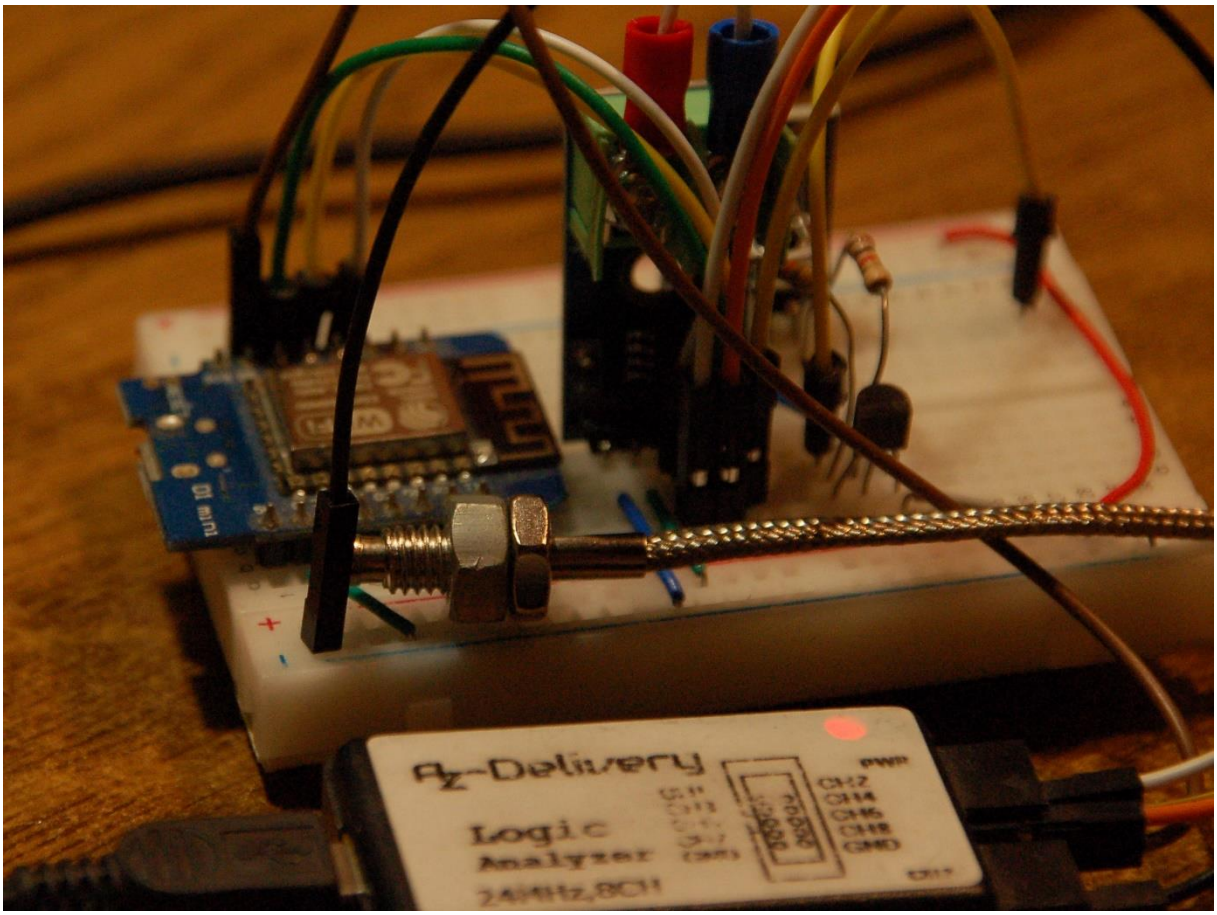

MAX6675-Aufbau mit ESP8266

Diesen Beitrag gibt es auch als PDF-Datei.

Vor einiger Zeit besorgte ich mir ein Temperaturmess-Kit, das mit einem Temperaturfühler in Form eines Thermoelements vom K-Typ arbeitet. Das heißt, der Fühler besteht aus einer **Chrom-ChromNickel**-Kombination. Das Gute daran ist, der Messbereich. Das vorliegende Element wird an einem **MAX6675-Chip** betrieben und kann in dieser Kombination Temperaturen von **0°C bis 1023,75°C** erfassen. Das Thermoelement wäre sogar für einen Bereich von -250°C bis 1200°C geeignet – mit gewissen Vorbehalten und mit einer anderen Auswerte-Elektronik.

Auf jeden Fall macht der Messbereich diese Kombination interessant, geht es doch mit den sonst üblichen Sensoren wie DS18B20, AHT10, SHT21, BME280 etc. stets nur um Bereiche bis maximal 125°C. Wir werden heute die Funktion des Thermoelements unter die Lupe nehmen, aus den Informationen des Datenblatts zum MAX6675 ein MicroPython-Modul bauen, ein Beispielprogramm als Anwendung schreiben und damit zeigen, wie man für bestimmte Temperaturen die Messgenauigkeit des Aufbaus bestimmen kann. Als Anzeigeelement wird ein kleines OLED-Display dienen.

Das alles erwartet Sie in dieser neuen Folge aus der Reihe

MicroPython auf dem ESP32, ESP8266 und Raspberry Pi Pico

heute

Ein Thermometer bis 1023,75°C

Etwas Theorie – Der Seebeck-Effekt

Ein thermoelektrischer Wandler besteht aus zwei verschiedenen Metallen oder Legierungen, deren Enden verschweißt oder verlötet sind. Einer der Leiter wird aufgetrennt.

Der thermoelektrische oder **Seebeck**-Effekt besteht darin, dass an den Kontaktstellen zweier unterschiedlicher Metalle eine Wanderung von Elektronen stattfindet. Wenn man die Kontaktstellen erhitzt oder abkühlt, führt das wegen der unterschiedlichen Geschwindigkeit der Elektronen an den Kontaktstellen zu einer elektrischen Spannung an der Stelle, wo einer der beiden Leiter aufgetrennt ist. Weil das Element selbst als Spannungsquelle wirkt, ist die technische Stromrichtung in dem Kreis innerhalb des Elements vom Minus- zum Pluspol, denn durch das Voltmeter fließt der Strom ja von plus nach minus.

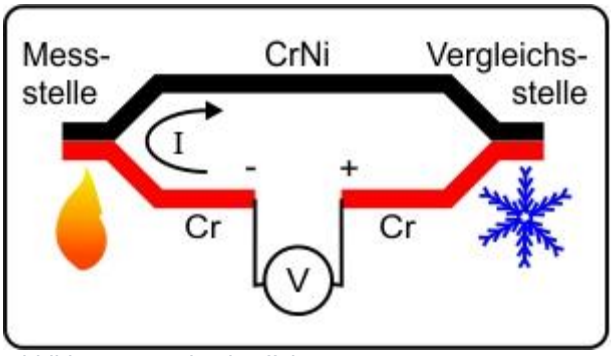

Abbildung 1: Seebeck-Effekt

In unserem Fall gibt es nur die linke Verbindung, denn das Thermo-Couple hat nur zwei Ausgänge, die Enden der beiden unterschiedlichen Leiter. Die rechte Verbindung, der kalte Kontakt, wird durch eine interne Schaltung im MAX6675 durch eine Diode ersetzt, die ähnliche Eigenschaften aufweist, wie die Vergleichsstelle. Dadurch wird die Vergleichstemperatur an der Position des MAX6675 simuliert und vom Wert an der Messstelle abgezogen.

