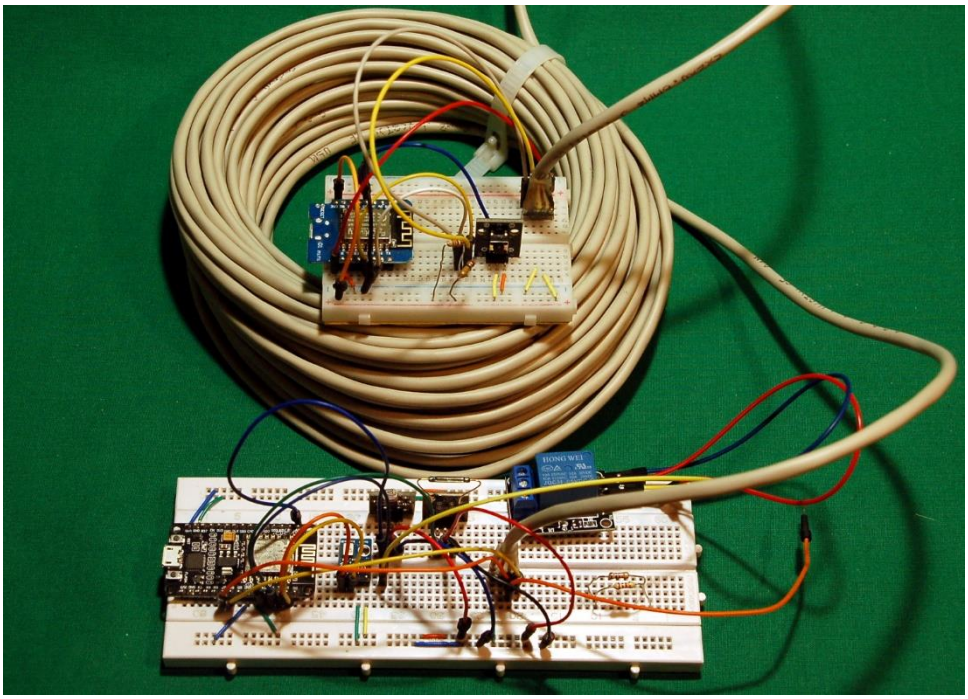

Gesamter Aufbau

Diesen Beitrag gibt es auch als PDF-Dokument zum Download.

Was tun, wenn im Keller eine Reihe von Sensoren Daten in die Wohnung im 2. Stock liefern soll, die Geschoßdecken aber den Einsatz eines Funknetzes nicht erlauben. Glück hatte ich in diesem Fall, weil mir ein nicht mehr benötigtes vieradriges Kabel einer alten Gegensprechanlage wieder in den Sinn kam. Was liegt näher, als darüber eine Datenleitung und eventuell Spannungsversorgung der Einheit im Keller zu etablieren.

Also flugs den elektrischen Widerstand der Leitung gemessen, für Hin- und Rückleitung 1,5 Ohm. Bei einer Stromaufnahme von maximal 100mA der Slave-Einheit im Keller bedeutet das einen Spannungsabfall von 0,15V. Von den 5V, die das Steckernetzteil liefert, bleiben dem ESP8266 also noch 4,85V übrig. Prima, das könnte funktionieren.

Aber welches Übertragungsprotokoll kann ich verwenden? RS232 am ESP8266 geht nicht, weil keine zweite serielle Schnittstelle zur Verfügung steht. Also I2C – aber nein - der ESP8266 kann mit dem, was MicroPython im Kernel mitbringt, keinen I2C-Slave spielen, der ESP32 übrigens auch nicht. Außerdem, 20m Kabellänge und I2C verträgt sich wohl ebenfalls nicht. Ein I2C-Slave-Modul für MicroPython im Internet suchen – Fehlanzeige! Na gut, geht nicht – gibt's nicht, also schreibe ich selbst ein solches Modul. Das Protokoll ist mir bekannt und das Ganze im **Master** in eine Signalfolge zu übersetzen, das ist sowohl beim ESP32 wie auch beim ESP8266 im Kernel verankert. Im Slave geht es darum diese Signalfolge abzutasten und Daten an den Master zurückzusenden. Und - es hat funktioniert. Allerdings schafft MicroPython dabei leider keine Taktrate von 100kHz, sondern grade mal bis zu 500Hz. Das liegt an der Arbeitsgeschwindigkeit des ESP8266 unter MicroPython aber auch an der Kabellänge. Aber was soll's, es werden ja keine Brockhausbände übertragen, sondern nur kurze Datenschnippsel. Eine Taktrate von 500 bringt ca. 50 Bytes pro Sekunde auf den Bus. Das reicht für die Übertragung von ein paar Kommandos oder Messwerten.

Wie das Vorhaben umgesetzt wurde, erfahren Sie in dieser Blogfolge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

ESP8266 und ESP32 als I2C-Slave

Bei beiden Controllerfamilien muss man sich erst einmal Gedanken zu den verwendbaren GPIO-Pins machen. Einige davon sind in den Boot-Prozess eingebunden und daher nicht uneingeschränkt einsetzbar. Die beiden Tabellen geben darüber Auskunft.

Label	GPIO	Input	Output	Notes
D0	GPIO16	Kein IRQ	Kein PWM or I2C support	HIGH at boot wake up from deep sleep
D1	GPIO5	OK	OK	SCL bei I2C-Nutzung
D2	GPIO4	OK	OK	SDA bei I2C-Nutzung
D3	GPIO0	pulled up	OK	FLASH button, wenn LOW Normales Booten, wenn HIGH
D4	GPIO2	pulled up	OK	Muss beim Booten HIGH sein verbunden mit der On-Board-LED, LOW aktiviert LED
D5	GPIO14	OK	OK	SPI (SCLK)
D6	GPIO12	OK	OK	SPI (MISO)
D7	GPIO13	OK	OK	SPI (MOSI)
D8	GPIO15	pulled to GND	OK	SPI (CS) Muss beim Booten LOW sein
RX	GPIO3	OK	REPL	Muss beim Booten HIGH sein
TX	GPIO1	TX pin	REPL	Muss beim Booten HIGH sein Debugausgang beim Booten
A0	ADC0	Analog Input	X	

Tabelle 1: Pinbelegung und Systemfunktionen beim ESP8266

GPIO	Input	Output	Notes
0	pulled up	OK	Muss beim Booten HIGH sein
1	TX pin	REPL	
2	pulled up	OK	Muss beim Booten HIGH sein
3	RX pin	REPL	Muss beim Booten HIGH sein
4	OK	OK	
5	OK	OK	
6	x	x	SPI flash
7	x	x	SPI flash
8	x	x	SPI flash

9	x	x	SPI flash
10	x	x	SPI flash
11	x	x	SPI flash
12	OK	OK	Darf beim Booten nicht HIGH sein
13	OK	OK	
14	OK	OK	
15	OK	OK	
16	OK	OK	
17	OK	OK	
18	OK	OK	
19	OK	OK	
21	OK	OK	
22	OK	OK	
23	OK	OK	
25	OK	OK	
26	OK	OK	
27	OK	OK	
32	OK	OK	
33	OK	OK	
34	OK		Nur Eingang
35	OK		Nur Eingang
36	OK		Nur Eingang (VP)
39	OK		Nur Eingang (VN)

Tabelle 2: Pinbelegung und Systemfunktionen beim ESP32

Da die SDA- und SCL-Leitung mit einem Widerstand von 10kΩ an +Vcc gezogen werden müssen (Pullups), kann man dafür kein GPIO-Pin verwenden, das beim normalen Bootvorgang nicht HIGH sein darf. Beim ESP8266 ist das D8 = GPIO15 und beim ESP32 ist es GPIO12.

