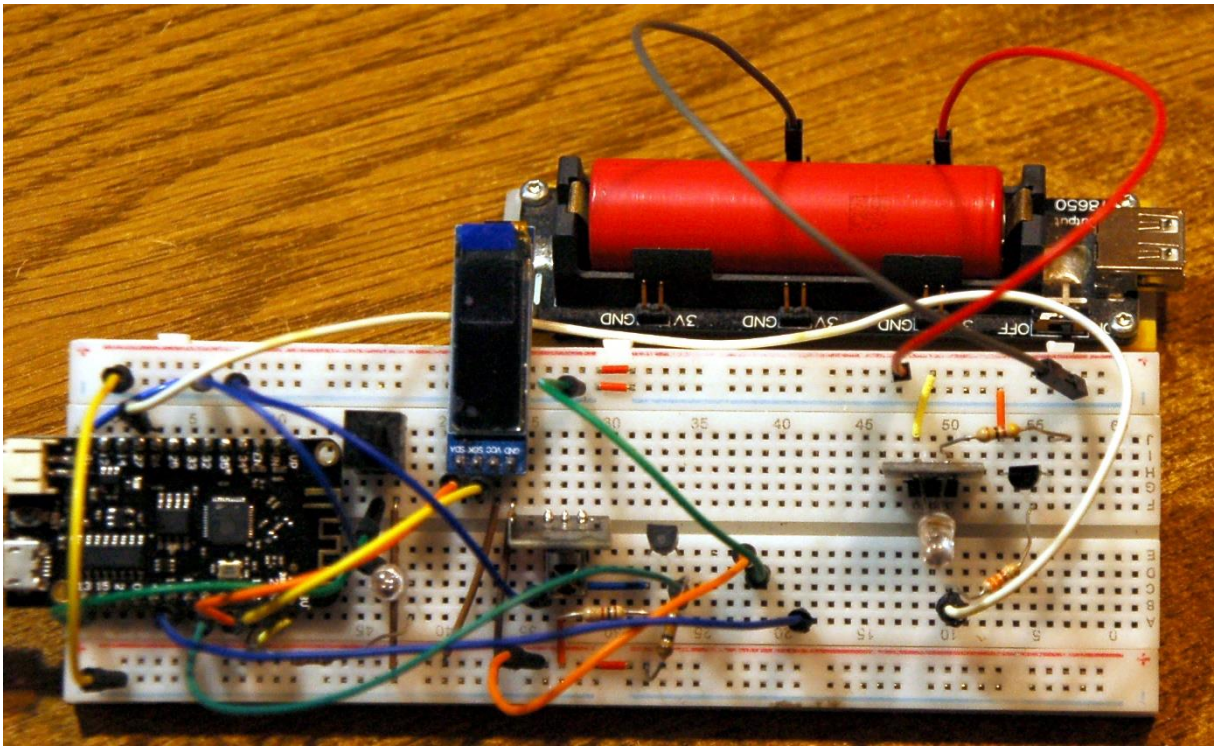

Zuwachs auf dem Breadboard

Diesen Beitrag gibt es auch als PDF-Dokument.

Nachdem der ESP32 inzwischen IR-Codes von einer RC (remote Control) übernehmen kann, will ich ihm heute beibringen, die gelernten RC-Codes auch zu senden. Damit erreichen wir, dass der ESP32 selbsttätig als RC arbeitet, um zum Beispiel, veranlasst durch Sensorwerte oder empfangene Funknachrichten, Geräte anzusteuern, die auf IR-Signale reagieren. Das kann ein Fernseher, eine Stereoanlage oder ein Rekorder und so weiter sein.

Auf dem Weg zu einer guten Lösung hat mir der Logic Analyzer wieder super Dienste geleistet, denn er kann uns in Verbindung mit dem kostenlosen Programm Logic 2 von Saleae sagen, welche Struktur die vom ESP32 gesendeten Bursts (kurze Pulsfolgen) haben. Natürlich verrät der Logic Analyzer uns auch die Dauer der Bursts und der einzelnen Pulse der Folgen. Daraus können wir die Frequenz berechnen und nachprüfen, ob die Zeitlimits des Protokolls eingehalten werden.

Wie das im Einzelnen funktioniert und welche Möglichkeiten das Modul RMT bietet, das verrate ich in der heutigen zweiten Folge zum Themenkreis RC-Steuerungen aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Der ESP32 sendet IR-Sequenzen

Zur bisherigen Schaltung kommen einige neue Bauteile hinzu. Im Zentrum steht die IR-Sendediode KY-005 zusammen mit einem Transistor und zwei Widerständen, die

gemeinsam das schwache Signal vom GPIO-Pin gehörig verstärken. Zur Anzeige von Bedieninformationen verwende ich ein kleines OLED-Display und als Auslöser für die ersten Transferaufträge kommt eine Taste dazu. Die Periode, in der mit der Taste Vorgänge ausgelöst werden können, zeigt eine LED von beliebiger Farbe an.