Thermoelemente können nie eine Temperatur absolut messen, sondern immer nur Temperaturdifferenzen zwischen den Kontaktstellen. Ferner ist zu bedenken, dass durch die Verbindung zwischen CrNi und Cu sowie zwischen Cr und Cu ebenfalls Thermospannungen auftreten, die aber ebenfalls vom MAX6675 berücksichtigt werden. Wir müssen uns darum also nicht kümmern.

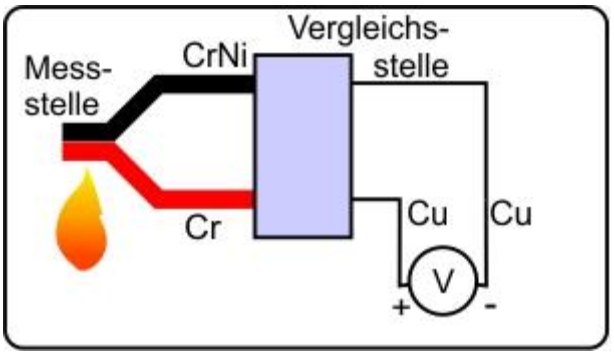

Abbildung 2: Seebeck-Effekt mit simulierter Vergleichsstelle

Die Messstelle ist in unserem Fall in einer Edelstahl Kapsel untergebracht, die ihrerseits in einem Zylinder mit 6mm-Außengewinde steckt. Die Ableitungen sind glasfaserisoliert und mit einem Edelstahlgeflecht ummantelt. Leider ist die Kapsel mit dem Zylinder nicht wasserdicht verpresst. Eine druckdichte Montage in einem Flüssigkeits- oder Gastank ist daher ohne weitere abdichtende Maßnahmen nicht möglich.


Abbildung 3: Thermo Couple

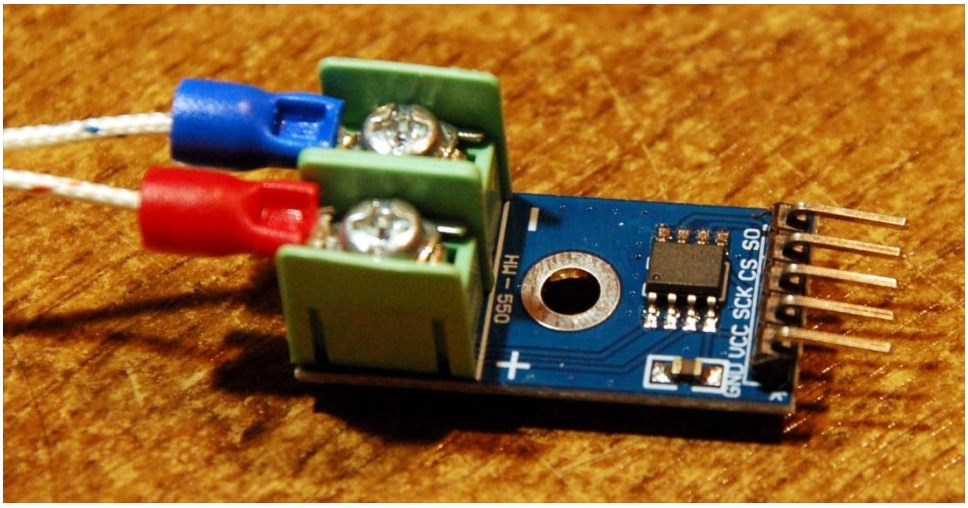

Abbildung 4: MAX6675

Werfen wir gleich einen Blick in das Datenblatt des MAX6675. Auf Seite 1 lesen wir, dass die Temperatur mit 12-Bit-Auflösung codiert wird, der höchste Wert aber 1024°C ist und die kleinste Stufe 0,25°C beträgt. Diese Info ist wichtig für die

Programmierung. Weiter unten im Datenblatt, auf den Seiten 4 bis 6 finden wir noch weitere Hinweise. Die Simulation der Spannung am kalten Ende durch die Vergleichsstufe (siehe Abbildung 2) funktioniert von -20°C bis 85°C Umgebungstemperatur des Chips. Die Nachführung der Korrekturspannung managt der MAX6675 durch eine temperaturabhängige Diode. Keine gute Idee ist es, eine Wärmequelle neben dem Chip zu platzieren, weil dadurch die Erfassung der Temperatur verfälscht wird.

Die tatsächlich höchste Temperatur, die gemessen werden kann, wird auf Seite 4 mit 1023,75°C angegeben. Das ist nachvollziehbar, wenn alle 12 Bits auf 1 stehen. Zehn Bits vertreten den ganzzahligen Anteil, die beiden niederwertigsten Bits den Bruchanteil des Messwerts.

Die Hardware

Die Hardware unterscheidet sich natürlich im eingesetzten Controller und den dafür nötigen Breadboards. ESP8266 und Raspberry Pi Pico kommen mit einem kleinen Breadboard mit 400 Pins und vier Stromschienen aus. Für den ESP32 braucht man entweder zwei solcher Boards oder besser gleich zwei Boards der Variante mit 820 Pins. Wegen der breiteren Platine des Controllerboards müssen wir zwei Breadboards über eine Stromschiene aneinanderstecken, damit noch Pins für die Jumperkabel übrigbleiben.

Die Schaltungen sind nicht besonders anspruchsvoll und daher für Einsteiger gut geeignet. Gleiches gilt auch für die Programmierung.