Wenn der Slave außerdem in seiner eigenen Sensor-Umgebung auch noch einen I2C-Master abgeben soll, ist es sinnvoll, die dafür üblichen GPIOs zu verwenden. Beim ESP8266 sind das D1(GPIO5=SCL) und D2 (GPIO4=SDA) und beim ESP32 GPIO22=SCL und GPIO21=SDA.

Ich werde in diesem Projekt einen DHT20 zusammen mit einem Relais am Slave, einem ESP8266, einsetzen. Das Hauptaugenmerk dieses Beitrags liegt auf dem Einrichten eines Controllers als I2C-Slave. Welche Arbeiten der Sklave zu verrichten hat, kann von Fall zu Fall variieren. Deshalb haben die Beschaltung und die Auswahl der Sensormodule und Aktoren nur Beispielcharakter, um das Zusammenspiel der verschiedenen Komponenten zu demonstrieren. Neben dem Relais werden vom Slave noch ein DHT20, ein LDR (Light Dependend Resistor) und ein Reedkontakt bedient. Reedkontakte sind Schalter, die durch ein äußeres Magnetfeld geschlossen werden. Mein Modul ist dem Aufdruck auf der Platine entsprechend angeschlossen "-" an GND, der mittlere Kontakt an +3,3V und "S" liegt an D4=GPIO2.

Beim LDR-Modul liegt "-" an +3,3V, der Mittelkontakt an GND und "S" an A0. Auf diese Weise bekomme ich bei größerer Helligkeit höhere Wandlerwerte. Die Umpolung am Modul ist möglich, weil der LDR als Widerstand keine Polung aufweist, im Gegensatz zu Fotodiode oder Fototransistor.

Beginnen wir mit der Hardwareliste.

Hardware

Welcher Controller als Master oder Slave eingesetzt wird, steht zur freien Auswahl. Natürlich werden zwei Module gebraucht, ein Master und ein Slave. Ich habe für den Slave einen ESP8266-Amica und für den Master einen ESP8266 D1 mini verwendet, die waren grade zur Hand. Der Slave verwendet neben der Übertragung über das 20m-Kabel mit 100Hz auch noch den systemeigenen I2C-Bus mit 100kHz zur Plausch mit dem DHT20. Anders als DHT11 und DHT22, die über ein eigenes Protokoll mit einer einzigen Datenleitung arbeiten, benutzt der DHT20 den I2C-Bus. Weil er keine Jumper zum Verändern der Geräteadresse (0x38) besitzt, kann nur einer dieser Bausteine am Bus betrieben werden. Beim Durcharbeiten des Datenblatts und während der Programmierung eines entsprechenden MicroPython-Moduls stellte sich heraus, dass der Baustein große Ähnlichkeiten mit dem AHT10 aufweist. Außer der gleichen Geräteadresse ist auch das Handling identisch.

2	D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder D1 Mini V3 NodeMCU mit ESP8266-12F oder NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F WIFI oder NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F WIFI oder 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	Fotowiderstand Photo Resistor
1	KY-021 Magnet Schalter Mini Magnet Reed Modul Sensor
1	DHT20 Digitaler Temperatursensor und Luftfeuchtigkeitssensor
1	MB-102 Breadboard Steckbrett mit 830 Kontakten
diverse	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F evtl. auch 65Stk. Jumper Wire Kabel Steckbrücken für Breadboard
optional	Logic Analyzer

Die Schaltungen

Für das Projekt werden zwei Schaltungen benötigt und ein vierpoliges Kabel vom Master zum Slave. Getestet habe ich mit 20m Klingelleitung 4 mal 0,6mm². Eine Versorgung des Slaves über dieses Kabel ist möglich, wenn der Master mit einem 5V-Stecker-Netzteil versorgt wird und der Betrieb läuft autonom ohne PC, wenn die Betriebsprogramme, wie oben ausgeführt, als **main.py** in die Controller hochgeladen werden.

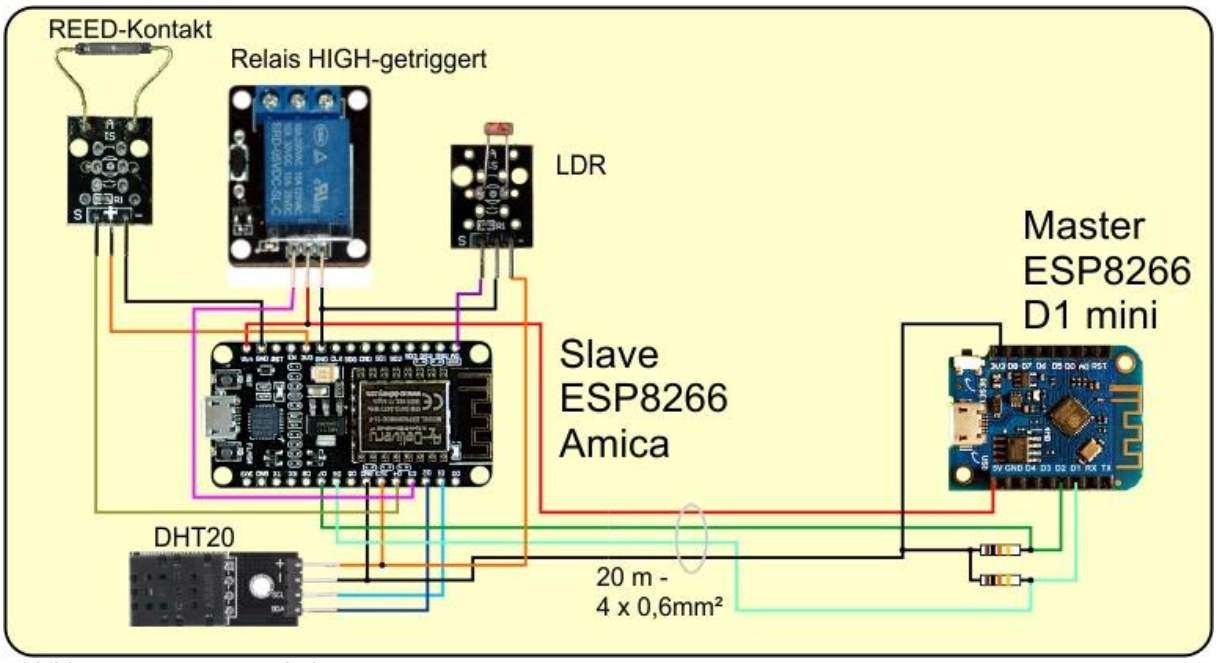

Abbildung 1: Gesamte Schaltung

Der linke Teil in Abbildung1 stellt den Slave dar, der rechte Teil den Master. Beide Teile können mit entsprechender Anpassung der GPIO-Pins auch auf ESP32-Einheiten übertragen werden, falls mehr als die noch freien Anschlüsse gebraucht werden sollten. Was der ESP8266 kann, kann ein ESP32 allemal.

