sie nun hinter `Data:` eine Zahl zwischen 0 und 255 ein (z. B. 123) und betätigen Sie die ENTER-Taste.

Im Terminal des Slaves wird dann angezeigt:

```
Empfangene Adresse: 176  
Empfangenes Byte: 123
```

und im Terminal des Masters erscheint:

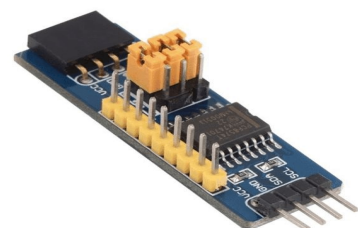
```
Data_ACK erhalten  
I2C-Master fertig
```

Testen Sie nun auch einmal selbst, was geschieht, wenn die Adresse 176 im Master-Programm abgeändert wird, z. B. zu 98.

Wie reagiert das Master-Programm, wenn der Slave den Empfang des Datenbytes mit einem NACK quittiert? Ändern Sie ggf. das Slave-Programm zur Beantwortung dieser Frage kurzfristig ab.

2.4 Bewährungsprobe: Bit-Banging-Master steuert PCF8574-Modul

Das I2C-Modul PCF8574 besitzt 8 Digital-Ausgänge. Sendet man ein Byte an dieses Modul, werden die Zustände dieser Ausgänge entsprechend eingestellt. Diese Zustände können wir mit Hilfe von LEDs anzeigen lassen. Dazu benutzen wir die Schaltung aus Abb. 8. Dabei haben die Widerstände einen Wert von 150 Ω .



Hat ein Ausgang den Zustand 0, dann fließt ein Strom durch die LED und sie leuchtet. Bei dieser Stromrichtung kann der Ausgang

Abb. 7: PCF8574-Modul

bis zu 25 mA liefern (vgl. 5.3 von [PCF]). Diese Schaltung wirkt auf den ersten Blick etwas irritierend: Die LED leuchtet, wenn am Ausgang ein 0-Signal vorliegt; umgekehrt leuchtet sie nicht, wenn das Ausgangssignal 1 ist. Das Leuchtmuster der LEDs ist also gegenüber dem Bitmuster des gesendeten Bytes genau umgekehrt.

Da stellt sich natürlich die Frage, warum man in Abb. 8 die Richtungen der LEDs nicht einfach umkehrt und die obere Leitung mit der Masse verbindet (und nicht mit VCC). Tatsächlich würde in diesem Fall ein Strom fließen, wenn der Ausgang den Zustand 1 hat (und kein Strom, wenn der Zustand 0 wäre). Allerdings liefert der PCF8574 bei **dieser Stromrichtung** nur eine sehr geringe Stromstärke (maximal 1 mA, vgl. 5.3 des Datenblatts [PCF]). Diese reicht nicht aus, um eine LED zum Leuchten zu bringen.

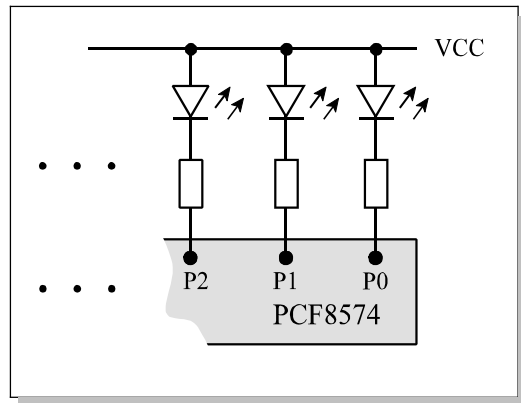


Abb. 8: Anschluss der LEDs an den PCF8574

Die LED-Schaltung aus Abb. 8 gibt es tatsächlich als fertiges Modul unter der Bezeichnung "LED Bar", z. B. bei AliExpress [LB]. Bitte beachten Sie: Es gibt LED Bars mit gemeinsamer Kathode und solche mit gemeinsamer Anode!



Abb. 9: LED Bar

Wir greifen jetzt auf ein ähnliches Modul zurück und verbinden es mit dem PCF8574-Modul. Dann schließen wir das PCF8574-Modul an unseren Master-TTGO an (s. Abb. 10). Nun müssen wir noch zwei Änderungen an unserem Programm `i2c_master_1.py` vornehmen: Zunächst ändern wir die Slave-Adresse zu 112; das ist die 8-Bit-Adresse des PCF8574, wenn die gelben Adress-Jumper des PCF8574 so wie in Abb. 7 gesetzt sind. Sodann ergänzen wir noch unser Programm: Wir fügen nach der Eingabe von `data` die Zeile

```
data = 255 - data
```

oder

```
data = data ^ 255 # bitweises XOR
```

ein; dadurch wird das Byte invertiert.

Wir starten nun das Programm und geben für `data` den Wert 85 ein. Das LED-Array zeigt dann

```
01010101
```

an (vgl. Abb. 10). Unser I2C-Master-Programm hat die Bewährungsprobe bestanden!

Probieren Sie auch einmal andere Werte für `data` und `clock_time` aus!

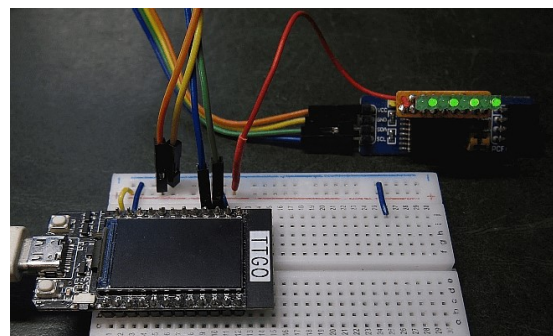


Abb. 10: TTGO und PCF8574 mit LED Bar

3. Die I2C-Klasse von Micropython (ESP32)

Unter Micropython stehen zwei I2C-Implementationen zur Verfügung, eine **Hardware-Implementation** (`machine.I2C`) und eine **Software-Implementation** (`machine.SoftI2C`). Die Hardware-I2C nutzt die zugrundeliegende Hardware-Unterstützung für schnelle und effiziente Kommunikation, sie ist jedoch oft an bestimmte Pins gebunden. Die Software-I2C, auch Bit-Banging genannt (Sie entspricht in etwa der Vorgehensweise aus Kapitel 2.), kann mit beliebigen Pins verwendet werden, ist aber weniger effizient. Beide Klassen bieten dieselben Methoden; sie unterscheiden sich hauptsächlich in der Art und Weise ihrer Programmierung. (nach [MiPy])

Wir werden im Folgenden nur mit der Hardware-I2C arbeiten; dabei benutzen wir für SDA den Pin 21 und für SCL den Pin 22; die Taktfrequenz soll 100 000 Hz betragen.