ESP8266

1	NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F WIFI Wifi Development Board mit CP2102 oder D1 Mini NodeMcu mit ESP8266-12F WLAN Modul kompatibel mit Arduino oder D1 Mini V3 NodeMCU mit ESP8266-12F
1	MAX6675 Temperatur Sensor mit Sonde K-Typ und Jumper Wire für Raspberry Pi
1	Mini Breadboard 400 Pin mit 4 Stromschienen
1	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F
optional	Logic Analyzer

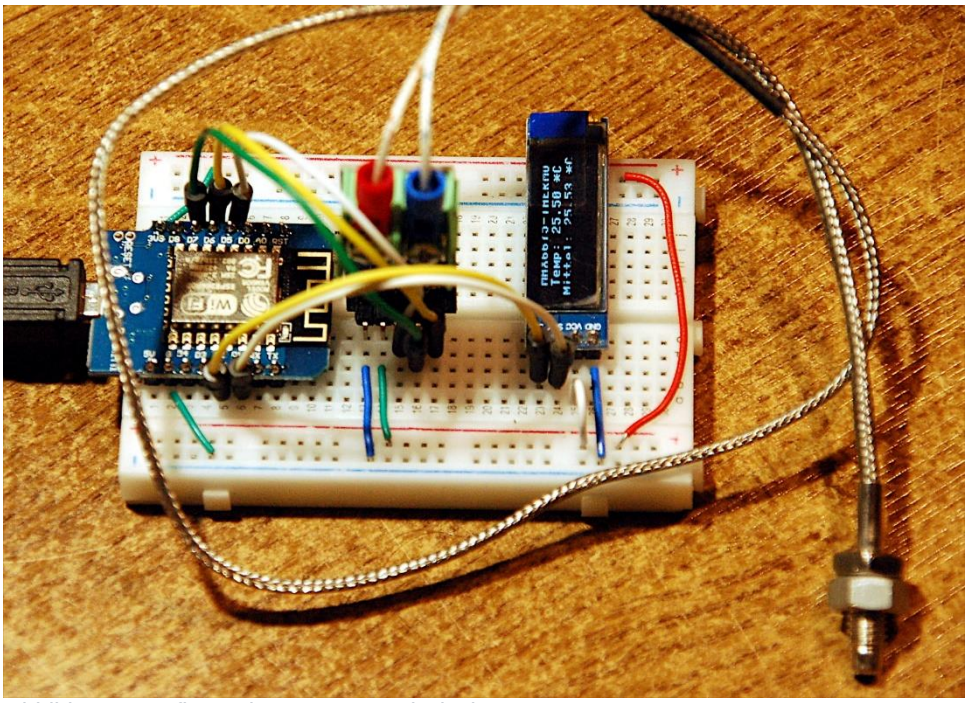

Abbildung 5: Aufbau mit ESP8266 und Display

Für meinen Aufbau habe ich einen D1 Mini V3 NodeMCU verwendet, weil der zusammen mit der restlichen Hardware, perfekt auf ein kleines Breadboard passt.

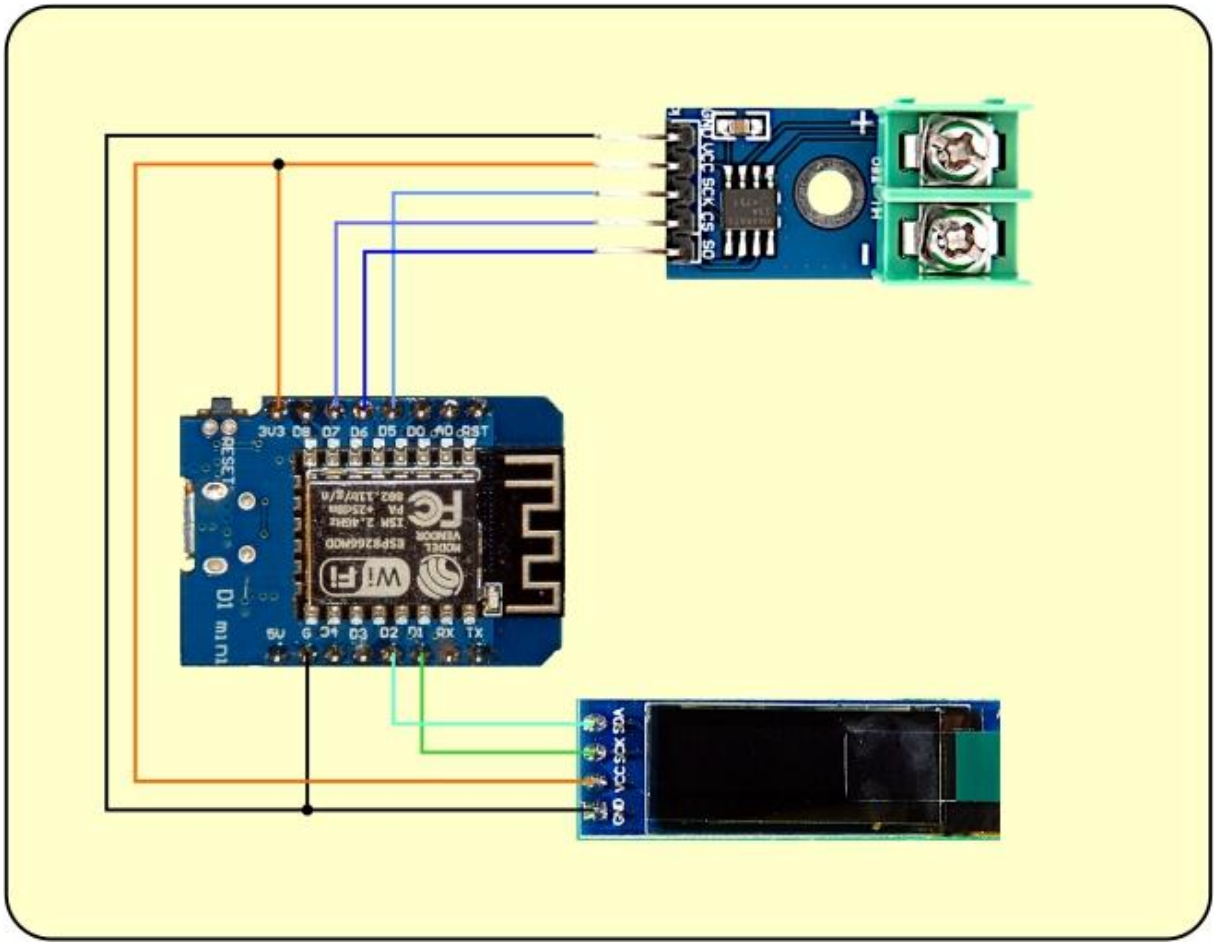

Abbildung 6: MAX6675-Thermocouple - Schaltung mit ESP8266

ESP32

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 ESP32 Lolin LOLIN32 WiFi Bluetooth Dev Kit
1	MAX6675 Temperatur Sensor mit Sonde K-Typ und Jumper Wire für Raspberry Pi
1	Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord
optional	Logic Analyzer