Ein Hinweis zum Relais ist unter Umständen wichtig. Sollten Sie ein Modul mit einem LOW-getriggerten Relais benutzen, beachten Sie bitte, dass

- die Module in der Regel mit 5V versorgt werden müssen und
- die Eingänge über Pullup-Widerstände auf 5V hochgezogen werden.

Das gefährdet die GPIOs eines ESP32/ESP8266. Daher müssen die Pegel mit einer Transistorstufe wie in Abbildung 2 von 3,3V auf 5V angepasst werden. Das betrifft, wie ich festgestellt habe sowohl magnetische Relais-Module als auch Solid-State-Module wie in Abbildung 2.

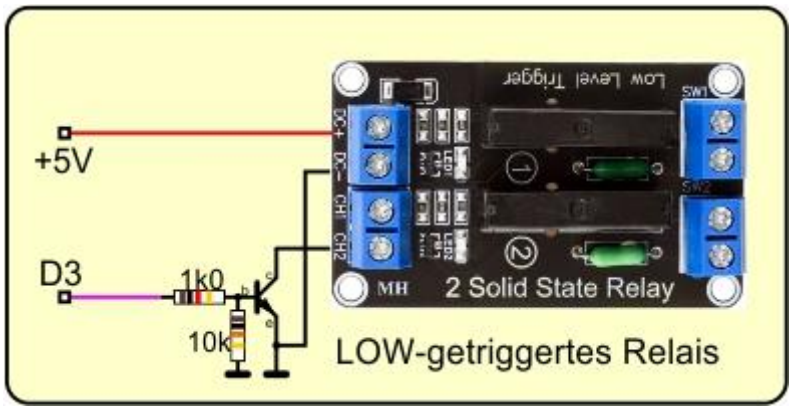

Abbildung 2: Schaltung für ein LOW-getriggertes Relais

In den folgenden Ausführungen wird der Aufbau der Schaltungen vorausgesetzt. Solid-State-Module können ausgangsseitig übrigens bauteilbedingt nur Wechselstrom-Leitungen schalten, elektromagnetische Relais können Gleich- und Wechselstrom schalten.

Die Software

Fürs Flashen und die Programmierung des ESP32:

Thonny oder
µPyCraft

Verwendete Firmware für den ESP32:

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

Verwendete Firmware für den ESP8266:

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

Die MicroPython-Programme zum Projekt:

dht20.py Treibermodul für den DHT20
slave.py Demoprogramm für den MicroPython-Slave
i2cslave.py Sklaventreiber
master.py Demoprogramm für den MicroPython-Master
timeout.py Nichtblockierender Software-Timer

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

Immer wenn es Probleme bei der Datenübertragung gibt, setze ich gerne das DSO (Digitales Speicher Oszilloskop) ein, oder ein um Welten billigeres, kleines Tool, einen Logic-Analyzer (LA) mit 8 Kanälen. Das Ding wird an den USB-Bus angeschlossen und zeigt mittels einer kostenlosen Software, was auf den Busleitungen los ist. Dort, wo es nicht auf die Form von Impulsen ankommt, sondern lediglich auf deren zeitliche Abfolge ist ein LA Gold wert. Und, während das DSO nur Momentaufnahmen des Kurvenverlaufs liefert, kann man mit dem LA über längere Zeit abtasten und sich dann in die interessanten Stellen hineinzoomen. Eine Beschreibung zu dem Gerät finden Sie übrigens in dem Blogpost "Logic Analyzer - Teil 1: I2C-Signale sichtbar machen" von Bernd Albrecht. Dort ist auch beschrieben, wie man den I2C-Bus abtastet. Am abgegriffenen Label des Tools in Abbildung 3 können Sie sehen, dass der Logic Analyzer häufig in Benutzung ist.

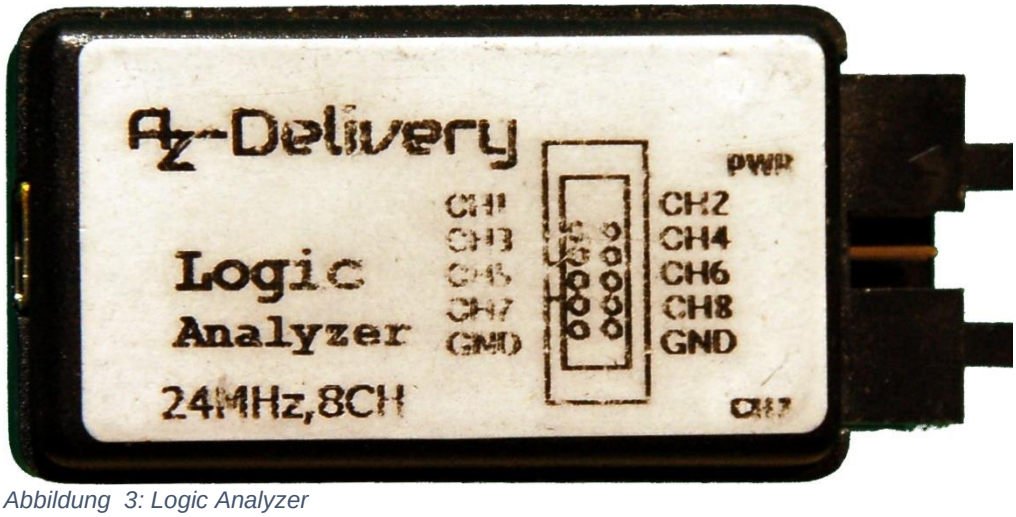

Abbildung 3: Logic Analyzer

Ich stelle Ihnen hier einmal auszugsweise die Übertragung der Hardware-Adresse (HWADR) an den DHT20, gefolgt von einem Daten-Byte vor. Dazu lege ich in der Slave-Schaltung den Pin 1 des Logic Analyzers mit einem Jumperkabel an die SCL-Leitung und den Pin 2 des Logic Analyzers an die SDA-Leitung des I2C-Busses. GND verbinde ich mit GND.

Nun starte ich Thonny, lege eine neue Datei im Editor an und gebe den folgenden Text ein. Speichern Sie das Prográmmchen unter einem beliebigen Namen in Ihrem Arbeitsverzeichnis.

```
from machine import Pin,SoftI2C
import sys

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")
```

Die Variable **sys.platform** sagt uns, welchen Controllertyp wir verwenden. Davon abhängig wird ein I2C-Objekt instanziiert, welches wir gleich zu einem ersten Test verwenden werden. Ist der ESP8266 und seine Beschaltung bereit und der Controller mit dem PC verbunden? Dann starten wir das Programm mit der F5-Taste. Das geht schneller als mit der Maus den grünen Startbutton mit dem weißen Dreieck anzufahren und zu klicken. Läuft das Programm ohne Fehlermeldung durch, dann ist alles OK. Im Terminal geben wir jetzt den ersten I2C-Befehl ein, wollen sehen, wer denn so alles da ist. Eingaben formatiere ich fett, die Antworten vom System kursiv.