Die **Instanziierung** sieht dann so aus:

```
from machine import Pin, I2C
i2c = I2C(1, scl = Pin(22), sda = Pin(21), freq = 100_000)
```

Die Quelle [MiPy] gibt einige typische Beispiele für den Einsatz an:

```
i2c.scan()                # scan for peripherals, returning a list of 7-bit addresses

i2c.writeto(42, b'123')   # write 3 bytes to peripheral with 7-bit address 42
i2c.readfrom(42, 4)       # read 4 bytes from peripheral with 7-bit address 42

i2c.readfrom_mem(42, 8, 3) # read 3 bytes from memory of peripheral 42,
                           # starting at memory-address 8 in the peripheral
i2c.writeto_mem(42, 2, b'\x10') # write 1 byte to memory of peripheral 42
                                # starting at address 2 in the peripheral
```

Abb. 11: Methoden von i2c

Mit der **scan**-Methode erhält man eine Liste mit den Adressen aller am I2C-BUS angeschlossenen Slaves. Im Gegensatz zu unserer bisherigen Vorgehensweise benutzt die I2C-Klasse von Micropython eine **7-Bit-Adresse**. Diese entspricht der von uns bislang benutzten 8-Bit-Adresse **OHNE das niederwertigste Bit (LSB)**. Diese Vorgehensweise hat den Vorteil, dass wir für einen angeschlossenen Slave eine einzige Adresse benutzen können: Das LSB wird dann je nach Schreib- oder Lesevorgang von den entsprechenden Methoden separat gesendet: Bei einem Schreib-Befehl wird es eine 0, bei einem Lese-Befehl eine 1 sein. Die 7-Bit-Adresse des PCF8574 lautet z. B. 56; wir erhalten sie, indem wir die 8-Bit-Schreibadresse durch 2 teilen. Dieses Ergebnis liefert auch das folgende Scan-Programm:

```
# i2c_scan.py

from machine import I2C, Pin
i2c = I2C(1, scl = Pin(22), sda = Pin(21), freq = 100_000)

print('I2C-Bus scannen...')
devices = i2c.scan()
```

```

if len(devices) == 0:
    print('Keine I2C-Geraete gefunden!')
else:
    print('I2C-Geraete gefunden:', len(devices))
    for device in devices:
        addr = device
        print('Dez-Adr: ', addr)
        print('Hex-Adr: ', hex(addr))

```

3.1 Umgang mit Bytes-Typen

Bei den Schreib-Methoden in Abb. 11 fällt auf, dass die zu sendenden Bytes nicht in Form einer Zahl (Typ `int`), sondern als Byte-Strings (Typ: `bytes`) angegeben werden. Auch die vom Slave empfangenen Daten sind Byte-Strings. Wie kann man nun Zahlen in Byte-Strings und umgekehrt Byte-Strings in Zahlen umwandeln?

Hier ein Beispiel-Programm für die Umwandlung eines Byte-Strings in eine Zahl:

```

data = b'\x05\x41\x46' # Byte-String
print('data', '=', data)
int_value = int.from_bytes(data, 'big')
print(data, '=>' , int_value)

```

Es ergibt sich folgende Ausgabe auf dem Terminal:

```

data = b'\x05AF'
b'\x05AF' => 344390

```

Wir sehen: Byte-Strings werden durch `b'` eingeleitet und mit `'` beendet. Zwischen diesen Begrenzern stehen Zahlen im Hexadezimalsystem (eingeleitet durch `\x`). Hexadezimalzahlen, welche ein darstellbares ASCII-Zeichen besitzen, können auch mit diesem Zeichen dargestellt werden: `\x41` entspricht z. B. dem Zeichen A. Micropython ersetzt bei der Ausgabe im Terminal solche Hexadezimalzahlen automatisch durch das zugehörige ASCII-Zeichen; das ist von Vorteil, wenn ein Byte-String eine Zeichenkette repräsentieren soll. Umgekehrt kann man bei der Eingabe von Bytes Hexadezimalzahlen auch durch ihr zugehöriges ASCII-Zeichen eingeben.

Die Werte der einzelnen Bytes von `data` in Form von Zahlen (im Zehnersystem) erhält man in diesem Fall übrigens durch `data[0]`, `data[1]` und `data[2]`.

Bei der Umwandlung von einer Zahl (im Zehnersystem) in einen Byte-String muss man die gewünschte Länge des Byte-Strings angeben:

```

data = 270
number_of_bytes = 2
b = data.to_bytes(number_of_bytes, 'big')
print(data, '=>', b)

```


Das Programm liefert:

```
270 => b'\x01\x0e'
```

Würde man als Länge die Zahl 3 eingeben, würde man erhalten:

```
270 => b'\x00\x01\x0e'
```

Wählt man als Länge die Zahl 1, dann ist das Ergebnis:

```
270 => b'\x0e'
```

Zeichenketten (Typ **str**) können wir sehr einfach in Bytes-Strings (Typ **bytes**) umwandeln und umgekehrt:

```
s = "Hallo!"  
b = bytes(s, 'utf-8')  
print(b)
```

liefert als Ergebnis `b'Hallo!'`

und

```
b = b'Hallo!'  
s = str(b, 'utf-8')  
print(s)
```

liefert als Ergebnis

Hallo!

Wir sehen: Die Byte-Strings können als die grundlegende Datenstruktur angesehen werden. Deswegen ist es sinnvoll, dass die I2C-Methoden die Daten in Form von Byte-Strings austauschen (empfangen und auch senden).