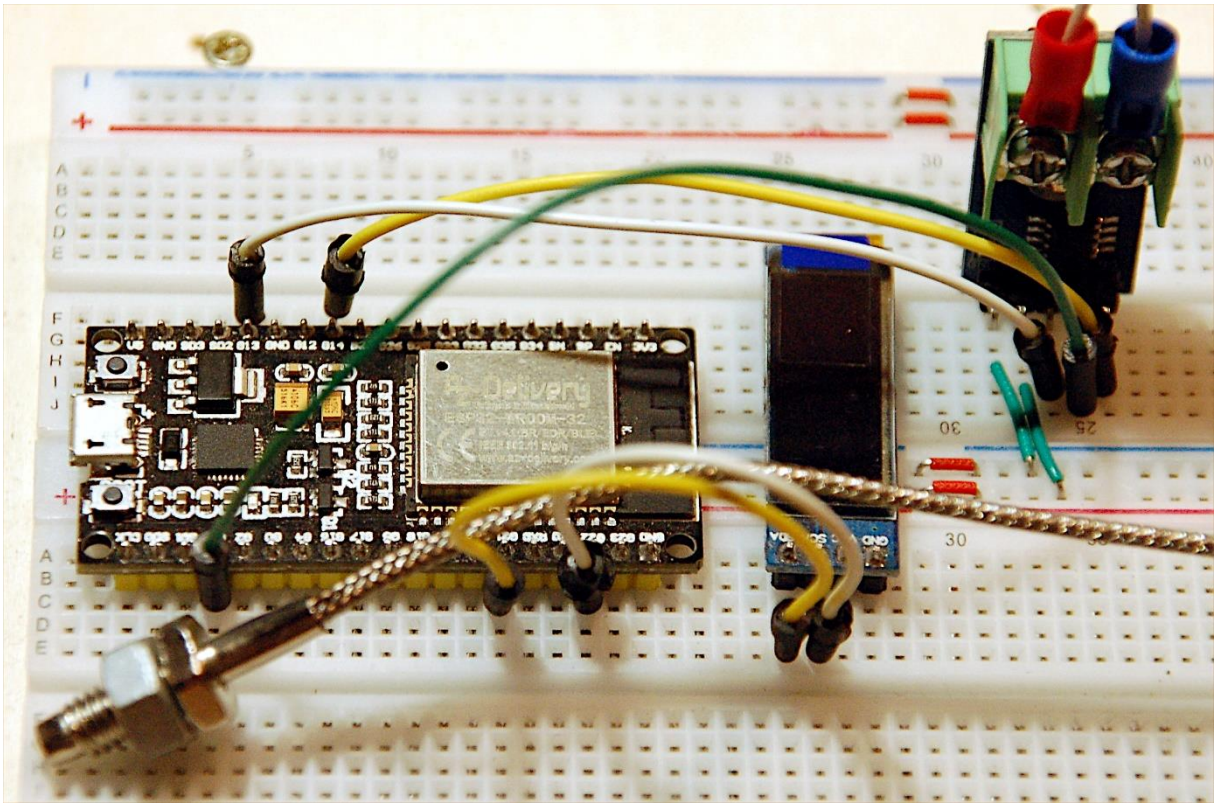

Abbildung 7: ESP32 mit MAX6675

Für den ESP32 braucht man, wegen der größeren Breite des Boards zwei Breadboards, die über eine Stromschiene zusammengesteckt sind.

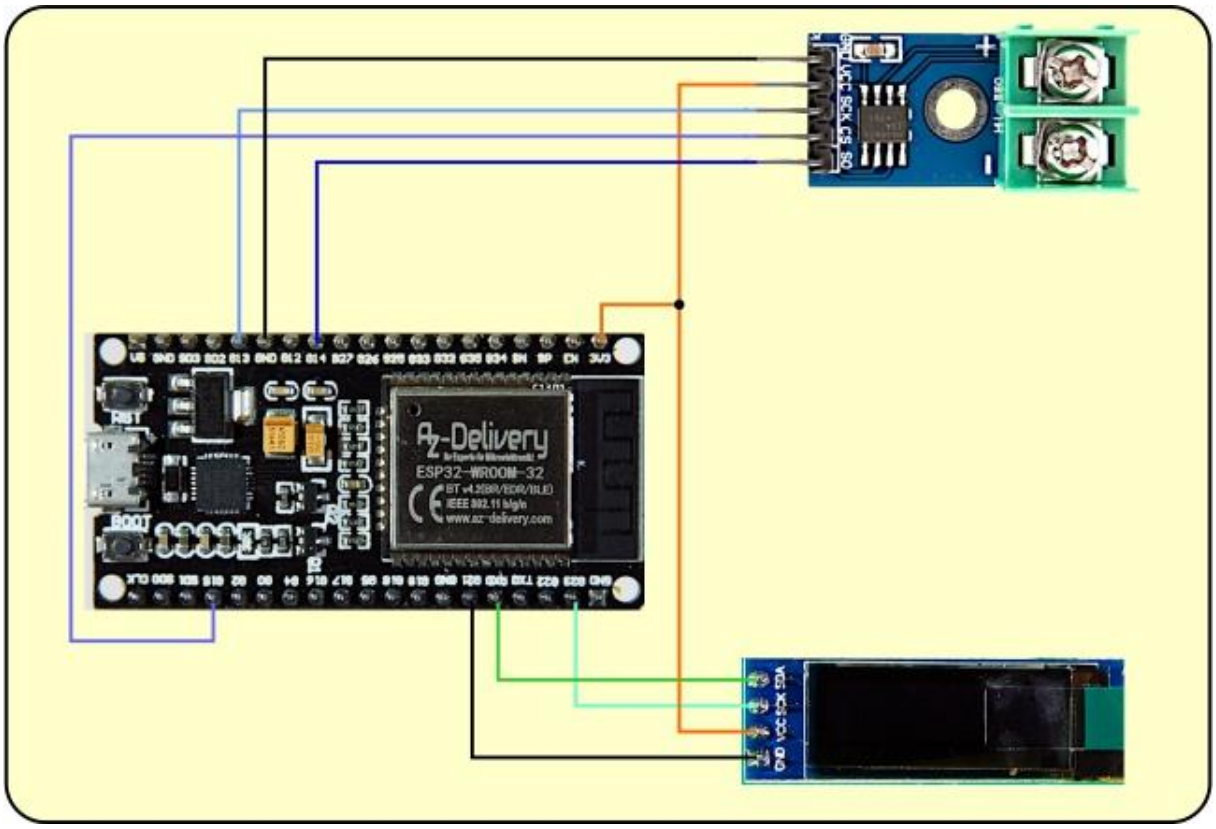

Abbildung 8: MAX6675-Thermocouple - Schaltung mit ESP32

Raspberry Pi Pico (W)

1	Raspberry Pi Pico RP2040 Mikrocontroller-Board oder Raspberry Pi Pico W RP2040 Mikrocontroller-Board
1	MAX6675 Temperatur Sensor mit Sonde K-Typ und Jumper Wire für Raspberry Pi
1	Mini Breadboard 400 Pin mit 4 Stromschienen
1	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F
optional	Logic Analyzer

