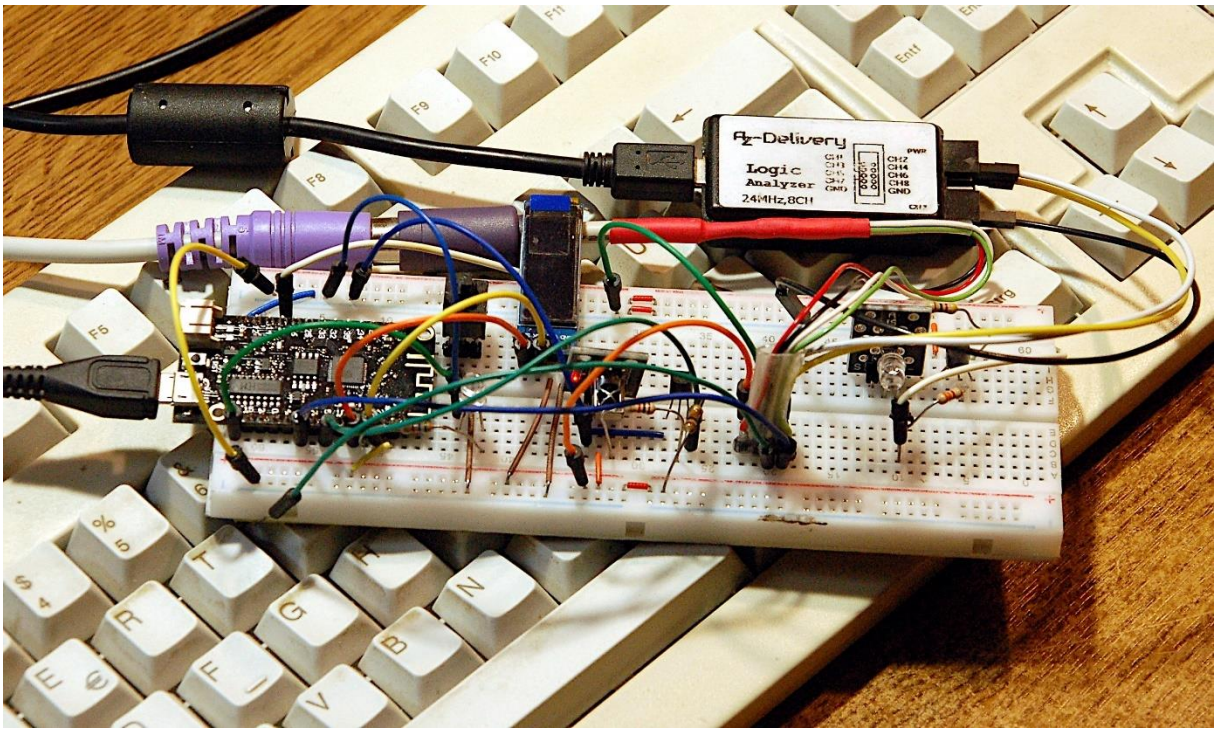

ESP32 mit Tastatur

Diesen Beitrag gibt es auch als PDF-Dokument.

Nachdem der ESP32 inzwischen IR-Codes von einer RC (remote Control) übernehmen und selbst solche senden kann, will ich ihm heute beibringen, die Tastencodes einer PC-Tastatur einzulesen und in ASCII-Zeichen umzusetzen. Damit sind wir in der Lage, die Bezeichnungen der RC-Tasten bei der Erfassung ohne PC zu erledigen. Außerdem gibt es sicher noch diverse weitere Einsatzmöglichkeiten für die Tastatur bei zukünftigen Projekten.

Beim Erforschen der Arbeitsweise einer PC-Tastatur war der Logic Analyzer erneut ein hilfreiches Werkzeug. Ohne das kleine Ding oder ein DSO (Digitales Speicher Oszilloskop) kommt man kaum auf die Eigenheiten der Tastencodes.

Ich lade Sie ein, die Schritte zum fertigen Modul **ps2.py** mit mir zu gehen, in der dritten Folge zum aktuellen Themenkreis IR-Transfer.

MicroPython auf dem ESP32 und ESP8266

heute

Der ESP32 und die PS/2-Tastatur

Zur bisherigen Schaltung kommen nur ein Adapter und eine PS/2-Tastatur dazu.