3.2 Noch einmal den PCF8574 steuern - diesmal mit der I2C-Klasse

Wir benutzen wieder das PCF8574-Modul aus dem Abschnitt 2.4 mit dem aufgesteckten LED-Array. Das Programm soll wieder eine am Terminal eingegebene Zahl im LED-Array anzeigen. Das Programm ist jetzt sehr einfach:

```
# pcf8574_i2c_class_write.py  
# 17.11.2025  
# www.g-heinrichs.de
```

```
from machine import I2C, Pin

i2c = I2C(1, scl = Pin(22), sda = Pin(21), freq = 100_000)
address = 56 # 7-Bit-Adresse

data = int(input('Eingabe:'))
data_inv = data ^ 0xFF # bitweises XOR
data_inv_byte = data_inv.to_bytes(1, 'big')
i2c.writeto(address, data_inv_byte)
```

Beachten Sie, dass die eingegebene Zahl kleiner als 256 sein muss. Wer möchte, kann das Programm so ergänzen, dass fehlerhafte Eingaben abgefangen werden.

Unter dem Dateinamen `pcf8574_count.py` finden Sie in dem Materialien-Ordner auch ein Programm für einen LED-Binär-Zähler sowie ein zugehöriges Video.

3.3 Port-Zustand beim PCF8574 lesen

Den Zustand vom Port des PCF8574 können wir auch lesen: Wir entfernen das LED-Array, welches wir im letzten Abschnitt benutzt haben, und verbinden den Anschluss P0 des PCF8574 über einen Taster mit GND. Der Zustand des Tasters soll nun ermittelt werden.

Die Anschlüsse des Ports liegen standardmäßig auf 1. Wenn der Taster gedrückt wird, wird das Potential von P0 auf 0 gezogen. Den Zustand des Ports P0 können wir mit Hilfe der Methode `readfrom` auslesen. Weil wir nur ein einziges Byte vom PCF8574 lesen wollen, geschieht das nach Abb. 11 mit der Zeile

```
data = i2c.readfrom(address, 1)
```

Da bis auf das Port P0 alle anderen den Zustand 1 haben, gilt:

```
data = b'\xff', wenn der Taster nicht betätigt ist (Bit 0 = 1)
data = b'\xfe', wenn der Taster betätigt ist (Bit 0 = 0)
```

Das folgende Programm liest im Abstand von 1 s den Zustand des Ports und gibt ihn im Terminal aus.

```
# pcf8574_i2c_class_read.py
# 18.11.2025
# www.g-heinrichs.de

from machine import I2C, Pin
from time import sleep_ms

i2c = I2C(1, scl = Pin(22), sda = Pin(21), freq = 100_000)
address = 56 # 7-Bit-Adresse
```

```

while True:
    data = i2c.readfrom(address, 1)
    int_data = data[0] # bytes => num
    print(data, int_data, bin(int_data), ' ', end = '')
    bit0 = int_data & 1
    if bit0 == 1:
        print('Taster nicht gedrückt')
    else:
        print('Taster gedrückt')
    sleep_ms(1000)

```

3.4 LM75-Modul: Temperaturwerte lesen (1 Byte, 2 Bytes)

Mit einem LM75-Modul können wir Umgebungs-Temperaturen zwischen -55 °C und 125 °C messen. Wie bei den meisten Modulen sind die Pull-Up-Widerstände schon auf der Modul-Platine fest eingebaut. Bei meinen Modulen kann man auf der Löt-Seite die 3 individuellen Adress-Bits (s. S. 5) mit Hilfe von Lötbrücken festlegen. Ich lasse sie offen; in diesem Fall ist die **7-Bit-Adresse 0x48**. Es gibt zwei Varianten: **LM75A** und **LM75B**. Sie liefern Rohwerte mit 9 bzw. 11-Bit: Die ersten 8 Bit geben die ganzen Grad-Werte an, die restlichen halbe bzw. achte Grad (s. [VERS]).

Der LM75 kann die Temperatur-Werte als ganze Zahlen oder auch als Dezimalzahlen angeben. Wir betrachten zunächst nur den ersten Fall. Hierbei braucht nur ein einziges Byte gelesen werden. Das zugehörige Programm:

```

# lm75_1.py
# 24.11.2025
# www.g-heinrichs.de
# nur für nicht negative ganze Temperaturwerte

from time import sleep
from machine import Pin, I2C

i2c = I2C(1, scl = Pin(22), sda = Pin(21), freq = 100_000)
addr = 0x48 # 7-Bit-Adresse im HEX-Format

# Funktionen
def get_value():
    raw_value = i2c.readfrom(addr, 1)
    temp_value = raw_value[0] # byte -> num
    return temp_value

# Hauptprogramm
while True:
    temperature = get_value()
    print(temperature, '°C')
    sleep(2)

```

Nun wollen wir auch die 3 Nachkomma-Stellen vom LM75 auslesen. Dazu müssen wir zwei Byte auslesen. Der Lese-Befehl muss demnach jetzt `readfrom(addr, 2)` lauten.

Um die Nachkomma-Stellen richtig auszuwerten, muss man wissen, wie sie kodiert sind. Die folgende Abbildung gibt die beiden Bytes beim LM75B schematisch wieder:

MSByte								LSByte							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	X	X	X	X	X

Abb. 12: Die beiden Bytes des LM75B (s. [LM75])

Das linke Byte (MSByte) wird zuerst ausgelesen, das rechte Byte (LSByte) danach. Dieses zweite Byte besitzt nur 3 gültige Bits (D2, D1 und D0); die restlichen mit X gekennzeichneten Bits werden nicht benutzt, sie haben alle den Wert 0. Beim LM75A sind D1 und D0 (meist) unbenutzt.

Wie oben schon erwähnt geben D2, D1 und D0 die **Achtel-Grade**. Ist z. B. LSByte = [1 0 1 0 0 0 0], dann ist der zugehörige Temperatur-Wert $\frac{5}{8} \text{ °C} = 0,625 \text{ °C}$. Diesen Wert erhält man auch, wenn man das vollständige LSByte zugrunde legt; wegen $0b10100000 = 160$ ist nämlich

$$\frac{160}{256} = 0,625$$

Unser Programm muss jetzt nur die Ganzen und die 256-tel addieren:

```
# lm75_2.py
# 24.11.2025
# www.g-heinrichs.de

from time import sleep
from machine import Pin, I2C

i2c = I2C(1, scl = Pin(22), sda = Pin(21), freq = 100_000)
addr = 0x48 # 7-Bit-Adresse im HEX-Format

# Funktionen
def get_value():
    raw_value = i2c.readfrom(addr, 2)
    temp_value = raw_value[0] + raw_value[1]/256 # in Grad
    # raw_value[0] = [XXXXXXXX] gibt die ganzen Grade an
    # raw_value[1] = [XXX00000], gibt die 256-tel Grade an
    return temp_value

# Hauptprogramm
while True:
    temperature = get_value()
    print(temperature, '°C')
    sleep(2)
```

Der LM75 kann Temperaturen zwischen - 55 °C und + 125 °C messen. Unser bisheriges Programm liefert für negative Temperaturen aber keine korrekten Werte. Woran liegt das?