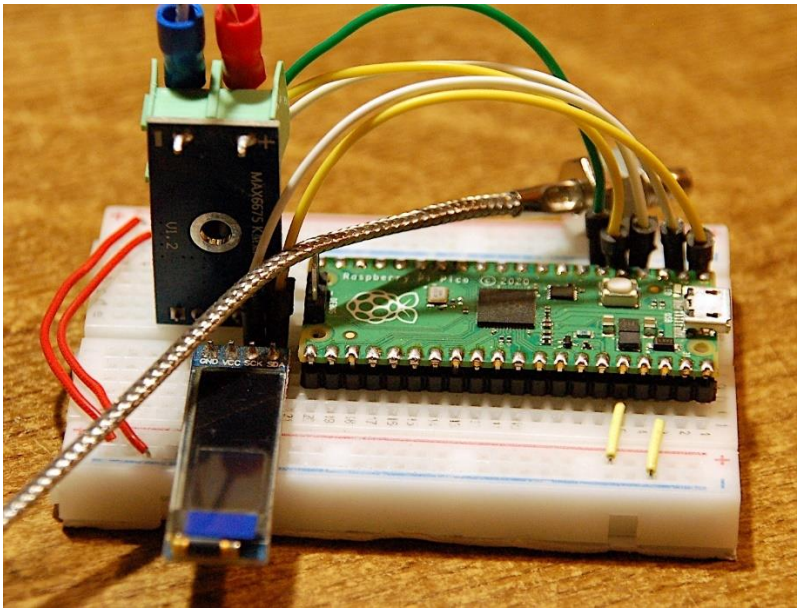

Abbildung 9: Raspberry Pi Pico mit MAX6675

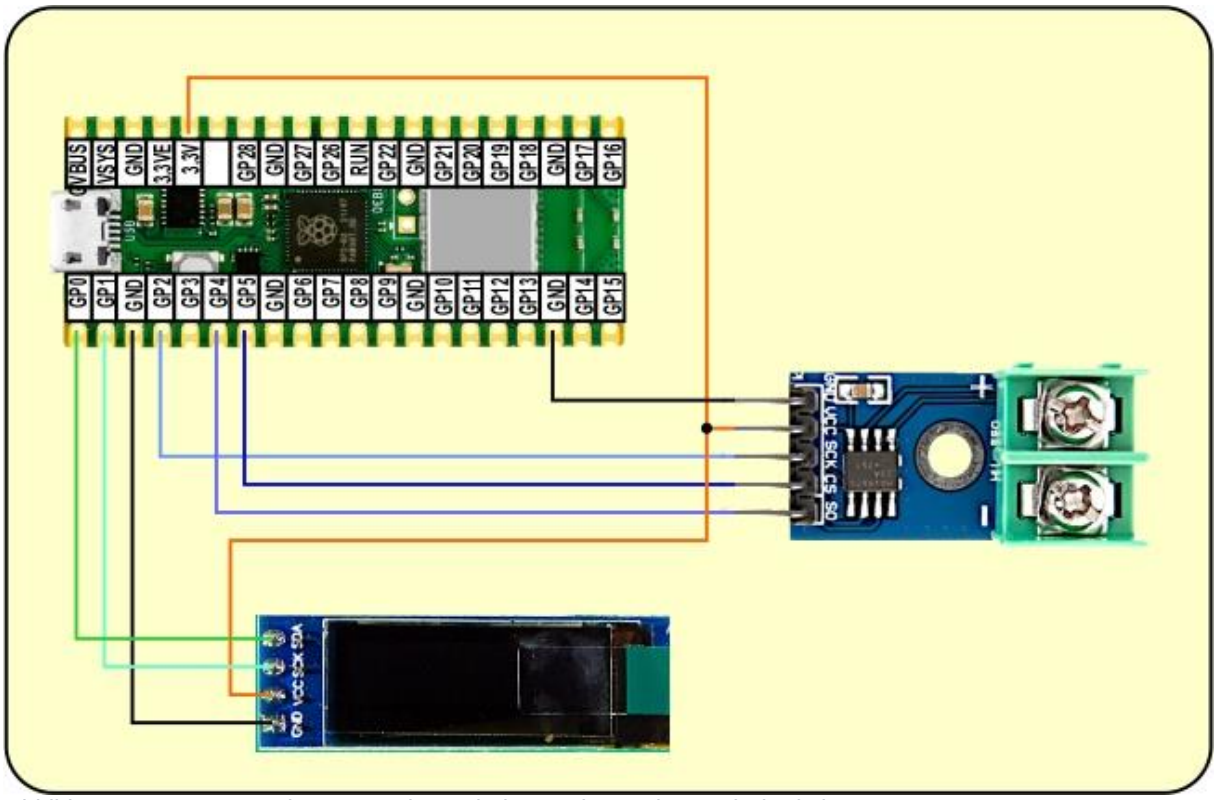

Abbildung 10: MAX6675-Thermocouple - Schaltung mit Raspberry Pi Pico(W)

Probleme bei der Datenübermittlung über den SPI- oder I2C-Bus kann man gut mit einem Logic Analyzer lösen. Zur Darstellung der Pulszüge auf dem PC dient das kostenlose Programm Logic2 von Saleae. Abbildung 11 zeigt die Zustände auf der CLK-, MISO-, CS- und einer Kontrollleitung, die ich während der Entwicklung benutzt habe.

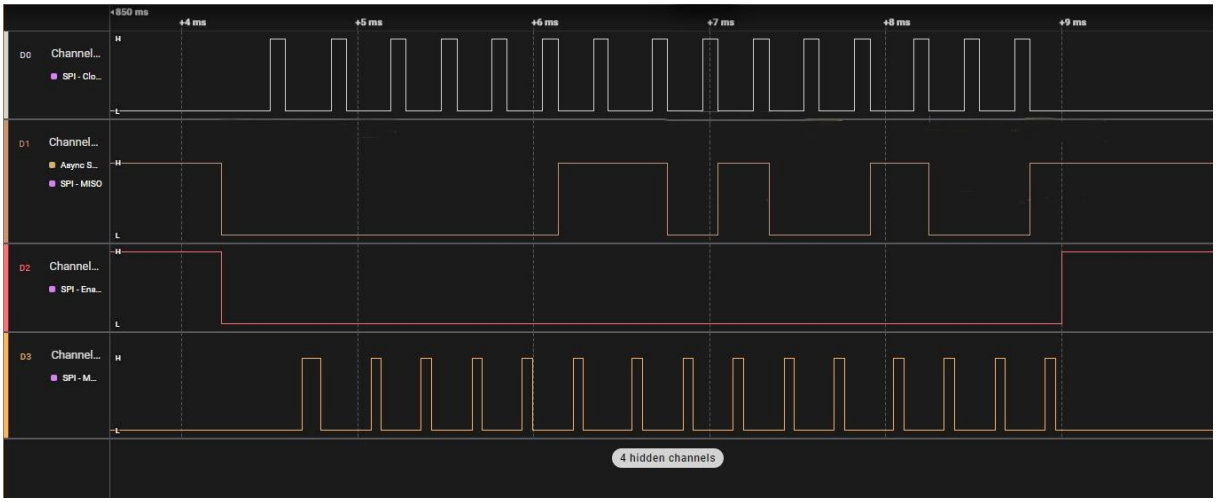

Abbildung 11: Sample mit Kontroll-Peaks

Die Software

Fürs Flashen und die Programmierung des ESP32:

Thonny oder
µPyCraft

Signalverfolgung:

Saleae Logic 2

Verwendete Firmware für einen ESP32:

Micropython Firmware Download
v1.19.1 (2022-06-18) .bin

Verwendete Firmware für einen ESP8266:

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

Verwendete Firmware für den Raspberry Pi Pico W:

RPI_PICO_W-20240105-v1.23.1.uf2

Verwendete Firmware für den Raspberry Pi Pico:

RPI_PICO-20240105-v1.23.1.uf2

Die MicroPython-Programme zum Projekt:

timeout.py: Nichtblockierender Software-Timer
max6675.py: Hardwaretreiber
max6675_spi.py: Hardwaretreiber mit SPI
max6675_thermocouple.py: Beispielprogramm
max6675_thermocouple_SPI.py: Thermometeranwendung
oled.py: OLED-API
ssd1306.py: OLED-Hardwaretreiber

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. Wie Sie den **Raspberry Pi Pico** einsatzbereit kriegen, finden Sie hier.

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.

Das MicroPython-Modul zum MAX6675

Laut Datenblatt wird der MAX6675 über eine abgemagerte Version des SPI-Busses angesprochen. Es gibt nur die Leitungen SCK, MISO und -CS. Man kann also keine Daten an den MAX6675 senden. Eine Messung wird gestartet, indem man -CS auf 0 und gleich danach wieder auf 1 setzt.

Seite 2 des Datenblatts verrät uns, dass die Bereitstellung eines Messwerts dann typischerweise 170ms und maximal 220ms dauert. Mit dem nächsten LOW-Setzen der -CS-Leitung wird das erste von 16-Bits auf die MISO-Leitung gelegt. Die Bits haben folgende Bedeutung.

BIT	DUMMY SIGN BIT	12-BIT TEMPERATURE READING												THERMOCOUPLE INPUT	DEVICE ID	STATE
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	MSB											LSB		0	Three-state

Abbildung 12: Bitbeschreibung

Bit15 ist stets 0 (Vorzeichen-Bit) und kann übergangen werden, weil nur positive Werte gesendet werden. Die nächsten 12 Bits, mit dem MSB voraus, enthalten den Temperaturwert, wie oben bereits beschrieben. Gesampelt wird der Bit-Zug mit der fallenden Flanke des Taktsignals. Die Steigende Taktflanke veranlasst den MAX6675, das nächste Bit auf die MISO-Leitung zu schieben.

Bit2 signalisiert mit einer 1, dass kein Thermoelement angeschlossen ist. Bit1 ist stets 0 und Bit0 wird vom MAX6675 hochohmig gesetzt.

Nach dem Einlesen der Bitfolge wird -CS wieder auf 1 gesetzt und damit eine neue Messung gestartet.

Im Vergleich zu anderen Messwandlern sind diese wenigen Bedingungen recht einfach umzusetzen. Wir brauchen ein paar Importe.

```
from machine import Pin
from time import sleep_us
from timeout import TimeOutMs
```

Dann definieren die Klasse MAX6675 und zwei Klassen-Attribute.

```
class MAX6675:
    messdauer = const(220)
    fehlerbit = const(0)
```

Die Methode __**init**__() stellt den Konstruktor **MAX6675**() der Klasse dar. Ihm werden die Pin-Nummern der Datenleitungen übergeben.

```
    def __init__(self,sck,miso,cs):
        self.sck=Pin(sck,Pin.OUT,value=0)
        self.miso=Pin(miso,Pin.OUT,value=1)
        self.cs=Pin(cs,Pin.OUT,value=0)
        self.fehlerbit=0
        self.temp=0
        self.complete=TimeOutMs(0)
        self.starteMessung()
        print("MAX6675-Objekt bereit")
```

Damit erzeugen wir die entsprechenden Pin-Objekte und deklarieren die Instanz-Attribute **fehlerbit** und **temp**.

TimeOutMs() ist ein nichtblockierender Softwaretimer, den wir benutzen, um die Wandlerzeit zu kontrollieren. Dahinter steckt eine sogenannte Closure. Das ist eine Funktion, die in **TimeOutMs**() gekapselt ist und durch einen Trick weiterlebt, auch

wenn die umgebende Funktion bereits beendet ist. Mehr über Closures erfahren Sie in dem PDF-Dokument Closures und Decorators.pdf. **TimeOutMs**() gibt eine Referenz auf die Closure **compare**() zurück, die wir der Variablen **complete** zuweisen. **complete** kann jetzt als Funktion aufgerufen werden. Der Aufruf von **complete**() liefert **False** zurück, wenn die eingestellte Dauer in Millisekunden noch nicht abgelaufen ist und **True**, wenn die Zeit vorbei ist. Mit 0 als Argument läuft die Zeit nie ab.

Wir starten eine Messung und geben die Instanziierung bekannt.

Den Takt erzeugen wir ganz einfach durch Ansteuern des GPIO-Pins **sck**.

```
    def takt(self):
        self.sck(1)
        sleep_us(2)
        self.sck(0)
        sleep_us(2)
```

Ähnlich sieht das mit dem Starten eines Wandlerprozesses aus. Danach setzen wir den Timer auf die Wandlerzeit von 220ms. Der Timer läuft jetzt im Hintergrund. Wir können ihn überall abfragen und in der Zwischenzeit andere Dinge tun.

```
    def starteMessung(self):
        self.cs(0)
        sleep_us(1)
        self.cs(1)
        self.complete = TimeOutMs(messdauer)
```

An den Status des Fehlerbits kommen wir über die Funktion **error**() heran. Natürlich könnten wir einfach das Instanzattribut **fehlerbit** direkt im Programm referenzieren. Aber das ist nicht die feine Art der objektorientierten Programmierung, einfach über unschuldige Instanzattribute herzufallen.

```
    def error(self):
        return self.fehlerbit
```

Mit der Funktion **read**() holen wir einen Temperaturwert ab. Hier sehen Sie die Anwendung des Timers **complete**(). Wir legen uns so lange auf die faule Haut, bis der Timer abgelaufen ist und **True** zurückgibt. Dann verlassen wir die Schleife, setzen **-CS** auf 0, warten noch kurz und leeren dann die Variable **data**, indem wir alle Bits auf 0 setzen.

```
    def read(self):
        while not self.complete():
            pass
        self.cs(0)
        sleep_us(10)
        data=0
        for i in range(15):
            self.takt()
            data=(data << 1) | self.miso()
        self.cs(1)
        self.temp=(data >> 3) / 4
        self.fehlerbit=(data & 0x04) >> 2
        self.complete = TimeOutMs(messdauer)
        return self.temp
```

Die for-Schleife wird 15-mal durchlaufen. Bit15 wurde ja bereits auf die MISO-Leitung gelegt, als **-CS** auf 0 ging. Wir ignorieren dieses Bit, weil es für uns keinen Nutzen bringt.