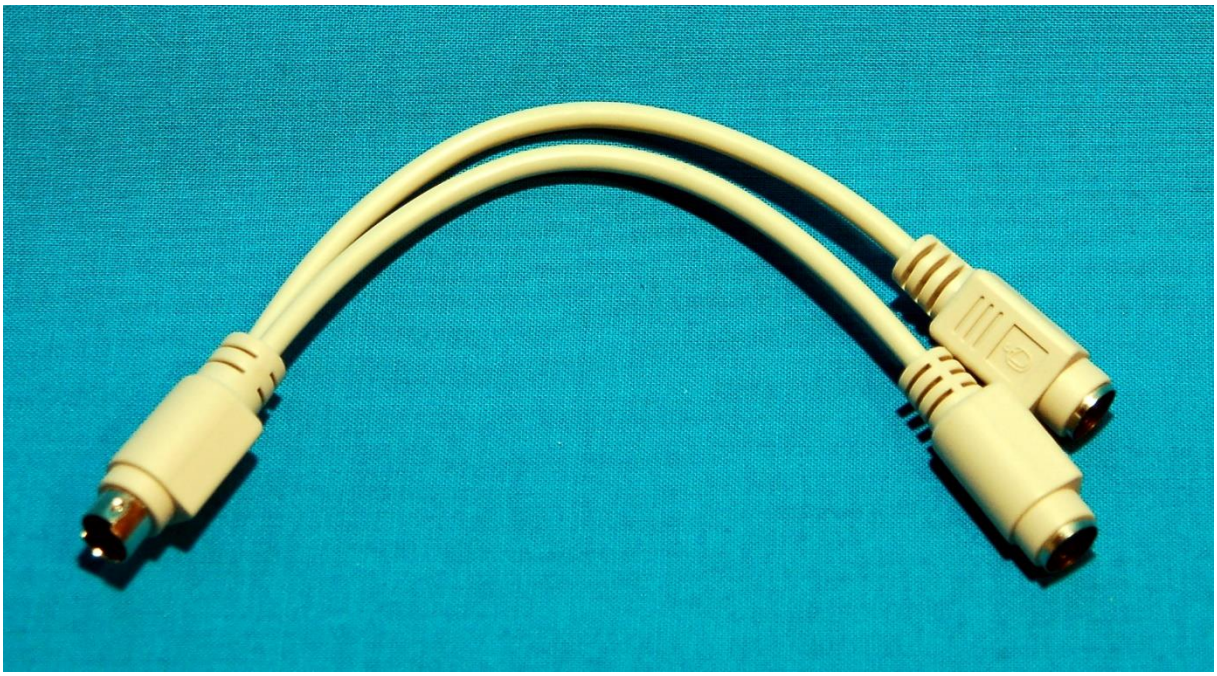

Abbildung 1: Einfacher Adapter

Der erste Versuch mit so einem Teil den Adapter herzustellen, schlug leider fehl. Die Kabel, rot, schwarz, weiß und grün, waren mit anderen Buchsen verbunden als normal üblich. Aber selbst nach dem Vermessen und in neuer Zuordnung kam kein Signal am Ende an. Also habe ich einen anderen Umsetzer probiert, mit dem ich dann auch sofort Erfolg hatte.

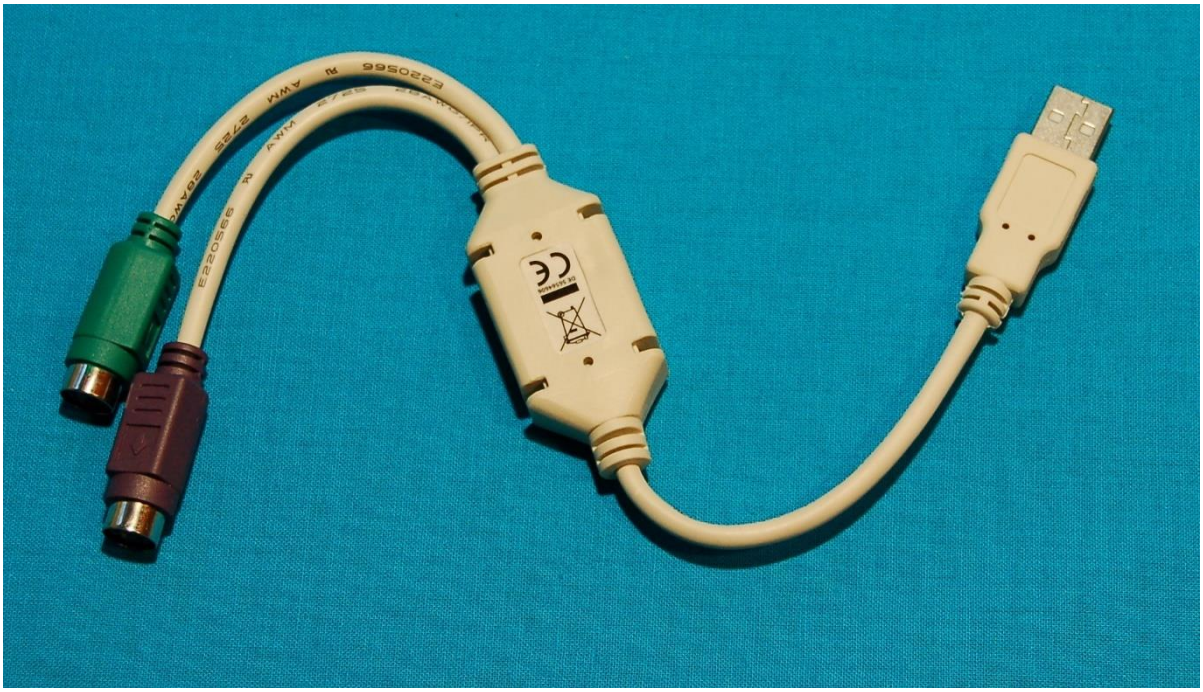

Abbildung 2: USB-Adapter

Den Adapter habe ich aus diesem Umsetzer von PS/2 auf USB hergestellt, indem ich einfach das Kabel für den Tastaturanschluss gekappt und mit vier dünnen Kabeln verlängert habe.

Ans andere Ende wurde eine vierpolige Stiftleiste gelötet, damit man das Ding auf ein Breadboard stecken kann.

Abbildung 3: PS2 Tastaturanschluss fürs Breadboard

Natürlich könnte man auch den Stecker der Tastatur abschneiden und die Steckleiste direkt an die Kabelchen löten, wenn man die Tastatur zu sonst nichts mehr braucht.

Übrigens geht das auch mit einer USB-Tastatur, man muss dann nur einen Adapter von USB auf PS/2 zwischenstecken.

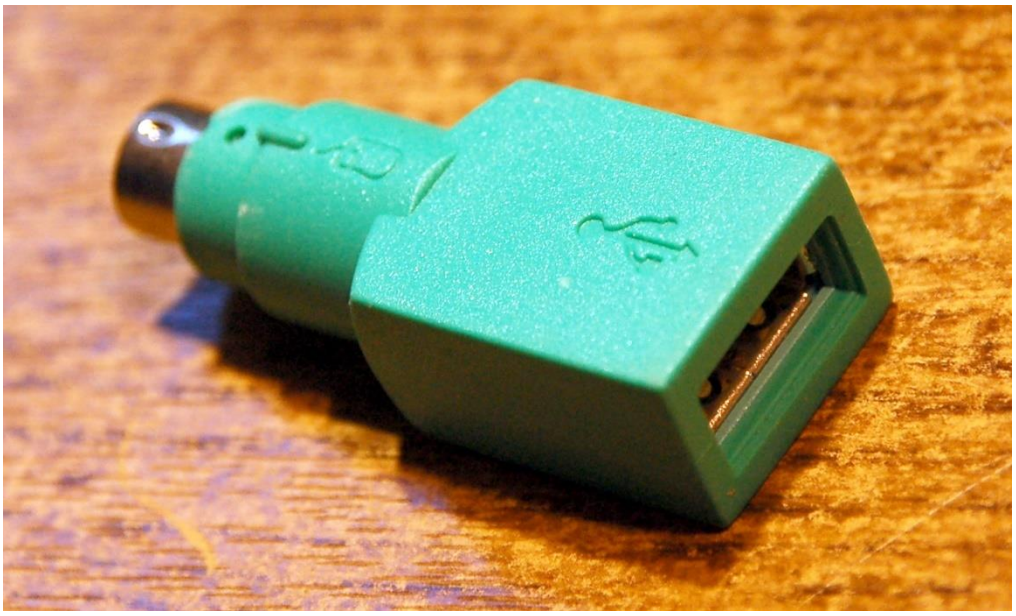

Abbildung 4: USB-PS2-Adapter

Die Schaltung

Die Schaltung aus dem zweiten Teil des Projekts habe ich direkt übernommen. Ich musste letzten Endes nur die Tastatur mit Spannung versorgen und die Datenleitung mit dem ESP32 verbinden. Die Tastatur ist sehr genügsam und gibt sich mit 3,3V zufrieden.