Der Grund dafür ist die Art und Weise, wie negative Zahlen im Zweiersystem dargestellt werden. Der Einfachheit halber verdeutliche ich dies an 4-Bit-Zahlen:

Zweiersys- tem	Zehnersystem
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

Hier geschieht beim Rückwärts-Zählen im Zweiersystem genau das, was man auch beim Rückwärts-Drehen eines mechanischen Zählwerks (im Zehnersystem) sehen kann:

Zählerstand	Deutung
...	...
0003	3
0002	2
0001	1
0000	0
9999	-1
9998	-2
9997	-3
...	...

Bei (älteren) mechanischen Kilometer-Zählern von Autos erhielt man durch Rückwärtsfahren tatsächlich (bei einer 4-stelligen Anzeige) den Zählerstand **9997**, wenn man bei einem Stand von **0002** eine Strecke von 5 km rückwärts gefahren war!

Bei den Zahlen im Zweiersystem zeigt das erste Bit an, ob die Zahl eine positive oder eine negative Zahl repräsentiert: Ist dieses Bit 1, dann stellt sie eine negative Zahl dar; hat es den Wert 0, dann ist ihr Wert nicht negativ (also 0 oder positiv). Dieses Bit bezeichnen wir deswegen als **Vorzeichen-Bit**.

Durch diese Art der Kodierung von negativen Zahlen kann die Addition von positiven und negativen Zahlen wie üblich durchgeführt werden. In der folgenden Box wird dies an einem einfachen Beispiel dargestellt.

5 + (-3)

```

      0 1 0 1
= + 11 1 01 1
  -----
  1 0 0 1 0

```

Rechnung im Zweiersystem; die rote 1 wird im 4-Bit-System ignoriert

= 2

Wie kann man nun zu einer negativen Zahl im Zweiersystem rechnerisch die zugehörige Zahl im Zehnersystem bestimmen? Dies geschieht üblicherweise mit Hilfe des **2-Komplements**. Wir zeigen dies an Hand der Zahl 1011_2 . Zunächst bilden wir davon das **1-Komplement**; dazu invertieren wir die einzelnen Bits, d. h. 1 wird gegen 0 ausgetauscht und umgekehrt. So erhalten wir die Zahl $0100_2 = 4$. Nun addieren wir die Zahl 1 und erhalten als Ergebnis 5; das ist das 2-Komplement. Zuletzt setzen wir ein Minuszeichen davor; das Ergebnis ist damit -5. (Genau diese Vorgehensweise wird z.B. auch im Datenblatt für den LM75B genannt!)

Das 1-Komplement einer Zahl `val` in einem Zahlensystem mit `num_of_bits` Bits erhalten wir durch die folgende Funktion `one_comp`; damit werden die einzelnen Bits von `val` invertiert, indem ihre XOR-Verknüpfung mit 1 gebildet werden: $0 \text{ XOR } 1 = 1$ und $1 \text{ XOR } 1 = 0$.

```
def one_comp(val, num_of_bits):
    all_bits_1 = 2 ** num_of_bits - 1
    return all_bits_1 ^ val # Bitweises XOR
```

Das 2-Komplement erhält man dann durch:

```
def two_comp(val, num_of_bits):
    return - (one_comp(val, num_of_bits) + 1)
```

Wir testen diese Funktionen anhand unseres Beispiels `val = 0b1011` und `num_of_bits = 4`:

```
val = 0b1011
num_of_bits = 4
t_c = (two_comp(val, num_of_bits))
print('Das 2-Komplement von', bin(val), 'ist', t_c)
```

Das Ergebnis lautet: Das 2-Komplement von `0b1011` ist -5.

Tatsächlich kann man an das 2-Komplement auch auf andere Art kommen. Wir gehen wieder von der Zahl 1011_2 aus und sehen sie als positive Zahl an; deren Wert im Zehnersystem ist 11. Von dieser Zahl subtrahieren wir $2^4 = 16$. (Würden wir z. B. in einem 8-Bit-System arbeiten, wäre der Exponent 8.) Das liefert $11 - 16 = -5$; dies stimmt mit der Angabe aus der ersten Tabelle der vorletzten Seite überein.

Mit diesen Kenntnissen ist es nun nicht mehr schwer, das Programm `LM75_2.py` so zu ergänzen, dass es auch für negative Temperaturen korrekte Werte liefert. Dazu müssen wir zunächst kontrollieren, ob `temp_value` eine negative Zahl darstellt. Dazu ermitteln wir das Vorzeichen-Bit von `temp_value`, indem wir einen 15-fachen **Rechts-Shift** durchführen:

```
sign_bit = temp_value >> 15
```

Ist `sign_bit` gleich 0, bleibt `temp_value` unverändert, ansonsten wird von `temp_value` die Zahl $2^{16} = 65536$ subtrahiert. Das vollständige Programm sieht dann so aus:

```
# lm75_3.py
# 24.11.2025
# www.g-heinrichs.de

from time import sleep
from machine import Pin, I2C

i2c = I2C(1, scl = Pin(22), sda = Pin(21), freq = 100_000)
addr = 0x48 # 7-Bit-Adresse im HEX-Format

# Funktionen
def get_value():
    raw_value = i2c.readfrom(addr, 2)
    temp_value = raw_value[0]*256 + raw_value[1] # in 256-tel Grad
    # temp_value = 0xC920 # zum Testen (54,875, vgl. Datasheet, S. 10)
    # raw_value[0] = [XXXXXXXX] gibt die ganzen Grade an
    # raw_value[1] = [XXX00000], gibt die 256-tel Grade an
    sign_bit = temp_value >> 15
    # print(sign_bit) # zum Testen
    if sign_bit == 1:
        temp_value = temp_value - 65536 # (2 << 15 bzw. 1 << 16)
    temp_value = temp_value / 256
    return temp_value

# Hauptprogramm
while True:
    temperature = get_value()
    print(temperature, '°C')
    sleep(2)
```

4 Noch mehr zum LM75: Mit verschiedenen Registern arbeiten

Die beiden Bytes, welche den Wert der gemessenen Temperatur beinhalten, sind in dem sogenannten **Temp(eratur)-Register** gespeichert. Neben diesem Temp-Register besitzt der LM75 noch 3 weitere Register, nämlich das Thyst-Register, das Tos-Register und das Conf-Register. Während das Temp-Register (sinnvollerweise) nur vom Master gelesen werden kann, handelt es sich bei den letzten drei Registern um Schreib-/Lese-Register. Der Master kann also Werte in ihnen speichern und diese auch wieder auslesen.