>>> i2c.scan()
[56]

Die eckigen Klammern stellen in MicroPython eine sogenannte Liste dar. Sie enthält als Elemente die 7-Bit-Hardwareadressen der gefundenen I2C-Bausteine. 56 =0x38 ist die Nummer auf die der DHT20 reagiert. Der Bus steht bereit und wartet auf die Kommunikation des ESP8266 mit den angeschlossenen Parteien.

Damit die Buschtrommel funktioniert, muss es einen geben, der den Takt angibt, das ist der ESP8266, er ist der Chef und der heißt im lokalen System des Slaves bei I2C Master. Der DHT20 ist lokal ein Sklave, ein Slave. - Ohhh! Ich dachte, das Zeitalter der Sklaverei ist schon lange vorbei! – Sei' drum, der Master gibt an, mit welchem Slave er zu parlieren wünscht. Dazu erzeugt er eine **Start Condition** als eine Art "Achtung an alle"-Nachricht. Dann legt er die Hardware-Adresse auf den Bus, an die er als LSB (Least Significant Bit) eine 0 anhängt, wenn er dem Slave Daten schicken möchte (Schreiben) oder eine 1, wenn er vom Slave eine Antwort erwartet (Lesen). Der Hardware-Adresse folgt im Falle eines Schreibzugriffs das zu sendende Datenbyte. Zum Abschluss kommt als "OK, das war's", eine **Stop Condition**. Wie die Signalfolge aussieht, das schauen wir uns gleich an.

Ich gebe jetzt im Terminalbereich von Thonny den folgenden Befehl ein, aber schicke ihn noch nicht ab. Der Logic Analyzer ist angeschlossen wie oben beschrieben.

>>> i2c.writeto(0x38,b"\xBA")

Als nächstes starte ich das Programm Logic 2. Im Menü am rechten Fensterrand klicke ich auf **Analyzers** und dann auf das Pluszeichen. Aus der Liste wähle ich **I2C**. Wenn diese Betriebsart noch nie benutzt wurde, muss man im nächsten Fenster angeben, welche Analyzer-Leitung an welcher Busleitung liegt.

Jetzt ist alles vorbereitet, wir starten den Analyzer mit der Taste **R.** Die Aufzeichnung beginnt und wir wechseln schnell zu Thonny und drücken die Entertaste, um den Befehl abzuschicken. Danach wieder zurück zu Logic 2 und mit **R** die Aufzeichnung stoppen.

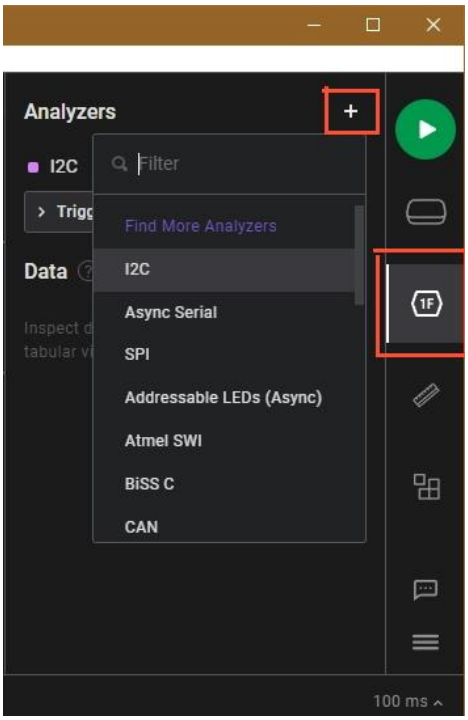

Abbildung 4: Logic 2 - Analyzers - I2C

Um die gesamte Signalfolge zu sehen, setze ich den Mauszeiger auf eine der Signalbahnen und drehe das Mausrad zu mir. Irgendwann taucht ein vertikaler Strich in den Aufzeichnungsbahnen auf. Dann setze ich den Mauszeiger auf diesen Strich und drehe das Mausrad von mir weg. Der Strich wird immer breiter, bis ich die Signalpulse erkennen kann. Ich habe die wichtigsten Stellen mit Zeitmarken gekennzeichnet.

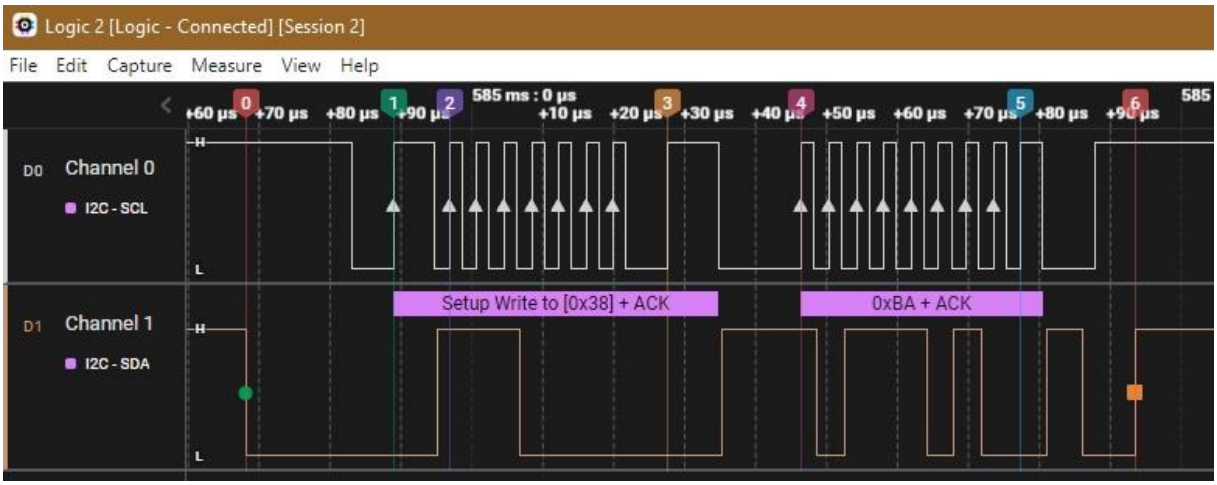

Abbildung 5: Softreset des DHT20

Die Marke 0 kennzeichnet die **Start Condition**, SDA geht auf LOW, während SCL HIGH ist. Dann wartet der Master ca 15µs, damit die Slaves aus den Federn kommen, um eine Hardware-Adresse zu empfangen. Wir haben 0x38 = 0b00111000 angegeben. Der ESP8266 macht daraus durch Linksschieben der Bits 0b011100 und hängt als LSB eine 0 an, weil er dem DHT20 ein Byte senden möchte, das ergibt 0x70 = 0b01110000. Dann legt er das Adressbyte auf den Bus. Mit jeder steigenden Flanke (zum Beispiel Marke 1 und 2) seines Taktsignals auf SCL sagt er den Slaves, dass sie sich den Zustand der SDA-Leitung jetzt merken sollen. In Abbildung 5 können Sie ablesen, dass der Master das Byte 0x70 = 0b01110000 gesendet hat. Mit dem Acknowledge-Bit (ACK) signalisiert der Slave an der neunten steigenden Flanke, ob er das Byte vom Master empfangen und als seine Adresse erkannt hat. In diesem Fall zieht er, wie hier der DHT20, die SDA-Leitung auf 0. Eventuelle weitere Slaves am Bus ziehen ihre Schlafmützen wieder über die Ohren und pennen weiter, weil sie nicht gemeint sind. Auf die gleiche Weise sendet der Master als nächstes das Byte 0xBA, das der DHT20 wieder mit einem ACK quittiert. Danach gibt der Slave die SDA -Leitung wieder frei. Der Master erzeugt keinen neuen Taktimpuls, SCL bleibt auf 1. Wenn jetzt der Master die SDA-Leitung auch freigibt, freigegebene Leitungen (auf Eingang geschaltet) gehen durch die Pullup-Widerstände auf HIGH-Pegel, ist das die **Stop Condition**. Nach diesem Schema arbeiten die I2C-Routinen des MicroPython-Moduls **dht20**.py und natürlich auch die unseres Moduls **i2cslave**.py.