Mehr Aufwand bereitete der erste programmtechnische Ansatz. Die Daten werden beim PC synchron zu einem Taktsignal, das in der Tastatur erzeugt wird, auf einer

separaten Datenleitung übertragen. Auf das Protokoll komme ich später detailliert zurück. Also hatte ich vor, das auf dem ESP32 genauso zu machen. Weil das nach einem Tag immer noch nicht funktionieren wollte, habe ich einen anderen Weg beschritten. Jetzt reicht eine Datenleitung, die Daten werden asynchron übertragen und an GPIO18 vom ESP32 entgegengenommen. Diese Lösung ist verträglich mit der bisherigen Peripherie. SCL bleibt an 22, SDA an 23, der IR-Empfänger an 5 und der Sender an 27.

Aufgebaut ist alles immer noch auf dem Breadboard mit 62 Kontaktreihen. Das heißt, dass dort auch immer noch ein ESP32 Lolin schuftet, der schmalbrüstiger ist als seine größeren Verwandten. Somit bleibt an den beiden Seiten je eine Kontaktreihe für Jumperkabel frei. Für einen ESP32 Dev Kit C V3 oder ESP32 Dev Kit C V4 braucht man zwei parallel gesteckte Breadboards.

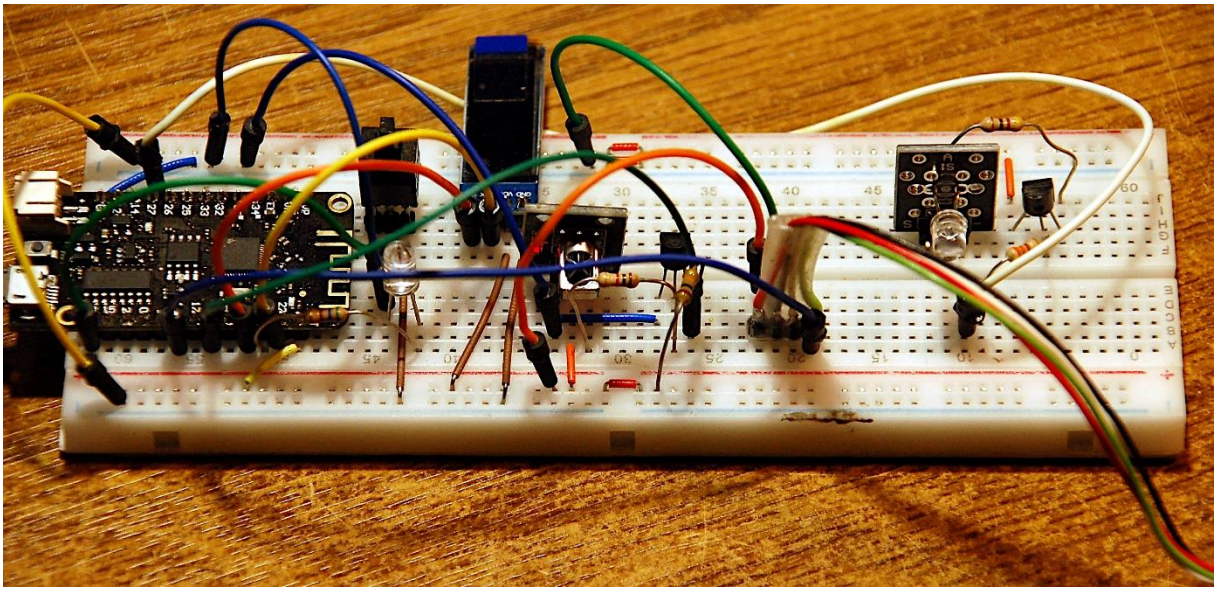

Abbildung 5: Aufbau mit PS2-Tastatur-Anschluss

Das Schaltbild zeigt die Verdrahtung etwas genauer.

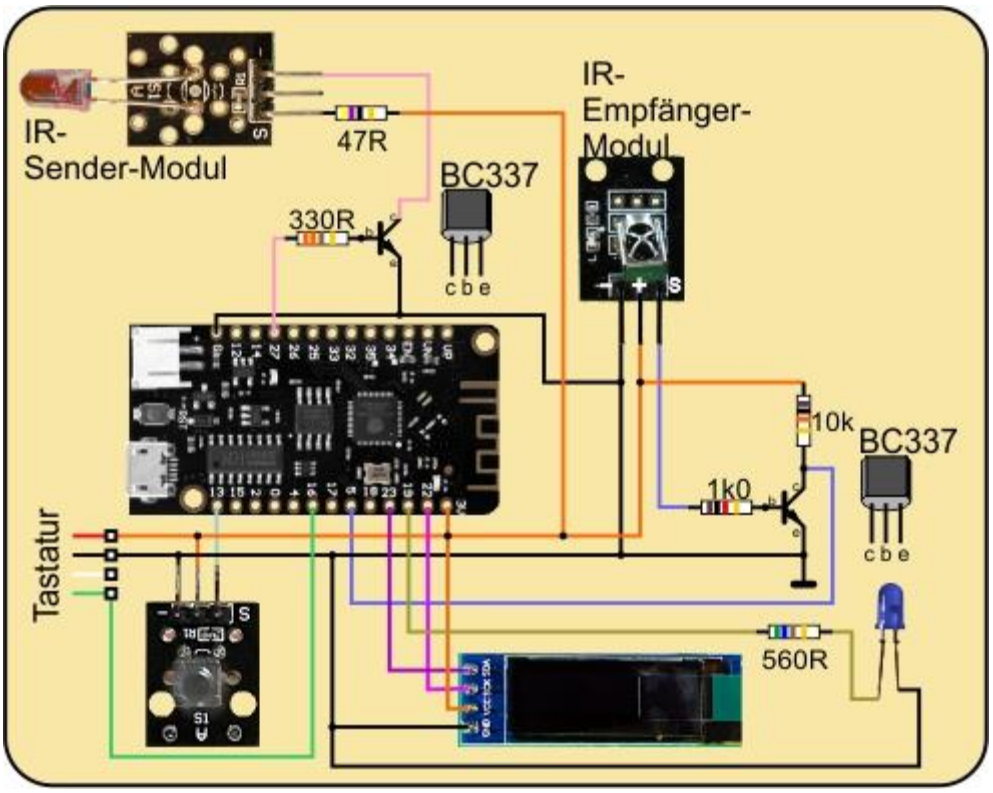

Abbildung 6: Schaltung mit Tastatur-Anschluss

Damit sind wir auch schon bei der Hardware-Liste angekommen.