Tos steht für **overtemperature shutdown threshold**-Register. Es bezeichnet die Temperaturschwelle, bei deren Übersteigung der LM75-Baustein am Ausgang OS "aktiv" wird; damit kann zum Beispiel eine Heizung abgeschaltet (Shutdown) oder eine Kühlung aktiviert werden. Genaueres dazu erfahren Sie im Abschnitt 4.2.

Thyst steht für **Hysteresis**-Register; es gibt an, welcher Temperaturwert beim Absinken überschritten werden muss, damit der "Normalzustand" wieder erreicht ist; in diesem Fall wird der Ausgang OS wieder "inaktiv". Genaueres dazu erfahren Sie im Abschnitt 4.2.

Mit dem **Conf**-Register können bestimmte **Konfigurationen** festgelegt werden, z. B. kann damit auch genauer bestimmt werden, wie das Shutdown-Signal aussehen soll (Mehr dazu in Abschnitt 4.4.)

An dieser Stelle wird deutlich, dass der LM75 kein einfaches Temperatur-Messgerät ist; er kann vielmehr auch Signale liefern, die zur Temperatur-Regelung benutzt werden können, ohne dass hierzu ein Mikrocontroller eingesetzt werden müsste (mehr dazu in den Abschnitten 4.3 u. 4.4).

4.1 Register: schreiben und lesen

Wie schreibt man einen Wert in ein Register und wie liest man ihn (z. B. zu Kontrollzwecken) aus? Stellvertretend werden wir uns hier mit dem Thyst-Register beschäftigen. Bei den anderen Registern geht man analog vor. Alle Register besitzen eine Register-Adresse (s. Abb. 13). Das Thyst-Register hat z. B. die Adresse 2. Um gezielt auf ein Register zugreifen zu können (egal, ob lesend oder schreibend) besitzt der LM75 ein weiteres Register, das **Pointer-Register**. Dieses Register kann nur beschrieben und nicht gelesen werden.

Um auf das Tos-Register zugreifen zu können, benutzen wir die folgenden Befehle:

```
pointer = b'\x02' # Adresse vom Thyst-Register
i2c.writeto(addr, pointer)
```

Damit wird die Register-Adresse 2 in den Pointer geschrieben. **Alle darauf folgenden Lese-Vorgänge werden sich nun auf dieses Register beziehen.** Probieren wir es aus:

```
thyst_value_byte = i2c.readfrom(addr, 1) # 1 Byte lesen
thyst_value = thyst_value_byte[0]
print(thyst_value)
```

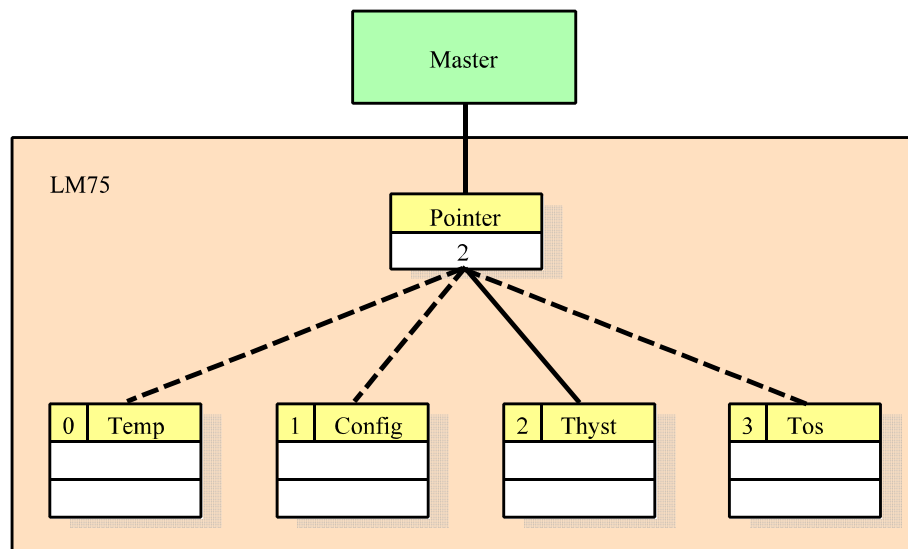



Abb. 13: Pointer- und Daten-Register

Wir erhalten den Standardwert/Default-Wert 75, auch wenn wir den Lesevorgang mehrfach ausführen lassen. (Übrigens: Der Wert 75 ist beim Kaltstart des Moduls automatisch im Thyst-Wert-Register abgelegt worden.)

Hier und im Folgenden werden wir uns der Einfachheit halber nur um das erste Byte des jeweiligen Registers kümmern, so wie wir es auch bei unserer ersten Temperatur-Messung gemacht haben (s. LM75_1.py in Kapitel 3.4).

Der aufmerksame Leser wird sich vielleicht wundern, dass in Kapitel 3.4 derselbe Lese-Befehl nicht den Thyst-Wert geliefert hat, sondern den Temperatur-Wert. Die Erklärung ist ganz einfach: **Nach einem Kaltstart steht im Pointer-Register immer 0**; der Pointer verweist dann auf das Temp-Register.

Wenn das LM75-Modul noch nicht von der elektrischen Quelle getrennt worden ist, nachdem der Pointer auf 2 gesetzt worden ist, sollte das Temperatur-Messprogramm jetzt den Wert 75 °C liefern. Probieren Sie es aus!