Das Modul i2cslave.py

An bestimmten Stellen muss sich der Controller kurz schlafen legen, um der Peripherie Zeit zum Erledigen des Messjobs zu geben. Dafür importieren wir **sleep** vom Modul **machine**.

```
from time import sleep

class DHT20_Error(Exception):
    pass

class CalibrationError(DHT20_Error):
    def __init__(self):
        super().__init__("Calibrierung fehlgeschlagen")
```

Von der Klasse **Exeption** leiten wir die Klasse **DHT20_Error** ab, von welcher **CalibrationError** erbt. Eine **CalibrationError**-Exception werfen wir, sollte die Calibrierung des DHT20 fehlschlagen.

Die Klasse **DHT20** erbt von **DHT20_Error**. Wir legen die Geräteadresse des DHT20 fest und deklarieren die drei Kommandobytes als Konstanten. Die Status-Flags **Busy** und **Calibrated** finden sich im Statusbyte, das als erstes Byte nach einem Triggerbefehl eingelesen wird.

```
class DHT20(DHT20_Error):
    DHT_HWADR=const(0x38)
    # Commands
    cmdCalibrate=const(0xE1)
    cmdTrigger=const(0xAC)
    cmdReset=const(0xBA)
    #Flags
    Busy=const(0x80)
    Calibrated=const(0x08)
```

Den Konstruktor der Klasse DHT20, die Routine **__init__()** übernimmt ein I2C-Bus-Objekt und wartet erst einmal 100ms bis der Baustein DHT20 sich eingerichtet hat. I2C-Schreib- und Leseanweisungen arbeiten ausschließlich mit Datenstrukturen, die das Bufferprotokoll unterstützen. Das tun Bytearrays und Bytes-Objekte. Zum Empfang der Rohmesswerte deklarieren wir daher ein Bytearray mit sechs Bytes an Speicherplatz. Die Attribute **temp** und **hum** werden deklariert, und die Referenz auf das I2C-Objekt weisen wir dem Instanz-Attribut **i2c** zu. Dann rufen wir die Methode **calibrate**() auf, um den DHT20 zu initialisieren.

```
    def __init__(self,i2c):
        sleep(0.1)
        self.data=bytearray(6)
        self.temp=None
        self.hum=None
        self.i2c=i2c
        self.calibrate()
```

Die Methode **writeRegs**() nimmt ein Kommando-Byte und ein Bytearray **dat** mit weiteren Daten-Bytes. Beides wird zu einem namenlosen Bytearray zusammengeführt und an den DHT20 gesendet.

```
    def writeRegs(self, cmd, dat):
        self.i2c.writeto(DHT_HWADR,bytearray([cmd])+dat)
```

Mit **reset**() senden wir einen Soft-Reset-Befehl an den DHT20.

```
    def reset(self):
        self.i2c.writeto(DHT_HWADR,bytearray([cmdReset]))
```

Bevor der DHT20 Messwerte liefern kann, muss seine Anwesenheit festgestellt und laut Datenblatt eine Kalibrierung durchgeführt werden. Das macht die Methode **calibrate**().

```
    def calibrate(self):
        data=bytearray((0x08,0x00))
        self.writeRegs(cmdCalibrate,data)
        status=self.readStatus()
        while status & Busy:
            sleep(0.01)
            status=self.readStatus()
        return bool(status & Calibrated)
```

Neben dem Kommando **cmdCalibrate** = 0xE1 sind die Bytes 0x08 und 0x00 zu übermitteln. Wir wandeln das Tupel (0x08,0x00) in ein Bytearray um und schicken es zusammen mit dem Kommando-Byte über den Bus. Danach holen wir das Status-Byte vom DHT20 und wiederholen das so lange, bis der Chip durch Rücksetzen des Bits 7 Vollzug meldet.

Wenn jetzt auch noch das Bit 3 (**Calibrated**=0x08=0b00001000) gesetzt ist, liefert das Undieren des Staus-Bytes mit **Calibrated** den Wert 0b00001000, was wir in einen boolschen Wert umwandeln und als **True** zurückgeben.

Die Methode **readStatus**() liest ein Byte vom Bus, das StatusByte und gibt es zurück.

```
    def readStatus(self):
        return self.i2c.readfrom(DHT_HWADR,1)[0]
```

Für eine Messung muss der Triggerbefehl zum DHT20 gesendet werden, zusammen mit den Bytes 0x33 und 0x00, das macht **readRawData**(). Nach maximal 20ms sind die Daten bereit und wir können das StatusByte und die fünf Temperatur- und Feuchte-Bytes abholen. Für die weitere Verarbeitung wandeln wir das empfangene Bytes-Objekt in ein Bytearray um und weisen dieses dem Instanz-Attribut **data** zu.

```
    def readRawData(self):
        data=bytearray((0x33,0x00))
        self.writeRegs(cmdTrigger,data)
        sleep(0.02)
        self.data = bytearray(self.i2c.readfrom(DHT_HWADR, 6))
```

Die Methode **temperature**() berechnet jetzt die Temperatur in Grad Celsius. Dazu werden die Rohdaten vom DHT20 abgeholt. Die Zellen 3,4 und 5 von **data** enthalten den Temperaturrohwert. Genau genommen sind nur die unteren 4 Bits von **data**[3], das Low-Nibble, der Temperatur zuzurechnen. Die bilden aber die vier höchstwertigen Bits des Zwischenwerts. Weil 16 Bitpositionen darunterliegen, isoliere ich das Low-Nibble von **data**[3] durch Undieren mit 0x0F und schiebe die Bits 16

Stellen nach links. Die nächsten 8 Bits liefert **data**[4], ich schiebe sie um 8 Positionen nach links und oderiere das mit dem bisherigen Wert. Die untersten 8 Bits können dann mit **data**[5] durch Oderieren aufgefüllt werden.

Folgende Darstellung kann den Vorgang vielleicht besser vermitteln.
Nehmen wir an, **data**[3:6] hat die Form (0b????xxxx,0byyyyyyyy, 0bzzzzzzzz), dann passiert folgendes, wobei ?,x,y und z Bitpositionen darstellen.

```
    0b????xxxx
&   0b00001111
=   0b0000xxxx

                  0b0000xxxx << 16
    0bxxxx 00000000 00000000
                  0byyyyyyyy << 8
         0byyyyyyyy 00000000

    0bxxxx 00000000 00000000
|        0byyyyyyyy 00000000
|                  0bzzzzzzzz
=   0bxxxx yyyyyyyy zzzzzzzz
```

```
    @property
    def temperature(self):
        self.readRawData()
        self.temp = ((self.data[3] & 0xF) << 16) |\
                     (self.data[4] << 8) | \
                     self.data[5]
        self.temp = ((self.temp * 200.0) / (1 << 20)) - 50
        return self.temp
```

Das Ergebnis in **temp** wird jetzt mit 200 multipliziert, durch 2 hoch 20 laut Datenblatt dividiert und um 50 erleichtert. Die Celsius-Temperatur geben wir zurück.