Hardware

1	ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
1	KY-022 Set IR Empfänger
1	KY-005 IR Infrarot Sender Transceiver Modul
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord kompatibel mit Arduino und Raspberry Pi - 1x Set
1	KY-004 Taster Modul
diverse	Jumper Wire Kabel 3 x 40 STK
2	NPN-Transistor BC337 oder ähnlich
1	Widerstand 1,0 kΩ
1	Widerstand 10 kΩ
1	Widerstand 330 Ω
1	Widerstand 47Ω
1	Widerstand 560Ω
1	LED (Farbe nach Belieben)
1	Adapter PS/2 nach USB oder PS/2-Buchse
1	Logic Analyzer
1	PS/2 - Tastatur

Die PS/2-Buchse habe ich mit ca. 5cm Kabel vom Adapter abgeschnitten und 2cm abisoliert, um die Verlängerungsleitungen anzulöten. Die wurden dann mit Schrumpfschlauchstücken isoliert.

Das Anschlussschema zeigt Abbildung 7.

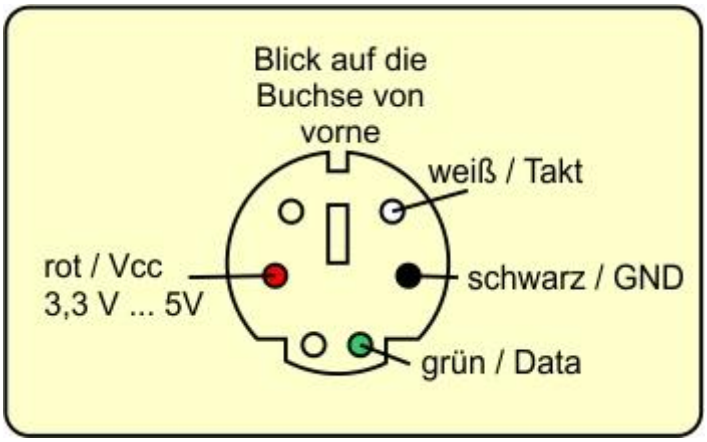

Abbildung 7: Belegung der Tastaturbuchse

Was sendet die Tastatur?

Ich hatte schon erwähnt, dass die Daten von der Tastatur über eine Takt- und eine Datenleitung synchron übertragen werden. Um dahinterzukommen, nach welchem Protokoll der Transfer funktioniert, habe ich mir die Signale zuerst einmal mit dem DSO angeschaut. Das hat mir verraten, dass der Takt mit 12kHz (12278Hz) läuft, was nicht gerade eine übliche Baudrate ist.

Als Nächstes musste ich erfahren, wann die Datenbits gesampelt werden müssen. Für diesen Zweck bin ich auf den Logic Analyzer umgestiegen, weil ich damit auch längere Sequenzen in guter Auflösung aufzeichnen kann.

Die Basiseinstellungen des kostenlosen Programms Logic 2 von Saleae können Sie der Abbildung 8 entnehmen.

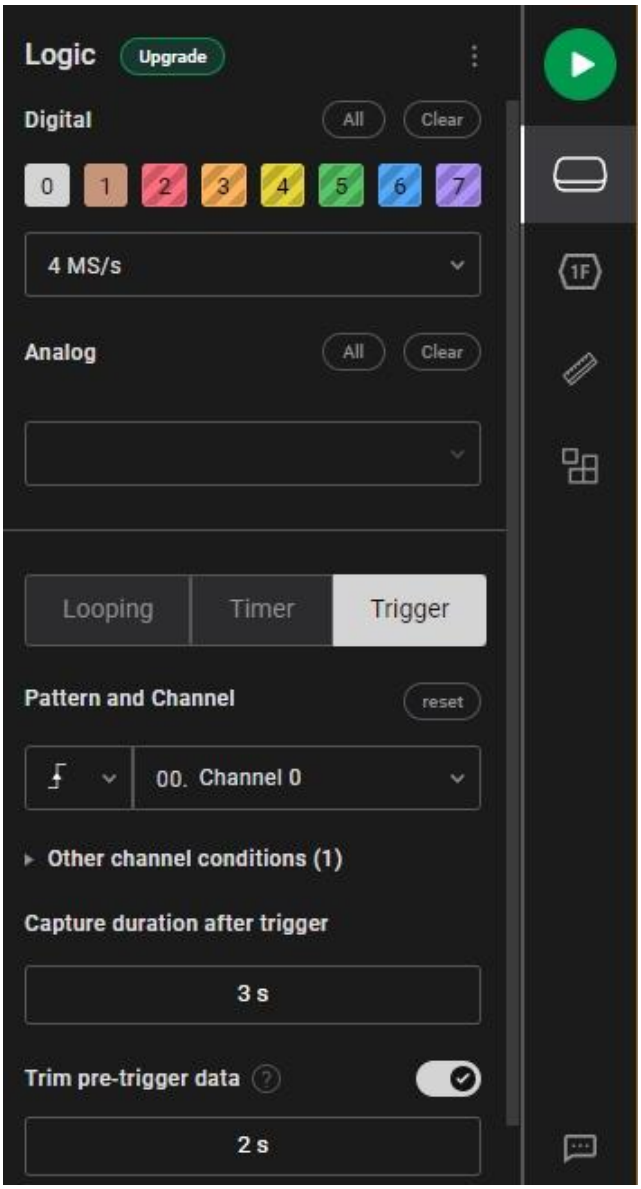

Abbildung 8: Basiseinstellungen des Analyzers

Ich habe mich für die Auswahl eines seriellen Protokolls entschieden. Die Auswahl des Analyzers erscheint, wenn man auf das "+"-Zeichen klickt.