Um mit dem Programm LM75_1.py wieder Temperaturwerte zu erhalten, muss der Pointer wieder auf die Adresse des Registers Temp gesetzt werden. Das können Sie mit den Programm-Zeilen

```
pointer = b'\x00' # Adresse vom Temp-Register
i2c.writeto(addr, pointer)
```

erreichen; Sie können aber auch den LM75 (kurzzeitig) von der elektrischen Quelle trennen.

```
pointer_and_thyst_bytes = b'\x02\x19'
```

4.2 Die I2C-Methoden `readfrom_mem` und `writeto_mem`

Mit den I2C-Methoden `readfrom_mem` und `i2c.writeto_mem` (vgl. Abb. 11) kann der Zugriff auf die Inhalte der Register des LM75 erleichtert werden: Hier können wir als zusätzlichen Parameter den Pointer angeben; dieser wird in Abb. 11 als **memory-Address (Register-Adresse)** bezeichnet.

Schauen wir uns die Beispiele aus Abb. 11 genauer an:

<code>i2c.readfrom_mem(42, 8, 3)</code>	read 3 bytes from memory of peripheral 42 starting at memory-address 8 in the peripheral
	lies 3 Bytes bei dem Slave mit der Slave-Adresse 42 aus dem Register mit der Register-Adresse 8
<code>i2c.writeto_mem(42, 2, b'\x10')</code>	write 1 byte to memory of peripheral 42 starting at address 2 in the peripheral
	schreibe 1 Byte (b'\x10') bei dem Slave mit der Slave-Adresse 42 in das Register mit der Register-Adresse 2

Hier zwei Beispiele für die Anwendung:

```
# Thyst-Wert schreiben, s. lm75_write_Thyst_mit_mem_1.py
addr = 0x48
pointer = 2 # Thyst-Register
thyst_value = 25
thyst_value_byte = bytes([thyst_value])
i2c.writeto_mem(addr, pointer, thyst_value_byte)
```

```
# Thyst-Wert lesen, s. lm75_read_Thyst_mit_mem_1.py
addr = 0x48
pointer = 2 # Thyst-Register
thyst_value_byte = i2c.readfrom_mem(addr, pointer, 1)
thyst_value = thyst_value_byte[0]
```

In der folgenden Abb. 15 ist das Signal-Diagramm für das letzte Beispiel zu sehen. Hier erkennt man deutlich, dass die `readfrom_mem`-Methode genau dieselben zwei Schritte durchführt wie die beiden aufeinander folgenden Befehle:

`writeto(addr, pointer)` und `readfrom(addr, 1)`,

die wir weiter oben schon eingesetzt haben. Zusätzlich sehen wir auch, dass die 8-Bit-Schreibadresse gleich $0x48 * 2 = 72 * 2 = 144$ und die 8-Bit-Lese-Adresse $144 + 1 = 145$ ist.

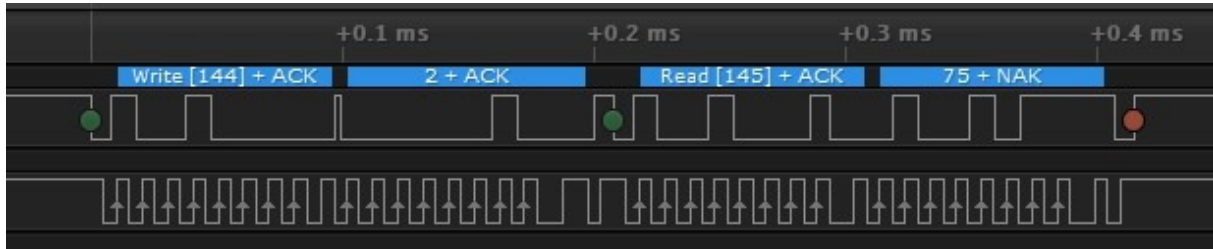


Abb. 15: `i2c.readfrom_mem(0x48, 2, 1)`

4.3 Der OS Compare Mode

O.S. steht für “over-temperature output”. Dieser “Übertemperaturausgang” signalisiert, ob der Tos-Wert überschritten oder der Thyst-Wert unterschritten worden ist. Der OS-Comparator ist standardmäßig im so genannten **Compare Mode**. Wir werden uns in diesem Abschnitt nur mit diesem Modus beschäftigen.

Bei unserem Modul ist der OS-Ausgang über einen Pull-Up-Widerstand und eine LED (wie man mit einer Lupe sehen kann) mit VCC verbunden (s. Abb. 16). Wenn nach dem Power-Up die Temperatur unter dem Tos-Wert liegt, ist der OS-Ausgang **inaktiv**, es fließt praktisch kein Strom: Die LED zwischen OS und VCC leuchtet nicht (s. Abb. 16). Wenn kein nennenswerter Strom fließt, ist die Spannung zwischen OS und VCC praktisch 0; d. h. der OS-Ausgang liegt auf (nahezu) gleichem Potential wie VCC: **OS hat den Zustand High**, vgl. Abb. 17.