Sie wundern sich über den Decorator **@property**? Diese Zeile macht es möglich, den Rückgabewert wie eine Referenz auf eine Variable zu handhaben. Statt durch einen Methodenaufruf wie

DHT20.temperature()

können wir jetzt schreiben

DHT20.temperature

und diesen Ausdruck so auch in Formeln verwenden. Mehr über diese Art von syntaktischem Zucker in MicroPython erfahren Sie hier.

Ähnlich arbeitet **humidity** (), nur sind hier die oberen vier Bits von **data**[3] das niederwertigste Nibble des Rohwerts. Ein Schieben um 4 Positionen nach rechts verfrachtet die Bits dorthin. **data**[1] und **data**[2] gesellen sich durch Linksschieben um 12 beziehungsweise um 4 Positionen dazu. Wieder entsteht eine 20-stellige

Binärzahl, die aber jetzt nur mit 100 zu multiplizieren und durch die Konstante 2^{20} zu dividieren ist, um den Wert der relativen Luftfeuchte zu erhalten.

```
    @property
    def humidity(self):
        self.readRawData()
        self.hum = ((self.data[1] << 12) | \
                            (self.data[2] << 4) | \
                            (self.data[3] >> 4))
        self.hum = (self.hum * 100 ) / (1<<20)
        return self.hum
```

Das Modul i2cslave

Es beginnt mit dem Import der Klasse Pin und einigen Funktionen.

```
from machine import Pin, SoftI2C
from time import sleep,sleep_us
from timeout import *
```

Mit der Klassen-Deklaration legen wir die Geräteadresse unseres I2C-Slaves fest, 0x63.

```
class I2CSLAVE:

    HWADR=const(0b1100011) # 0x63
```

Der Konstruktor nimmt die Nummern der Pins für SCL und SDA sowie die Taktfrequenz und die Hardwareadresse.

```
    def __init__(self,scl=18,sda=19,freq=100,hwadr=HWADR):
        self.clock=Pin(scl,Pin.IN)
        self.data=Pin(sda,Pin.IN)
        self.freq=freq
        self.HWADR=hwadr
        self.puls=int(1000000/freq/2) #us
        self.stop=False
        print("Slave started @ {} Hz".format(freq))
```

Wir erzeugen die Pin-Objekte, merken uns die Frequenz und die Hardwareadresse in entsprechenden Instanz-Attributen, um in jeder Methode der Klasse darauf zugreifen zu können. Die Dauer eines Taktimpulses in Mikrosekunden berechnen wir aus der Frequenz.

Die Methode **devAddress**() macht es uns möglich, die Geräteadresse nachträglich zu verändern oder abzurufen.

```
    def devAddress(self,adr=None):
        if adr is None:
            return self.HWADR
        else:
            self.HWADR=(adr & 0xff)
```

Ähnliches gilt für die Taktfrequenz.

```
    def frequency(self,freq=None):
        if freq is None:
            return self.freq
        else:
            self.freq=freq
            self.puls=int(1000000/freq/2)
```

setIdle() versetzt die Busleitungen in den hochohmigen Zustand, indem sie als Eingänge programmiert werden. Durch die beiden Pullup-Widerstände wird der Pegel auf 3,3V = HIGH gezogen.

```
    def setIdle(self):
        self.clock(Pin.IN)
        self.data(Pin.IN)
```

Die Signalpegel auf den Busleitungen werden über Schleifen abgetastet, die einen Pegelwechsel erkennen.

```
    def waitDataLow(self):
        while self.data.value()==1:
            pass

    def waitClockLow(self):
        while self.clock.value()==1:
            pass

    def waitClockHigh(self):
        while self.clock.value()==0:
            pass
```

Jede Übertragung wird mit einer Start-Condition eingeleitet. SCL und SDA sind zunächst beide HIGH. Zuerst geht SDA auf LOW, etwas später SCL. Dann folgt die Übertragung der Datenbits.

```
    def awaitStart(self):
        while self.clock.value()==1:
            if self.data.value()==0:
                while self.clock.value()==1:
                    pass
```

Den Abschluss der Übertragung bildet die Stop-Condition. Dazu muss die SDA-Leitung auf LOW sein, dann warten wie auf ein HIGH auf SCL. Wenn nach einer Taktperionendauer die SDA-Leitung auf HIGH gegangen ist, wurde eine Stop Condition erkannt, das melden wir mit dem Zustand des SDA-Eingangs zurück.

```
    def awaitStop(self):
        self.waitDataLow()
        self.waitClockHigh()
        sleep_us(self.puls*2)
        return self.data.value()==1
```

Um ein Byte einzulesen, setzen wir den Bytewert erst mal auf 0 und **stop** auf False. In der for-Schleife warten wir in acht Durchgängen auf eine steigende Flanke des Taktsignals. Das empfangene Bit oderieren wir zum bisherigen Bytewert nachdem wir diesen um eine Position nach links verschoben haben. Die neue Runde beginnt, wenn an SCL der Pegel auf LOW gefallen ist.

```
    def readByte(self):
        byte=0
        self.stop=False
        for i in range(8):
            self.waitClockHigh()
            byte = ((byte )<<1 ) | self.data.value()
            self.waitClockLow()
        return byte
```

writeByte() nimmt einen Bytewert. Wir warten auf eine fallende Taktflanke, sie ist das Zeichen dafür, ein Bit auf SDA zulegen. Natürlich muss der Pin erst als Ausgang geschaltet werden. Mit dem Wert in **mask** maskieren wir ein Bit nach dem anderen, beginnend mit dem MSB (Most Significant Bit). Das Undieren mit **mask** liefert einen Bytewert, welcher der Bitposition entspricht oder 0. Ersteres wird von MicroPython als True gewertet, die 0 als False. Dementsprechend wir die SDA-Leitung auf HIGH oder LOW gelegt. Jetzt muss der Master die Taktleitung zuerst auf HIGH und dann wieder auf LOW legen. Sind alle Bits übertragen, schalten wir die Datenleitung wieder als Eingang.

```
    def writeByte(self,byte):
        self.waitClockLow()
        self.data.init(Pin.OUT)
        mask=0x80
        for i in range(0,8):
            bit=byte & mask
            if bit:
                self.data.value(1)
            else:
                self.data.value(0)
            mask=mask >> 1
            self.waitClockHigh()
            self.waitClockLow()
        self.data.init(Pin.IN)
```

Zur Validierung des Datenbytes dient das Acknowledge Bit, das als neuntes Bit an die acht Datenbits angehängt wird. Ein LOW auf der Datenleitung signalisiert ein ACK ein HIGH ein NACK (Not Acknowledge). Dieses Bit wird nach demselben Schema übertragen beziehungsweise abgefragt wie die Datenbits.

```
    def sendAck(self,ack):
        self.waitClockLow()
        self.data.init(Pin.OUT,value=ack) # access data
        self.waitClockHigh()
        self.waitClockLow()
        self.data.init(Pin.IN)# release data

    def awaitAck(self):
        self.waitClockLow()
        self.waitClockHigh()
        ackBit=self.data.value()
        self.waitClockLow()
        return ackBit
```