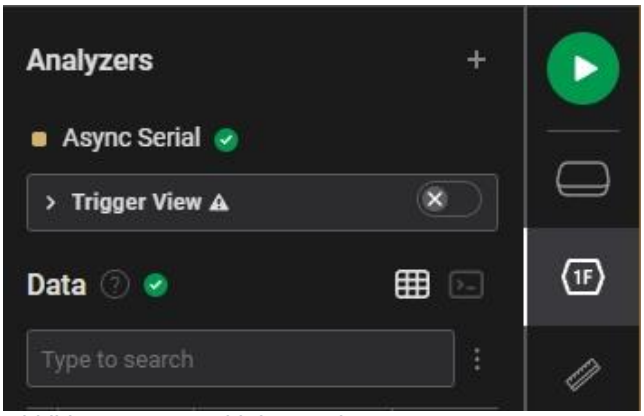

Abbildung 9: Auswahl des Analyzers

Das serielle Protokoll bietet verschiedene Einstellmöglichkeiten wie Anzahl der Bits, Paritätsbit sowie Anzahl der Stop-Bits und so weiter. Mit dem ersten Probe-Scan konnte ich feststellen, dass es acht Bits pro Byte sein müssten und dass das neunte Bit ein Paritätsbit sein müsste, gefolgt von einem Stop-Bit. Also Einstellungen aufrufen, um diese Angaben einzutragen.

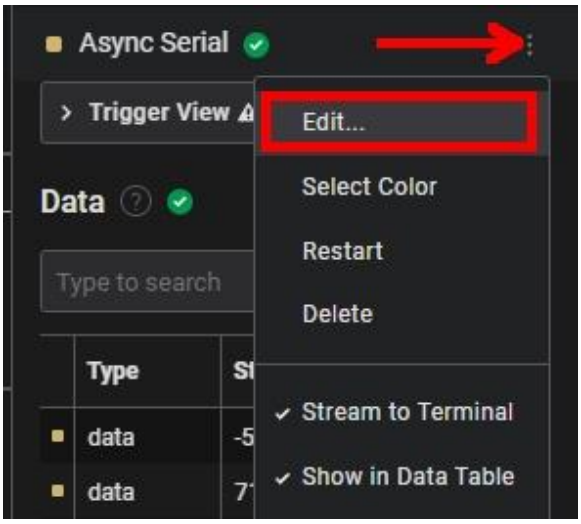

Abbildung 10: Einstellungen öffnen

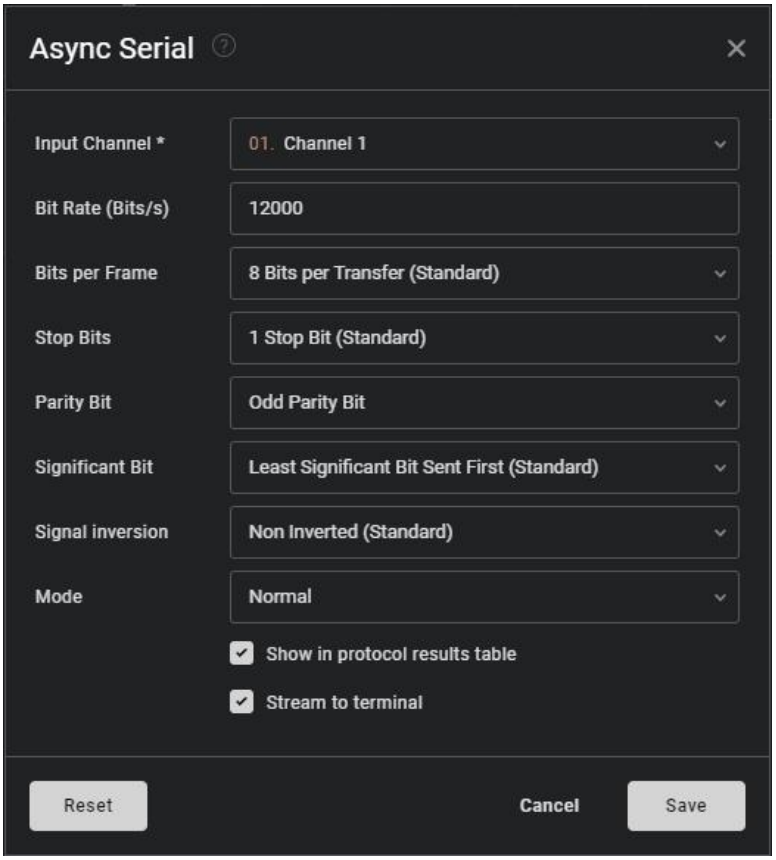

Abbildung 11: Einstellungen für den Analyzer

Mit **Save** das Ganze abspeichern. Die Tastatur ist am ESP32 angeschlossen und mit Spannung versorgt. Ich starte den Logic Analyzer mit der Taste **R** (PC) und tippe auf der PS/2-Tastatur auf **a**. Nach drei Sekunden Messzeit, rastet der Analyzer

automatisch beim ersten gesampelten Byte ein. Die braunen Punkte kennzeichnen die Bit-Positionen und am Start-Bit erkennt man, dass das Datenbit jeweils bei fallender Flanke des Takts gesampelt wird. Der Frame beginnt mit dem LSb (Least Significant Bit = niederwertigstes Bit).

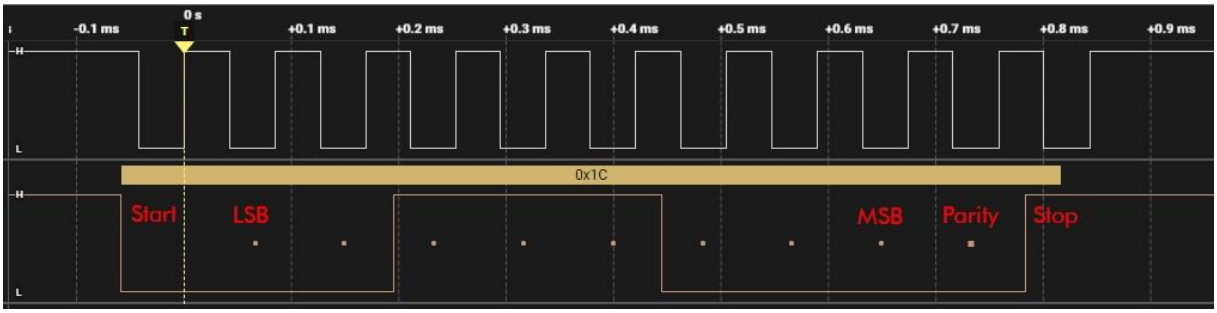

Abbildung 12: Sampeln bei fallender Flanke

Zoomen wir mit dem Mausrad heraus, dann erkennen wir, dass dem ersten Burst (Pulsfolge) nach ca. 70ms zwei weitere folgen. Betrachten wir sie näher.