Die Schaltung

Die Schaltung aus dem ersten Teil des Projekts habe ich direkt übernommen und für den Sendebetrieb aufgerüstet.

Aufgebaut ist also immer noch alles auf dem langen Breadboard mit 62 Kontaktreihen. Gut, dass ich das lange Board genommen habe, so musste ich nicht umbauen, denn das kleinere Board hätte mich jetzt doch in Bedrängnis im Hinblick auf Steckplätze gebracht.

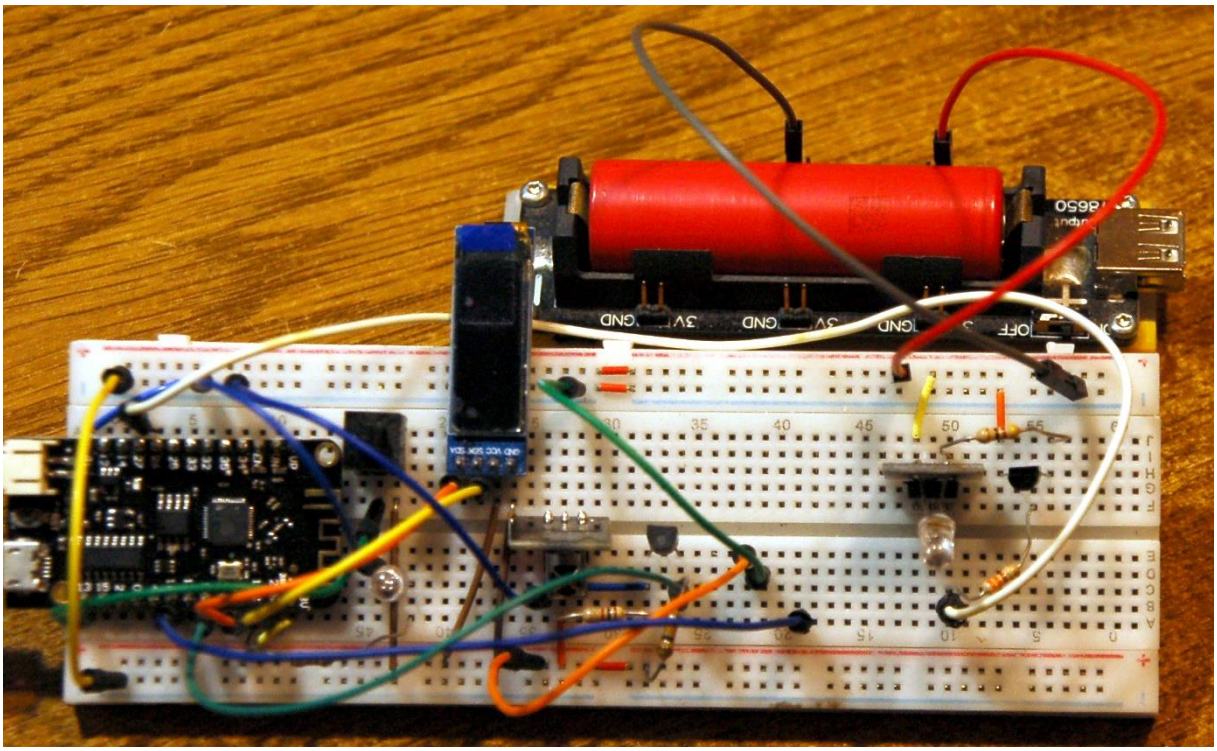

Abbildung 1: Zuwachs auf dem Breadboard

Abbildung 2 zeigt die Sendeeinheit im Detail. Aber das Schaltbild bildet die Verdrahtung noch wesentlich besser ab.