Der I2C-Master

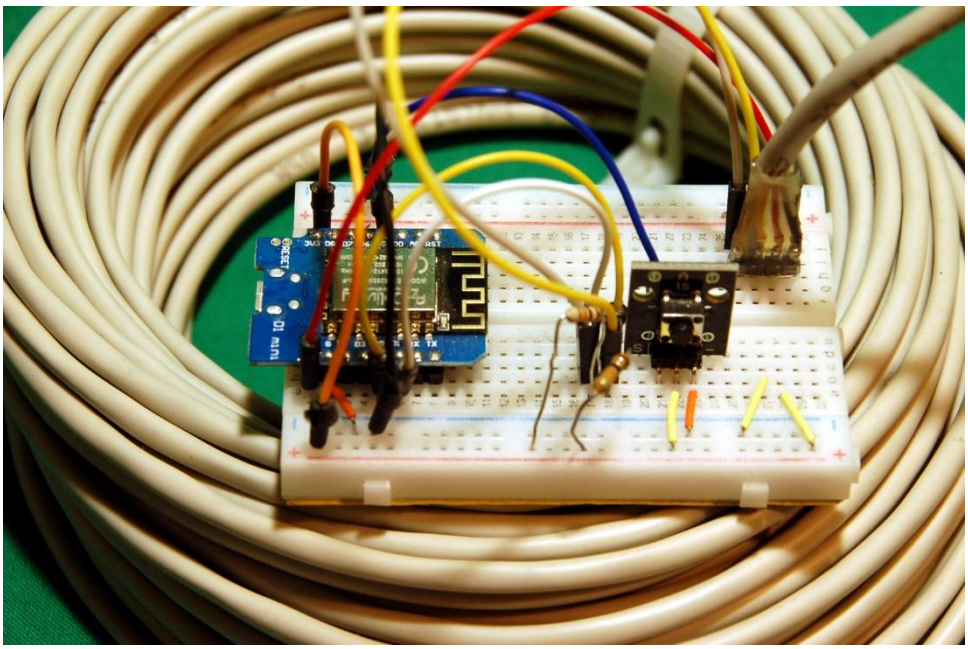

Abbildung 6: I2C-Master

Der Master bedient sich bei der Übertragung des hauseigenen MicroPython-I2C-Moduls. Beim Import ist das Modul **struct** mit der Methode **unpack** erwähnenswert. Damit entpacken wir das übertragene Bytes-Objekt und wandeln es zurück in einen Byte-, Integer- oder Fließkommawert. Näheres dazu später.

Ein I2C-Objekt wird an den Standard-Pins erzeugt, dessen Frequenz wir auf sichere 100 Hz festlegen. Die Geräteadresse unseres Slaves wird 0x63 sein.

```
import sys
from machine import Pin, SoftI2C
# from oled import OLED
from time import sleep
from timeout import *
from struct import unpack, pack

i2c=SoftI2C(Pin(5),Pin(4),freq=100,timeout=10000)
hwadr=0x63
```

Der Slave wird uns sechs Aktionen erlauben, Beleuchtung messen, Temperatur und rel. Luftfeuchte liefern, Reedkontakt abfragen und das Relais ein- und ausschalten.

```
readLight = const(0x01)
readTemp  = const(0x02)
readHum   = const(0x04)
readKontakt=const(0x08)
RelaisOn  = const(0x10)
RelaisOff = const(0x20)
```

Damit verbunden ist als Ergebnis ein Bytes-Objekt, das unterschiedlich viele Stellen aufweisen kann. Der Lichtwert ist eine Ganzzahl mit zwei Bytes, die Werte vom DHT20 sind Fließkommezahlen mit 4 Bytes und die restlichen Werte sind einzelne Bytewerte. Die Zuordnung regelt das Dictionary commands.

Je nach der Stellenanzahl wird für die Dekodierung ein Formatstring benötigt. "b" steht für ein Byte, "H" für eine vorzeichenlose Ganzzahl mit zwei Bytes, und "f" kennzeichnet eine Fließkommazahl mit vier Bytes. Die Verbindung stellt das Dictionary **p** her.

```
p={1:"b",
   2:"H",
   4:"f"
   }
```

Das Programm meldet seine Einsatzbereitschaft und geht in die Main-Loop, die Hauptschleife.

Die Eingabeschleife wartet auf einen Zahlenwert für die Auswahl des Auftrags. Der String wird in eine Zahl umgewandelt und der Variablen **wahl** zugewiesen.

```
    wahl=int(input("1, 2, 4, 8, 16, 32 -> "))
```

Zum Abfangen von Fehlern bauen wir den Rest in eine try-except-Struktur ein. Wir prüfen zunächst, ob die Eingabe in **commands** als Schlüssel vorkommt. Ist das der Fall dann setzen wir den Schlüssel und den zugehörigen Wert aus **commands** zu

einem Tupel zusammen. Das Kommandobyte senden wir an den Slave und warten 300ms, bis dieser mit dem Ergebnis aufwarten kann. Im zweiten Feld des Tupels steht die Anzahl der zu empfangenden Bytes. Die geben wir an **i2c.readfrom**() weiter und erhalten ein Bytes-Objekt dieser Länge. Aus **p** holen wir den entsprechenden Formatstring, mit dem **unpack**() das Bytes-Objekt in den Zahlenwert verwandelt. Den Wert lassen wir uns in REPL anzeigen.

Aufgetretene Fehler meldet uns der except-Block ohne das Programm abzubrechen.

```
    try:
        if wahl in commands.keys():
            command=(wahl,commands[wahl])
            i2c.writeto(hwadr,bytearray([command[0]]))
            sleep(0.3)
            code=(i2c.readfrom(hwadr,command[1]))
            s=p[command[1]]
            wert=unpack(s,code)[0]
            print(wert)
    except Exception as e:
        print(e)
```