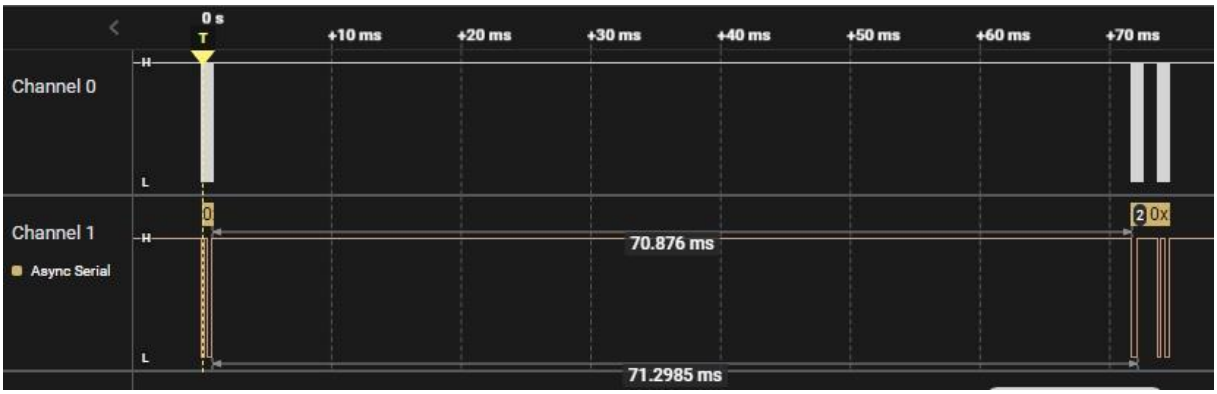

Abbildung 13: Ein Paket besteht aus drei Bursts

Abbildung 14: Das zweite und dritte Byte

Das zweite Byte hat den Wert 0xF0 und nach etwa 1ms wird das erste Byte wiederholt. So verhält es sich für alle Buchstaben, Ziffern und Funktions-Tasten. Bleibt eine Taste dauerhaft gedrückt, dann feuert die Tastatur deren Code nach einer Pause von 530ms im Abstand von ca.100ms erneut. Den Abschluss bilden wieder 0xF0 und der Tastencode.

Abbildung 15: Code-Wiederholung bei Halten der Taste

Für Großbuchstaben startet die Sequenz mit dem Code für die Shift-Taste, 0x12. Nach einer Pause von ca. 270ms folgt der normale Tastencode des Zeichens. Dann kommen 0xF0 und der Zeichencode. Den Abschluss bilden 0xF0 und 0x12. Für ein "a" finden wir 0x1C, 0xF0, 0x1C, für "A" 0x12, 0x1C, 0xF0, 0x1C, 0xF0, 0x12. Es sieht wohl so aus, als stünde 0xF0 für das Loslassen der Taste.

Einige Tasten der Haupt Tastatur und der Zehner- und Steuerblock feuern ein 5-er-Paket, das für **Cursor links** wie folgt aussieht.

0xE0, 0x6B, 0xE0, 0xF0, 0x6B

Ganz verrückte Sequenzen von 10 Bytes und mehr haben Exoten wie **Druck**, **Pause** und **Rollen**. Diese Tasten habe ich im folgenden Programm nicht berücksichtigt.

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft
SALEAE – Logic-Analyzer-Software (64 Bit) für Windows 8, 10, 11

Verwendete Firmware für einen ESP32:

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

Verwendete Firmware für einen ESP8266:

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

Die MicroPython-Programme zum Projekt:

irsend.py Treiber-Modul
timeout.py Nichtblockierende Softwaretimer
ps2.py Tastatur-Treiber

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 05.02.2022) 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.

Das Programm

Diese Erkenntnisse sollten für das Erstellen eines Programms genügen. Dachte ich und machte mich mit Feuereifer ans Werk. Aber die Hochstimmung verkroch sich nach und nach ins Mauseloch. Weder mit Interrupts noch mit Polling-Methoden kam der ESP32 ans Entschlüsseln der Tastencodes heran. MicroPython an sich ist wohl einfach zu langsam.

Aber Moment mal – das Signal von der Tastatur setzt ja auf einem seriellen Protokoll auf, vergleichbar dem der RS232, wenn auch mit einer exotischen Baudrate. Und der ESP32 besitzt mit UART2 eine freie serielle Schnittstelle. Also schnell mal ein UART-Objekt erzeugen und einen Test fahren. Die Datenleitung kommt an GPIO16, das ist RDX2, wie der folgende Plot zeigt.

>>> from machine import UART
>>> uart = UART(2, baudrate=12000,
bits=8,
parity=1,
stop=1)
>>> print(uart)
*UART(2, baudrate=12000, bits=8, parity=1, stop=1, tx=17, **rx=16**, rts=-1, cts=-1, txbuf=256, rxbuf=256, timeout=0, timeout_char=0)*

Taste "a" auf der PS/2-Tastatur drücken und Code einlesen. Puhuuu!
>>> uart.read()
b'\x1c\xf0\x1c'

Puhuuu! – Funktioniert!
Auf das Abfragen der Takt-Leitung kann ich jetzt verzichten.

Aus dem ersten kleinen Test-Prögrämmchen ist dann sehr schnell ein ausgewachsenes Modul mit der Klasse PS2 geworden, das wir uns jetzt genauer anschauen.