Mit einem Takt holen wir das nächste Bit auf die MISO-Leitung. Wir lesen den Leitungszustand ein und oderieren das Bit auf die Variable **data**, in der wir zuvor alle Bits um eine Stelle nach links geschoben haben. Für die restlichen Bits wird der Vorgang wiederholt, dann haben wir alles in der Kiste und können **-CS** wieder auf 1 setzen. Damit startet automatisch die neue Wandlung.

Der Temperaturwert steckt in Bit14:3. Wenn wir die Bits in **data** um 3 Stellen nach rechts schieben, liegt das LSB an Bit-Position 0. Dividieren wir jetzt durch 4, dann erhalten wir die Temperatur als Dezimalzahl mit einer Auflösung von 0,25°C.

Das Fehlerbit isolieren wir durch Undieren mit 0x04. Dadurch wird Bit2 isoliert. Durch Schieben um zwei Positionen nach rechts landet das Bit an Position 0 und liefert den Wert 0 oder 1 an **fehlerbit** ab.

Nach dem Einlesen stellen wir den Timer neu und geben den Temperaturwert zurück.

Mit der Funktion **getTemp**() kann der zuletzt abgeholte Temperaturwert erneut abgefragt werden.

```
    def getTemp(self):
        return self.temp
```

Das war's auch schon. Alles was wir im Beispielprogramm zum Thema MAX6675 brauchen, stellt jetzt die Klasse bereit.

Ein Beispielprogramm

Wie wäre es mit einem Programm, das auf allen drei Controller-Familien läuft? Finger zur Raute – DAS SCHAFFEN WIR!

Beginnen wir mit dem schwächsten Kameraden, dem ESP8266. Er besitzt die wenigsten herausgeführten GPIOs und wird von den meisten Restriktionen eingeengt. Was wir wo anschließen können, zeigen die nächsten Zeilen

```
#LUA-Pins     D0 D1 D2 D3 D4 D5 D6 D7 D8 RX TX
#ESP8266 Pins 16  5  4  0  2 14 12 13 15  3  1
#Achtung      hi       hi hi          lo hi hi
#Belegung        sc sd       cl so cs
```

Als wirklich freiverfügbare Pins bleiben nur GPIO14, 13 und 15 übrig. GPIO4 und 5 brauchen wir für das Display. Die restlichen Pins müssen beim Booten die angegebenen Pegel führen, sonst startet der Controller nicht ordentlich. Also werden beim ESP8266 die SPI-Leitungen wie angegeben belegt.

CLK – GPIO14
MISO – GPIO12
-CS – GPIO13

Beim ESP32 und Raspberry Pi Pico (W) haben wir genug freie Leitungen und können nach Belieben oder anderen Erfordernissen entscheiden, was wo hinkommt.

Doch, importieren wir erst einmal das Nötige. Wir brauchen den **I2C**-Bus, die Klasse **MAX6675**, die **OLED**-API, von **SYS** die Funktion **exit**() und die Konstante **platform**. Die Methode **TimeOut**() ist ein Timer auf Sekundenbasis ähnlich **TimeOutMs**() und **collect**() ist der Besen, mit dem der Datenmüll zusammengekehrt und entsorgt wird.

```
from machine import Pin, SoftI2C
from max6675 import MAX6675
from oled import OLED
from sys import exit, platform
from timeout import TimeOut
from gc import collect
```

Die Konstante **platform** ist für jede Controller-Familie mit einer eindeutigen Kennung belegt. Wir nutzen das, um für jede Familie gesondert I2C-Bus und SPI-Leitungen festzulegen.

```
if platform == "esp8266":
    print("ESP8266-Family")
    scl=Pin(5)
    sda=Pin(4)
    i2c=SoftI2C(scl,sda,freq=100000)
    tc=MAX6675(14,12,13) # sck,miso,cs
elif platform == "esp32":
    print("ESP32-Family")
    scl=Pin(22)
    sda=Pin(21)
```

```
    i2c=SoftI2C(scl,sda,freq=100000)
    tc=MAX6675(13,14,15) # sck,miso,cs
elif platform == "rp2":
    print("RASPI PI PICO-Family")
    scl=Pin(1)
    sda=Pin(0)
    i2c=SoftI2C(scl,sda,freq=100000)
    tc=MAX6675(2,4,5) # sck,miso,cs
else:
    print("Unknown platform")
    exit()
```

Jetzt ist in jedem Fall ein I2C-Bus-Objekt und eine MAX6675-Instanz definiert.

Es folgt ein OLED-Objekt. Dem Konstruktor übergeben wir das I2C-Objekt und sagen im, dass wir ein Display mit 32 Pixel Höhe verwenden.

```
d=OLED(i2c,heightw=32)
d.writeAt("MAX6675-THERMO",1,0)
```

Einige Variablen müssen noch definiert werden. Die Folgezeit für Messungen setzen wir auf 3 Sekunden in **delay** fest und stellen auch gleich den Wecker damit.

```
delay=3
refresh = TimeOut(delay) # Refresh every 3 seconds
n=50
l=[0 for _ in range(n)]
runde=0
mini=1023.75
maxi=0
```

Wir werden jeweils den gleitenden Mittelwert aus den letzten **n** = 50 Messungen berechnen. Dafür benutzen wir die Liste **l**, die wir mit 50 Nullen vorbelegen. Das Mittel meiner Wahl war eine List-Comprehension. Weil dazu keine spezielle Variable nötig ist, benutze ich den Unterstrich für die Iteration.

Außerdem brauchen wir einen Rundenzähler und Speicher für den kleinsten und größten Messwert. Wir können damit für mehrere jeweils feste Temperaturen die Abweichung vom Mittelwert als wahrscheinlichstem Wert feststellen und dadurch Aufschluss über die Genauigkeit der Messwerte erhalten. Für die Anwendung der Schaltung als dynamisch arbeitendes Thermometer muss die Mittelwertbildung anders erfolgen. Wir werden später das Programm dahingehend ändern.

```
while 1:
    if refresh():
        collect()
        tc.starteMessung()
        t=tc.read()
        refresh = TimeOut(delay)
        print(t)
        l.append(t)
        l.pop(0)
```

In der Hauptschleife reagieren wir auf das Auftreten des Timer-Events **refresh**(). Es liefert **True**, wenn der Timer abgelaufen ist. Dann sammeln wir den Datenmüll, starten eine Messung und holen das Ergebnis ab. Der Wecker wird neu gestellt und der Messwert ausgegeben. Wir hängen ihn an die Liste hinten an und entfernen dafür das erste Element. Nach 50 Durchläufen ist die Liste komplett gefüllt,

```
        runde+=1
        s=0
        for i in range(n):
            s+=l[i]
        m = s / runde if runde < n else s / n
```

Rundenzähler erhöhen, Summenwert auf 0 und Aufaddieren der Listenelemente. Beim Berechnen des Mittelwerts verwenden wir bis zur 50. Runde den Rundenzähler als Divisor, danach unser **n**.