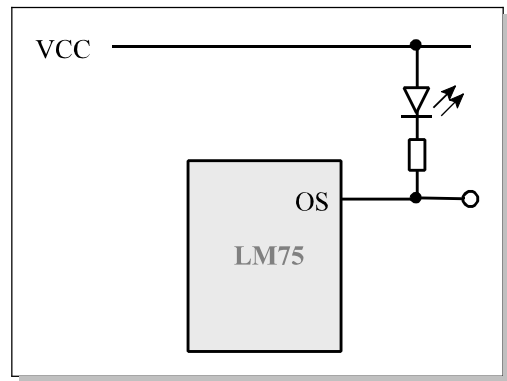


Abb. 16

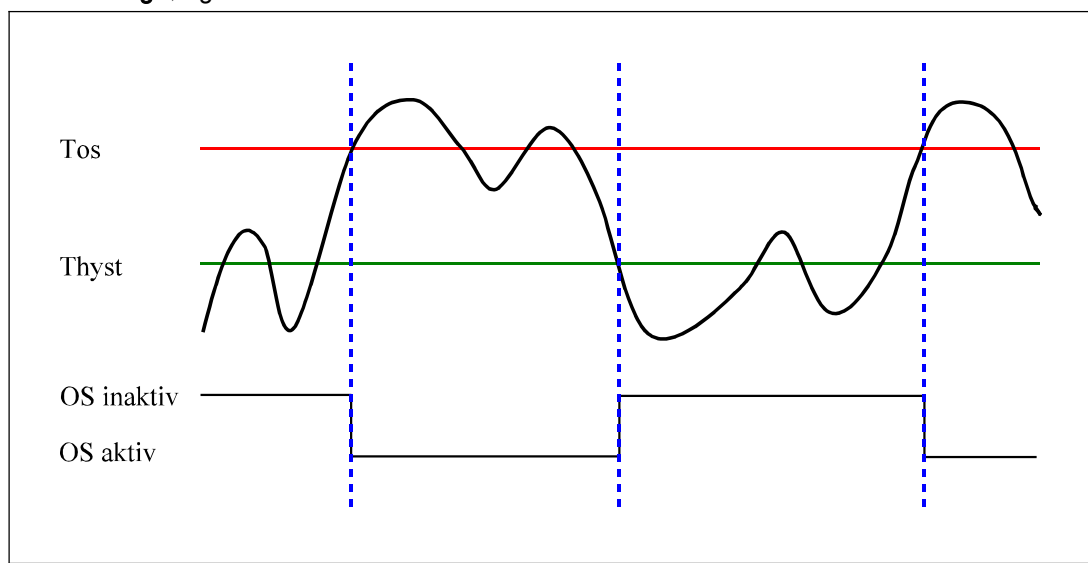


Abb. 17

Wenn Tos nun überschritten wird, wird der OS-Ausgang **aktiv**, es fließt ein Strom und die LED zwischen OS und VCC leuchtet. Das Potential am **OS-Ausgang ist dann Low** (ungefähr GND). Erst wenn die gemessene Temperatur danach gleich oder unter Thyst ist, wird der OS-Ausgang wieder inaktiv und die LED erlischt (s. Abb. 17).

Man beachte: Der Zustand von OS ändert sich nicht, wenn der Thyst-Wert überschritten oder der Tos-Wert unterschritten wird (s. Abb. 17).

Mit dem folgenden Programm können Sie dies mit Hilfe eines Föhns leicht überprüfen. Beachten Sie, dass bei diesem Programm nur nicht negative Temperaturen korrekt angezeigt werden (vgl. Abschnitt 3.4).

```
# lm75_OS_comp.py
# 28.11.2025
# www.g-heinrichs.de

from time import sleep
from machine import Pin, I2C

i2c = I2C(1, scl = Pin(22), sda = Pin(21), freq = 100_000)
addr = 0x48 # 7-Bit-Adresse im HEX-Format

# Funktionen
def get_value(): # liefert nicht negativen ganzen Temperaturwert
    raw_value = i2c.readfrom(addr, 1)
    temp_value = raw_value[0] # byte -> num
    return temp_value

# Hauptprogramm
# Tos-Wert festlegen
tos_pointer = 3
tos_value = int(input('Tos in °C:'))
tos_value_byte = bytes([tos_value])
i2c.writeto_mem(addr, tos_pointer, tos_value_byte)

# Thyst-Wert festlegen
thyst_pointer = 2 # Thyst
thyst_value = int(input('Thyst in °C:'))
thyst_value_byte = bytes([thyst_value])
i2c.writeto_mem(addr, thyst_pointer, thyst_value_byte)

# Pointer soll wieder auf Temp-Register verweisen
temp_pointer = 0 # Adresse vom Temp-Register
i2c.writeto(addr, bytes([temp_pointer]))

# Mess-Schleife
while True:
    temperature = get_value()
    print(temperature, '°C')
    sleep(2)
```

Nachdem Sie z. B. 40 °C für Tos und 30 °C für Thyst eingegeben haben, erwärmen Sie den LM75-Baustein vorsichtig mit dem Föhn. Wenn die 40 °C - Marke überschritten wird, dann leuchtet die LED auf. Lassen Sie nun den Baustein abkühlen. Sobald nun die 30 °C - Marke unterschritten wird, geht die LED wieder aus. Mit einem Multimeter können Sie auch einmal das Potential am OS-Ausgang prüfen. Sie werden feststellen, dass OS auf Low (High) liegt, wenn die LED leuchtet (nicht leuchtet).

Anschließend sollten Sie das Programm einmal anhalten, die Stromversorgung des LM75 aber nicht unterbrechen. Erwärmen Sie nun noch einmal den LM75, so geht die LED nach einiger Zeit an. Umgekehrt geht sie nach dem Abkühlen wieder aus. Offensichtlich ist das Übertemperatur-Management unabhängig davon, ob beim LM75 Temperaturwerte vom TTGO abgefragt werden oder nicht. Aus diesem Grund könnte nach einmaligem Konfigurieren der Tos- und Thyst-Werte der LM75 selbstständig (d. h. ohne Unterstützung durch einen Mikrocontroller) einen Kühlaggregaten steuern.

4.4 Ein einfacher Thermostat

In diesem Abschnitt wollen wir ein einfaches Modell für einen Thermostaten auf der Basis unseres LM75 vorstellen. Dieses Modell soll über eine Glühbirne ein Alarmsignal geben, wenn der Tos-Wert überschritten wird. Daraufhin können Kühlungsmaßnahmen ergriffen werden. Sobald die Temperatur unter den Thyst-Wert sinkt, soll das Alarmsignal beendet werden. Die Erwärmung und die Abkühlung bewerkstelligen wir dabei mit einem Föhn.

Alternativ kann die Glühbirne auch durch einen Ventilator ersetzt werden; dieser besteht aus einem Mikro-Motor mit Propeller (s. [MOT]). Die Erwärmung soll diesmal durch die Wärmestrahlung einer (Rotlicht-) Lampe erfolgen. Wenn der LM75 nun durch die Bestrahlung den Tos-Wert überschreitet, soll der Motor durch den LM75 eingeschaltet werden. Der LM75 wird dann durch den Zufuhr der kalten Umgebungsluft gekühlt; wenn die Temperatur dabei unter den Thyst-Wert sinkt, wird der Motor wieder ausgeschaltet.

In beiden Fällen wird dazu der OS-Ausgang des LM75 benutzt. Da dieser nur geringe Ströme zur Verfügung stellen kann, setzen wir als Verstärker einen Transistor ein. Wir benutzen hier den npn-Transistor BC 547; dieser kann am Emitter einen Strom von bis zu 100 mA liefern, wenn an der Basis (und damit auch am OS-Ausgang) ein High-Signal vorliegt (vgl. Abb. 18).