Die Klasse PS2

Wir benötigen einige Zutaten. Als Kernstück kommt die Klasse **UART** mit dem Import aus dem Modul **machine**. Es fällt viel Datenmüll an, den wir mit der Funktion **collect()** einsammeln werden. Für kurze Pausen holen wir **sleep**. Einen Softwaretimer liefert das Modul **timeout**. Alle importierten Bezeichner werden durch die Schreibweise mit **from** in den globalen Namensraum (Scope) eingefügt.

```
from machine import UART
from gc import collect
from time import sleep
from timeout import *
```

Es folgt die Klassendeklaration mit dem Konstruktor, der Methode __**init**__().

```
class PS2():

    def __init__(self,debug=False):
        self.uart = UART(2, baudrate=12000,
                         bits=8,
                         parity=1,
                         stop=1)
        print(self.uart)
        self.codeTable = {}
        self.debug=debug
        self.flushBuffer()
```

Der einzige Parameter debug ist optional, er regelt die Ausgabe von Debugmeldungen während der Laufzeit, wenn er auf True gesetzt wird. Wird kein Parameter beim Aufruf des Konstruktors übergeben, dann erhält **debug** den Defaultwert False.

UART0 ist durch REPL belegt, UART1 hat seine Leitungen im GPIO-Bereich, der für den SPI-Transfer zum Flash-EEPROM gebraucht wird. Da sowieso also nur UART2 in Frage kommt, instanziieren wir das UART-Objekt in der Klasse selbst. Das läuft für den Benutzer transparent, das heißt, er bekommt davon gar nichts mit. Die Parität bei der Übertragung wird mit **parity=1** auf **odd** eingestellt. Das heißt, dass die Anzahl von 1-Bits zusammen mit dem Paritätsbit ungerade sein muss. Das Paritätsbit wird also gesetzt, wenn im Datenbyte eine gerade Anzahl 1-Bits vorkommt. Vergleichen Sie hierzu Abbildung 11. In 0x1c sind drei 1-Bits, daher ist das Partätsbit nicht gesetzt.

Alle Einzelheiten über die Schnittstelle erfahren wir durch **print(self.uart)**. Für den Eingang **rx** wird GPIO16 bestätigt.

Die Übersetzung von Tasten-Code in ASCII-Code bekommen wir später durch das Dictionary **.codeTable**. Dazu muss die Tabelle noch gefüllt werden. Das erledigen wir zum Schluss.

Den Inhalt des Parameters **debug** übernehmen wir in die Instanz-Variable **.debug**, damit er in allen Methoden zur Verfügung steht. Für einen sauberen Start löschen wir den Empfangspuffer der Schnittstelle.

Die Methode **keysPresent**() meldet die Anzahl Zeichen im Empfangspuffer. Das flüstert uns die Methode **.uart.any**().

```
    def keysPresent(self):
        return self.uart.any()
```

Die Rohwerte von der Tastatur holt **readRaw**(). Die Liste **keys** wird die Bytes aufnehmen. Wir schauen nach wie viele Bytes im Puffer anstehen. Die nächst Zeile gibt diese Anzahl aus, wenn **debug** auf True steht. Den Trick, der hinter dieser Zeile

steckt, habe in der ersten Folge dieser Reihe genau erklärt. Natürlich gibt es nur etwas abzuholen, wenn **n** größer als Null ist.

```
    def readRaw(self):
        keys=[]
        n=self.keysPresent()
        self.debug and print(n)
        if n > 0:
            data = bytearray(n)
            self.uart.readinto(data, n)
            self.debug and print(data)
            for i in range(n):
                code=data[i]
                self.debug and print(i,hex(code))
                keys.append(hex(code))
            collect()
            self.debug and print(keys)
            return keys
```

Die Routine .**readinto**() erfordert die Angabe eines Objekts, welches das Bufferprotokoll unterstützt, weswegen wir ein Bytearray mit n Elementen erzeugen. Die **for**-Schleife schaufelt die Bytes nach code und hängt den Hex-String an die Liste **keys** dran. Wir sammeln den Datenmüll und geben die Liste zurück.

Die eben deklarierte Methode wird von **showCodes**() genutzt, um die Tastencodes fortlaufend anzuzeigen. Wir setzen n auf 0 und betreten die while-Schleife.

```
    def showCodes(self):
        n=0
        while 1:
            while n == 0:
                n=self.keysPresent()
            sleep(0.2)
            codes=self.readRaw()
            print(codes)
            n=0
```

Die zweite while-Schleife wird erst verlassen, wenn zumindest ein Byte angekommen ist. In Abbildung 11 haben wir schon gesehen, dass das zweite Byte etwas auf sich warten lässt, wenigstens um die 70 bis 80 ms, zusammen mit der Shift-Taste sogar 170ms. Deshalb müssen wir eine kurze Pause von 200ms einlegen, bis alle Bytes im Empfangspuffer eingetrudelt sind. Dann holen wir die Codes ab, geben sie aus und setzen n wieder auf 0 für die nächste Runde.

readKey() behandelt die drei oben beschriebenen Fälle und gibt den Tastencode zurück. Wir bereiten uns auf den Empfang von mindestens drei Bytes vor. Ist mindestens ein Byte im Speicher, dann kommen sicher noch mehr und wir können ans Decodieren gehen. Wir warten aber noch 100ms bis wir wirklich anfangen. Dann holen wir schon mal 3 Bytes ab. Ist das erste Byte 0xE0, dann besteht die Nachricht insgesamt aus fünf Bytes. Aber im zweiten Arrayelement steht bereits der

Tastencode. Dennoch müssen wir noch zwei Bytes abholen, um den Empfangspuffer zu leeren und die Integrität des Tastencodes zu überprüfen. Der ist OK, wenn das zweite der zuletzt abgeholten Bytes mit dem Wert in Code übereinstimmt.

```
    def readKey(self):
        data = bytearray(3)
        n=self.keysPresent()
        self.debug and print(n)
        if n>0:
            sleep(0.1)
            self.uart.readinto(data, 3)
            self.debug and print(data)
            if data[0] == 0xE0: #es folgen code 0xE0 0xF0 code
                code=data[1]
                self.uart.readinto(data, 2)
                if data[1] == code:
                    return code
```

Ist das erste empfangene Byte vom Wert 0x12, dann folgen noch 5 Bytes, von denen wir bereits drei haben. Das zweite Byte ist der wieder Tastencode, den wir uns merken. Von den drei weiteren Bytes sollte das erste mit dem Wert in code übereinstimmen. Für die Codetabelle kennzeichnen wir Großbuchstaben und Sonderzeichen wie "!", "§" etc. durch Voranstellen von 12, was durch oderieren mit 0x1200 erreicht wird.

```
            elif data[0] == 0x12: # code 0xF0 code 0xF0 0x12
                code=data[1]
                self.uart.readinto(data, 3)
                if data[0] == code:
                    return 0x1200 | code
```