Wir lassen uns die Nummer der Runde, den Mittelwert und die Liste in REPL ausgeben. Die auf zwei Nachkommastellen normierten Werte der aktuellen Temperatur und des Mittelwerts erscheinen auch im Display,

```
        print(runde,m,l)
        d.writeAt("Temp: {:3.2f} *C".format(t),1,1)
        d.writeAt("Mittel: {:3.2f} *C".format(m),0,2)
        mini=min(mini,t)
        maxi=max(maxi,t)
        dmin=m - mini
        dmax=maxi - m
        print(dmin, m, dmax,"\n")
```

Wir berechnen ferner den minimalen und maximalen Messwert sowie die Abweichungen vom Mittelwert. Mit steigender Anzahl an Messungen pendeln sich die Abweichungen auf bestimmte Grenzwerte ein.

Bei einer Temperatur von ca. 25°C erhielt ich, bei mehr als 1000 Messungen, Abweichungen von -1,28°C und +0,47°C von 25,53°C. Das entspricht -5,0% / +1,8%.

Im weitern Verlauf der Hauptschleife könnten nun, über ähnliche Timer oder andere Ereignisse gesteuert, weitere Features in das Programm eingebaut werden, etwa:

- Abfragen weiterer Sensoren
- z.B. Schalten bei Unter-/Überschreitung eines Limits
- Senden von Werten via WLAN
- Schreiben in eine Datei auf SD-Card in festen Abständen

Zum Testen des Programms und des Moduls kann nun jeder der drei Aufbauten hergenommen werden.

Das etwas andere Programm

Im Datenblatt des MAX6675 steht nun auch, dass der Bus SPI-compatibel wäre. Das ist doch eine Herausforderung. Als Erstes ändern wir das Modul ein wenig ab. Die Stellen im Listing sind fett formatiert.

```
from machine import Pin, SPI
from time import sleep_us
from timeout import TimeOutMs

class MAX6675:
    messdauer = const(220)
    fehlerbit = const(0)

    def __init__(self,sck,mosi,miso,cs):
        self.spi=SPI(0,baudrate=1000000,
                     sck=Pin(sck),
                     mosi=Pin(mosi),
                     miso=Pin(miso))
        self.cs=Pin(cs,Pin.OUT,value=0)
        self.complete=TimeOutMs(0)
        self.starteMessung()
        self.fehlerbit=0
        self.temp=0
        print("MAX6675-Objekt bereit")

    def takt(self):  # kann entfernt werden
        self.sck(1)
        sleep_us(2)
        self.sck(0)
        sleep_us(2)

    def starteMessung(self):
        self.cs(0)
        sleep_us(1)
        self.cs(1)
        self.complete = TimeOutMs(messdauer)

    def error(self):
        return self.fehlerbit

    def read(self):
        while not self.complete():
            pass
        self.cs(0)
        sleep_us(10)
        msB,lsB=self.spi.read(2)
        data=(msB << 8) | lsB
        self.cs(1)
        self.temp=(data >> 3) / 4
        self.fehlerbit=(data & 0x04) >> 2
        self.complete = TimeOutMs(messdauer)
```

```
        return self.temp

    def getTemp(self):
        return self.temp
```

Die Taktung übernimmt jetzt die Klasse **SPI**, ebenfalls das Shiften der Bits. Das **bytes**-Objekt, das uns **spi.read**() liefert, entpacken wir in derselben Zeile in msByte und lsByte. Das msByte schieben wir um 8 Positionen nach links und oderieren mit dem lsByte – fertig. Alles andere folgt wie gehabt.

Das Listing speichern wir als max6675_spi.py ab.

Ich habe oben versprochen, ein Programm für ein Thermometer vorzustellen. Bitteschön, hier ist es, zusammen mit dem geänderten Modul - max6675_thermocouple_SPI.py.

```
from machine import Pin, SoftI2C
from max6675_spi import MAX6675
from oled import OLED
from sys import exit, platform
from gc import collect

if platform == "esp8266":
    print("ESP8266-Family")
    scl=Pin(5)
    sda=Pin(4)
    i2c=SoftI2C(scl,sda,freq=100000)
    tc=MAX6675(14,12,13)
elif platform == "esp32":
    print("ESP32-Family")
    scl=Pin(22)
    sda=Pin(21)
    i2c=SoftI2C(scl,sda,freq=100000)
    tc=MAX6675(13,14,15) # sck,miso,cs
elif platform == "rp2":
    print("RASPI PI PICO-Family")
    scl=Pin(1)
    sda=Pin(0)
    i2c=SoftI2C(scl,sda,freq=100000)
    tc=MAX6675(2,3,4,5) # sck,miso,cs
else:
    print("Unknown platform")
    exit()

d=OLED(i2c,heightw=32)
d.writeAt("MAX6675-THERMO",1,0)

n=10
while 1:
    collect()
    s=0
    for i in range(n):
```

```
        t=tc.read()
        s+=t
    m = s / n
    d.writeAt("Mittel: {:3.1f} *C".format(m),0,2)
    # Abfragen weiterer Sensoren
    # z.B. Schalten bei Unter-/Ueberschreitung eines Limits
    # Senden von Werten
    # Schreiben in eine Datei auf SD-Card in festen Abständen
```

Nicht umsonst habe ich bewusst die Anschlüsse am Raspberry Pi Pico so gewählt, dass die GPIOs zu den Leitungen der SPI0-Schnittstelle passen. So war ein Umstecken für den Test des SPI-Moduls nicht nötig. Die Klasse **max6675.py** ist damit ein Beispiel für ein Software-SPI-Read-Only-Interface.

Fassen wir zusammen

Der MAX6675 zusammen mit dem Thermoelement ist zwar nicht die schnellste Hardware für ein Thermometer und auch nicht die genaueste Lösung, aber es ein Ansatz, mit dem Temperaturen bis 1000°C erfasst werden können. Zumindest bis 700°C arbeitet das System relativ zuverlässig. Arbeitet man bei Temperaturen zwischen 700°C und 1000°C, dann findet beim Abkühlen unter 700°C in der CrNi-Legierung eine Umgruppierung der Atomverteilung statt, die zu falschen Ergebnissen führen kann.

Ansonsten könnte ich mir den Einsatz des Thermometers als Messstelle in Emaile-Öfen oder Brennöfen vorstellen. In Flüssigkeits- oder Gastanks ist ein Einsatz nur möglich, wenn es gelingt, den Spalt zwischen dem Sensor und der umgebenden Hülle mit dem 6mm-Gewinde hitzefest zu verschließen.