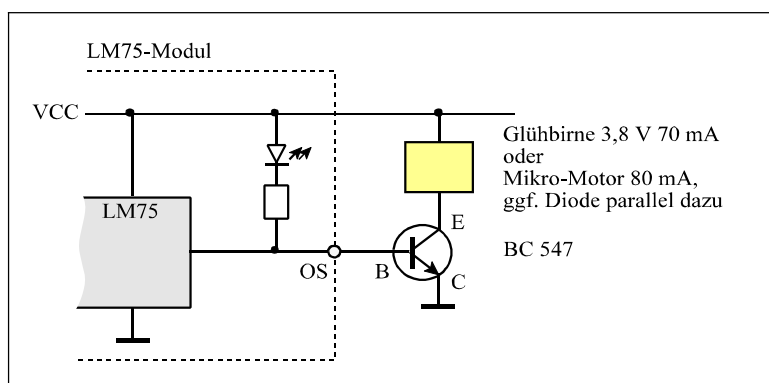


Abb. 18

Nun hatten wir im letzten Abschnitt festgestellt, dass am OS-Ausgang ein Low-Signal vorliegt, wenn OS aktiv ist. Man kann den LM75 aber so konfigurieren, dass der OS-Ausgang im aktiven Zustand ein High-Signal (und umgekehrt im inaktiven Zustand ein Low-Signal) liefert. Dazu benutzen wir das Conf-Register. Dieses hat die Register-Adresse 1. Standardmäßig ist der Wert dieses Registers 0. Im Datenblatt des LM75 finden wir:

Table 8. Conf register ...continued

Legend: * = default value.

Bit	Symbol	Access	Value	Description
B2	OS_POL	R/W		OS polarity selection
			0*	OS active LOW
			1	OS active HIGH

Abb. 19

Die Umkonfigurierung erfolgt somit durch die folgenden Programmzeilen:

```
# Conf-Wert festlegen
conf_pointer = 1
conf_value = 4 # OS active High
conf_value_byte = bytes([conf_value])
i2c.writeto_mem(addr, conf_pointer, conf_value_byte)
```

Diese Programmzeilen müssen wir in unser Programm `lm75_OS_comp.py` einfügen. Das neue Programm finden Sie unter dem Dateinamen `lm75_OS_comp_inv.py` im Materialien-Ordner `Materialien.zip`.

Wir bauen die Schaltung aus Abb. 18 auf und starten das Programm `lm75_OS_comp_inv.py`. Wir geben die folgenden Temperaturwerte ein:

`Tos = 35` und `Thyst = 25`

Das Programm gibt jetzt jede 2 Sekunden den aktuellen Temperaturwert auf dem Display des TTGO an. Die Glühbirne leuchtet nicht; die OS-LED auf dem Modul ignorieren wir. Nun erwärmen wir das LM75-Modul mit warmer Luft aus einem Föhn. Die vom Programm ausgegebenen Temperaturwerte steigen langsam an. Unmittelbar bevor auf dem Display der Wert 35 °C angezeigt wird, leuchtet die Glühbirne auf. Jetzt wird die Heizwendel beim Föhn ausgeschaltet, und das LM75-Modul wird gekühlt. Die angezeigte Temperatur sinkt; dabei leuchtet die Glühbirne weiter. Kurz bevor der Thyst-Wert erreicht wird, geht die Glühbirne aus. Unser Modell-Thermostat funktioniert! (In dem Materialien-Ordner finden Sie das Video `LM75_OS_2.mp4`, welches diesen Vorgang dokumentiert.)

Mancher fragt sich an dieser Stelle vielleicht, warum der Thermostat schon reagiert, kurz bevor der Tos- bzw. Thyst-Wert auf dem Display angezeigt wird. Der Grund dafür ist: **Der LM75 misst die Temperatur fortwährend im Abstand von 100 ms** und ändert ggf. sofort den OS-Zustand. Die Temperaturwerte werden von unserem Programm hingegen nur jede zwei Sekunden zur Anzeige gebracht. Deswegen können im Extremfall zwischen dem An- bzw. Ausgehen der Glühbirne und der Anzeige des Tos- bzw. Thyst-Wertes auf dem Display ca. 2 Sekunden verstreichen.

Beiträge zu weiteren I2C-Modulen

Auf meiner Webseite <https://www.g-heinrichs.de/wordpress/index.php/attiny/module/> finden Sie folgende Artikel:

- Entfernungsmessung mit dem SRF02
- I2C-LCD von YwRobot
- Kompassmodul HDMM0
- Lux-Sensor BH-1750

Auf meinem Forum <https://www.forum.g-heinrichs.de/> gibt es Artikel und Programme zu folgenden I2C-Modulen:

- AD/DA-Wandler PCF8591
- Entfernungsmessung mit dem Ultraschallsensor srf02
- Das DHT20-Modul (Temperatur und Luftfeuchtigkeit)
- Eine Klasse für das DHT20-Modul
- Messung von Pulsraten mit dem MAX3010
- Der APDS9930-Sensor (Messung von Entfernungen und Lux-Werten)
- Der APDS9960-Sensor (Farbsensor, Gestensensor...)
- Wetter-Station mit dem BME/BMP280-Modul

Quellen

[LB] <https://de.aliexpress.com/item/32784458240.html>

[PCF] Datenblatt zum PCF8574: PCF8574.pdf in `Materialien.zip`

[MiPy] <https://docs.micropython.org/en/latest/library/machine.I2C.html>

[LM75] Datenblätter zum LM75: LM75.pdf, LM75A.pdf, LM75A_nxp.pdf und LM75B.pdf in `Materialien.zip`

[RC5] s. Kap. 3 im Skript von <https://www.forum.g-heinrichs.de/viewtopic.php?f=18&t=202>

[VERS] Sämtliche LM75B-Module liefern Rohwerte mit 11 Bit. Bei den LM75A-Modulen erhält man - je nach Hersteller und Produktionsdatum - Rohwerte mit 9 oder 11 Bit. Hier hilft ein Blick in die entsprechenden Datenblätter (vgl. [LM75]).