Drei Bytes sind angekommen, wenn das erste und dritte Byte wertgleich sind und das zweite Byte 0xF0 ist. Bei sonstigen Bytefolgen wird 0x00 zurückgegeben.

```
            elif data[0] == data[2] and data[1] == 0xF0:
                self.debug and print(hex(data[0]))
                return data[0]
            else:
                return 0x00
```

Die Methode **awaitKey**() nutzt die eben definierte Methode. Der Parameter **delay** bestimmt, wie lange auf die Taste gewartet werden soll. Mit **delay** = 0 wird bis zum Sankt-Nimmerleins-Tag gewartet. Mit **TimeOutMs**() erzeugen wir einen nichtblockierenden Softwaretimer. Dahinter steckt eine sogenannte Closure. Mehr über diese Objekte erfahren Sie in Closures und Decorators.

```
    def awaitKey(self,delay=200):
        timedout=TimeOutMs(delay)
        n=0
        code=0x00
        while n==0 and not timedout():
            n=self.keysPresent()
        if not timedout():
            sleep(0.1)
            code=self.readKey()
            self.debug and print(code)
            n=0
        return code
```

Wir setzen **n** auf 0 und belegen auch **code** schon mal mit 0 vor. Die while Schleife läuft, bis wenigstens ein Byte angekommen ist und der Timer noch nicht abgelaufen ist. Mit **sleep**() wäre eine solche Konstruktion nicht möglich. Wurde die Schleife verlassen, bevor der Timer abgelaufen ist, warten wir wieder ein wenig, holen uns dann den Tastencode und geben ihn zurück. Im Falle eines Timeouts ist das 0x00.

Die Methode **toAscii**() nimmt den Tastencode und versucht ihn in ASCII-Code umzuwandeln, indem sie in der Codetabelle nach dem Schlüssel des Tastencodes sucht. Eine Exception wird geworfen, wenn der Code nicht existiert. Wir müssen also mit try und except arbeiten, damit das Programm nicht abstürzt. Im Fehlerfall wird der leere String zurückgegeben, sonst das ASCII-Zeichen.

```
    def toAscii(self, sc):
        try:
            c=self.codeTable[sc]
        except:
            c=""
        return c
```

Mit **scan**() bauen wir die Codetabelle **.codeTable** auf. Auf der PC-Tastatur warten wir auf die Eingabe der Tastenbezeichnung. Der Vorgang wird abgebrochen, wenn "quit" eingegeben wird. Die Codetabelle wird ausgegeben. Den Plot können wir in die Zwischenablage kopieren und den Inhalt in den Konstruktor bei **self.codeTable** = { } anstelle von { } einfügen.

```
    def scan(self):
        taste=input("Tasten-Name: ")
        if taste == "quit":
            print(self.codeTable)
            self.debug and print(taste)
            return taste
        else:
            if self.keysPresent():
                sc=self.readKey()
                print(taste, hex(sc))
                self.codeTable[hex(sc)]=taste
```

Wenn Bytes da sind, holen wir den Code, geben Tastenname und Hex-String des Tastencodes aus und tragen das Schlüssel-Wert-Paar in **codeTable** ein. Der hex-String ist der neue Schlüssel und der Tastenname der ASCII-Code. Am Schluss benutzen wir diese Routine, um in einem Durchgang die Codetabelle aufzubauen.

In der Regel wird man nicht den Tastencode brauchen, sondern den ASCII-Code, **getChar**() erfüllt uns den Wunsch. Wenn Bytes da sind, holt die Routine einen Tastencode und gibt den ASCII-Code zurück.

```
    def getChar(self):
        if self.keysPresent():
            sc=hex(self.readKey())
            return self.toAscii(sc)
```

Eingaben längerer Texte sollen auch möglich sein. **getWord**() macht das. Mit dem leeren String deklarieren wir **word** und **c**. in der Endlosschleife warten wir unendlich lang auf eine Taste. Der Tastencode der Enter-Tasten ist **0x5a**. Damit beenden wir die Eingabezeile und verlassen die Schleife. Alle anderen Tasten werden in ASCII-Code umgewandelt, in der Eingabezeile ausgegeben und an **word** angehängt.

```
    def getWord(self):
        word=""
        c = ""
        while 1:
            c=self.awaitKey(0)
            if c == int("0x5a"):
                print("\n")
                break
            else:
                c=self.toAscii(hex(c))
                print(c, end="")
                word += c
        return word
```

Beim Löschen des Empfangspuffers mit **flushBuffer**() schauen wir nach, ob Bytes da sind, warten dann die obligatorischen 200ms und lesen alles ein, was der Puffer zu bieten hat, um es ins Nirwana zu schicken.

Auf die Klassendeklaration folgt ein kurzes Programm, das (nur) dann ausgeführt wird, wenn das Modul **ps2.py** als Hauptprogramm gestartet wird.

```
if __name__ == "__main__":
    # Tastencodes einlesen
    # Tasten fuer die Code-Tabelle scannen
    ps2=PS2()
    key=""
    while 1:
        key=ps2.scan()
        if key=="quit":
            break
```

Ein PS2-Objekt wird instanziiert und **key** gelöscht. In der Hauptschleife rufen wir die Methode **scan**(), um einen Eintrag an die Codetabelle anzufügen. Liefert **scan**() "quit" zurück, verlassen wir die Schleife.

Auf der PS/2-Tastatur wird eine Taste gedrückt, und dann auf der PC-Tastatur der entsprechende Tastenname eingegeben. Das kann auch im Block geschehen, was die Sache sehr erleichtert. Wir können zum Beispiel eine ganze Tastenreihe auf der PS/2-Tastatur eintippen und dann im gleichen Zug von der PC-Tastatur aus in die Tabelle schicken. Die Reihenfolge muss natürlich dieselbe sein. Nachdem der Inhalt eines Dictionarys in zufälliger Folge ausgegeben wird, kommt es bei der Erfassung nicht auf eine alphabetische Reihenfolge an. Einzelne Tasten lassen sich auch nachträglich mit **scan**() erfassen und durch copy and paste in die Tabelle einfügen.

Was kommt als Nächstes?

Nun das Ziel ist ein Wecker mit Spezialfunktionen. Dazu ist eine möglichst genaue Uhr mit Weckfunktion nötig. Genau damit beschäftigt sich die nächste Folge, in der wir einen DS3231 in Dienst stellen werden.

Bis dann!