Der I2C-Slave

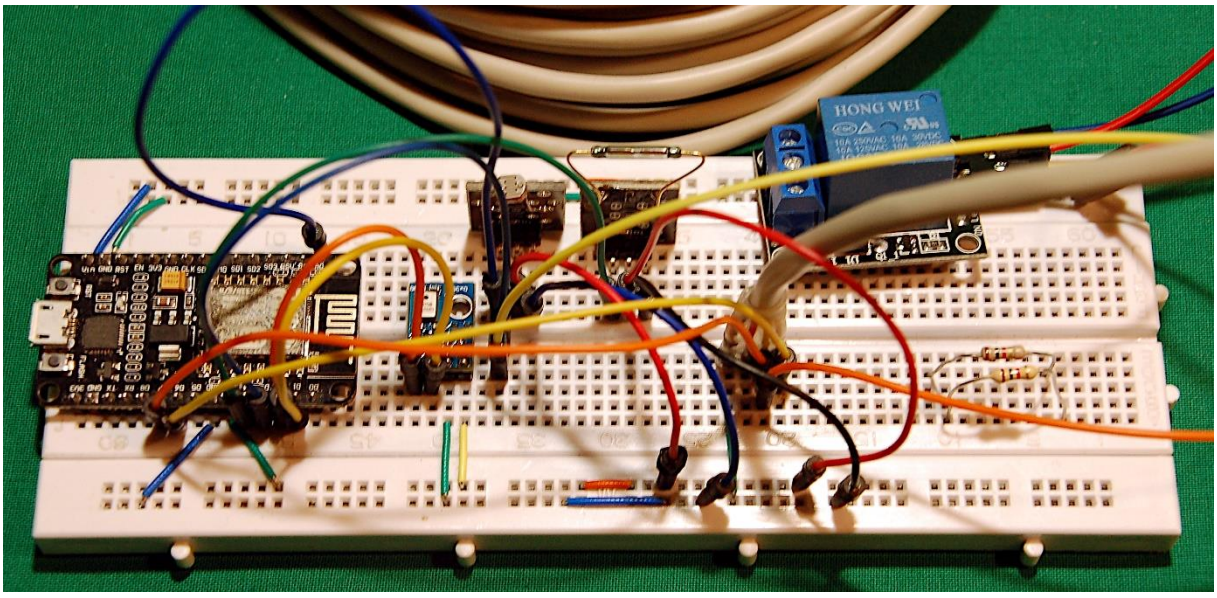

Abbildung 7: I2C-Slave

Der Slave braucht die Klassen **SoftI2C** für den DHT20, außerdem **Pin** und **ADC**, ferner die Klassen **DHT20** und natürlich **I2CSLAVE**. Zum Packen der Zahlenwerte dient **pack**.

```
from machine import SoftI2C, Pin, ADC
from dht20 import DHT20
from i2cslave import I2CSLAVE
from struct import pack
```

An den Standard-Pins entsteht ein MicroPython-I2C-Objekt für den Zugriff auf den DHT20.

```
i2c=SoftI2C(Pin(5),Pin(4),freq=100000)
```

Die Verbindung zum Master liefert der Konstruktor der Klasse I2CSLAVE an den Pins D6=GPIO12 und D7 = GPIO13.

```
hwadr=0x63
slave=I2CSLAVE(scl=12,sda=13,freq=100,hwadr=hwadr)
```

Wir deklarieren ein Pin-Objekt für den Relais-Ausgang, den wir auf LOW setzen.

```
relais=Pin(0,Pin.OUT,value=0)
relaisState=0
```

Mit dem hauseigenen I2C-Objekt instanziieren wir das DHT20-Objekt.

```
dht20=DHT20(i2c)
```

Den Türkontakt lesen wir über GPIO2 ein. Der LDR-Pegel wird an A0 abgeholt.

```
kontakt=Pin(2,Pin.IN)
ldr=ADC(0)
```

Kommandoregister und **val** werden deklariert sowie die Konstanten für die Kommandobytes.

```
cmdReg=None
val=None
# Kommandos
readLight = const(0x01)
readTemp  = const(0x02)
readHum   = const(0x04)
readKontakt=const(0x08)
relaisOn  = const(0x10)
relaisOff = const(0x20)
```

Einige Funktionen holen die Werte von den Sensoren und geben sie, dem Zahlenformat entsprechend, gepackt zurück. Hier sehen Sie den Unterschied zwischen einem Funktionsaufruf (**ldr.read**(), **kontakt.value**()) und dem Abfragen einer Property (**dht20.Temperature**, **dht20.humidity**).

```
def getLight():
    return pack("H",(1024-ldr.read())) # Normieren auf ein
Byte

def getTemp():
    return pack("f",dht20.temperature)

def getHum():
    return pack("f",dht20.humidity)

def getTuerKontakt():
    return pack("b",kontakt.value())
```

Die Funktion **relaisSwitch**() nimmt den Schaltzustand 0 oder 1, der an den Ausgang weitergegeben wird. Die Abfrage des Ausgangspuffers liefert zur Kontrolle den aktuellen Zustand.

```
def relaisSwitch(val):
    relais.value(val)
    return pack("b",relais.value())
```

Die Hauptschleife bildet unsere MicroPython-I2C-Slave-Schnittstelle ab. Die beiden Busleitungen gehen auf Eingang, und wir warten auf eine Start-Condition.

```
while 1:
    slave.setIdle()
    slave.awaitStart()
```

Wurde diese erkannt, dann lesen wir als erstes Byte die Hardware-Adresse ein und senden ACK, also SDA auf LOW.

```
    hwa=slave.readByte()
    slave.sendAck(0)
```

Aus dem Byte isolieren wir das R/-W-Bit und stellen die 7-Bit-Geräteadresse wieder her. Zur Kontrolle werden die Werte in REPL ausgegeben.

```
    rw=hwa & 0x01 # Richtungsbit isolieren
    hwa=hwa>>1 # 7-Bitadresse bilden
    print("HWADR:",hex(hwa),rw)
```

Wir sind gemeint, wenn die empfangene Geräteadresse mit der oben definierten übereinstimmt.

```
    if hwa==slave.HWADR:
```

Jetzt geht es darum ob ein Kommando empfangen wurde oder ob Daten gesendet werden müssen. Einen Befehl erkennen wir daran, dass **rw** = 0 ist.

```
        if rw==0: # befehl empfangen, decodieren, ausfuehren
```

In diesem Fall muss das Kommando-Byte gelesen werden, dann senden wir ACK.

```
            cmd=slave.readByte() # Kommandobyte lesen
            slave.sendAck(0)
```

Wird eines der Kommandobytes erkannt, lösen wir die entsprechende Aktion aus. den Rückgabewert verstecken wir vorerst in der Variablen **val**. Wird kein Kommando erkannt, passiert gar nix – **pass**.

```
            if cmd==readLight:
                val=getLight()
            elif cmd==readTemp:
                val=getTemp()
            elif cmd==readHum:
                val=getHum()
            elif cmd==readKontakt:
                val=getTuerKontakt()
            elif cmd==relaisOn:
                val=relaisSwitch(1)
            elif cmd==relaisOff:
                val=relaisSwitch(0)
            else:
                pass
```

War **rw** = 1, dann müssen die Bytes in **val** gesendet werden. Danach warten wir jeweils auf ein ACK vom Master.

```
        else: # Daten senden
            for i in range(len(val)):
                slave.writeByte(val[i])
                ack=slave.awaitAck()
```

Unabhängig vom Modus warten wir nach der Übertragung auf eine Stop-Condition vom Master und setzen dann die Leitungen auf Eingang.

Die Klasse I2CSLAVE umfasst nicht alle Features der MicroPython-eigenen Schnittstelle. So fehlt zum Beispiel die Angabe eines Timeouts, wenn längere Zeit vergeblich auf eine Stop-Condition gewartet wird. In diesem Fall müssten nach Überschreiten der Wartezeit die Leitungen freigegeben werden.

Zunächst ist es auch nicht vorgesehen, mit dem Kommando weitere Datenbytes zu empfangen, wie es zum Beispiel der DHT20 mit dem Calibrier- oder Trigger-Befehl macht. Allerdings wären dafür weitere Steuerstrukturen nötig, die zu einer

zusätzlichen Verlangsamung des Transfers führen würden. Vielleicht bearbeite ich dieses Feld mal irgendwann. Im Moment freue ich mich über die gut funktionierende Datenverbindung zu meinem I2C-Sklaven.