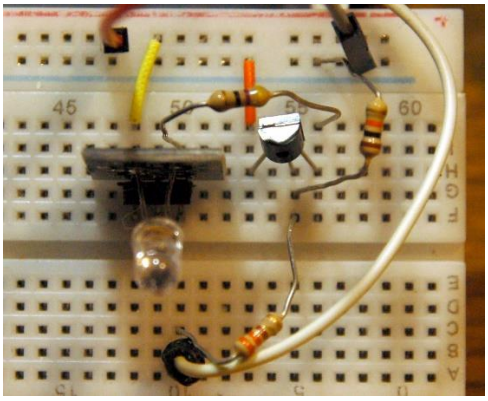

Abbildung 2: Sendeeinheit

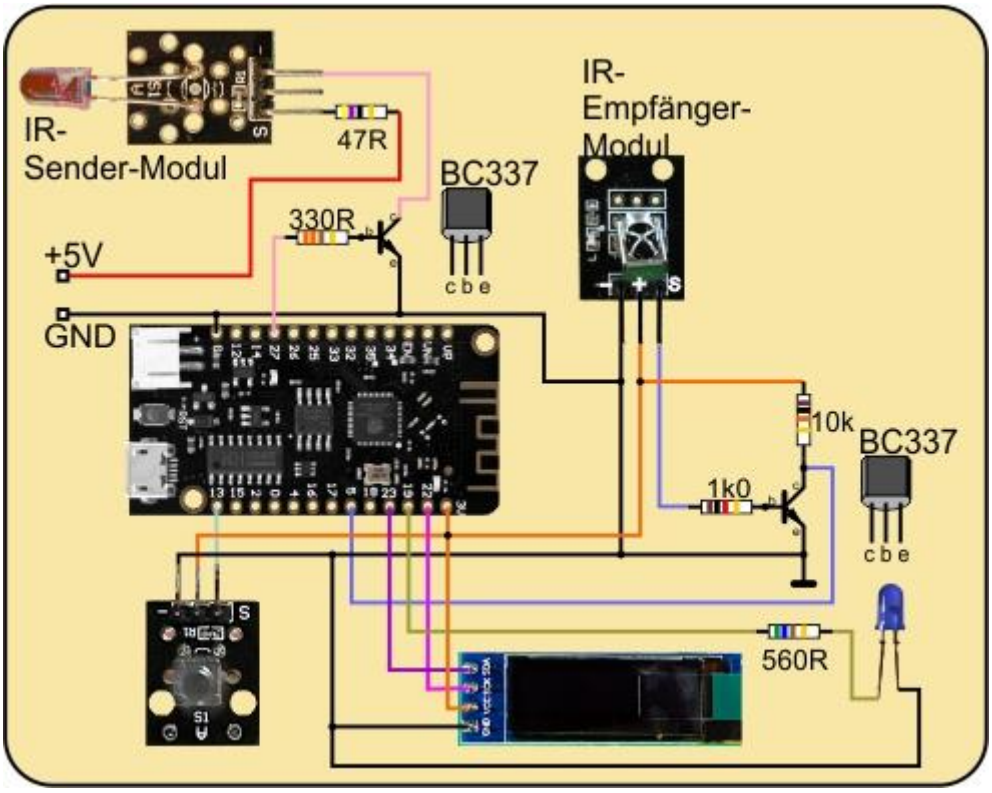

Abbildung 3: Erweiterter Aufbau - Schaltung

Um die Sendeleistung zu steigern, wird die IR-Sende-LED nicht an 3,3V sondern an 5V betrieben. Weil dieser Controller keinen 5V-Anschluss an den Stiftleisten hat, musste ich eine externe 5V-Quelle einsetzen. Der verwendete Li-Akku mit Zellenhalter ist einfacher zu handhaben als ein Steckernetzteil und bietet den Vorteil der mobilen Verwendung, weil die Halterelektronik auch die 3,3V zur Verfügung stellt, die der Rest der Schaltung benötigt.

Mit dem Widerstand von 47Ω in der Kollektorleitung kann man die Stromstärke während der Pulse einstellen und damit die maximale Reichweite des Senders festlegen. Durch den Widerstand von 47 Ω, also auch durch die Diode, fließt bei Vcc=5V aber ein Strom von 73mA, wenn der Transistor durchschaltet. Das wäre suboptimal für die IR-LED, die einen Dauerstrom von ca. 20 bis 30mA verträgt.

Eine Lebensversicherung für die IR-Diode ist daher ein weiterer Schaltungskniff, der 10kΩ-Widerstand von der Basis des Schalttransistors auf GND. Er verhindert, dass der Transistor in der Einschaltphase, während der GPIO-Pin noch als Eingang läuft, unkontrolliert durchschaltet. Für kurze Impulse von wenigen Microsekunden ist das OK, aber nicht für längeren Dauerstrom. Der 10kΩ-Widerstand zieht die Basis auf GND-Potenzial, sodass der Transistor sperrt, bis er einen eindeutigen 3,3V-Pegel von GPIO16 über den 330Ω-Widerstand bekommt. GPIO16 arbeitet nach der Initialisierung als Ausgang und liefert kontrollierte Pegel.

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
2	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 mit Programm Logic 2 von Saleae
1	Li-Akku (18650) mit Halter oder Netzteil

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 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 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.

Lichtfunk

OK, denken wir jetzt mal an den ersten Teil zurück. Damit der Fernseher mit dem ESP32 zusammenarbeitet, müssen wir ihm Pulse und Pausen von 889µs oder 1778µs senden. Aber damit nicht genug, die Pulse müssen noch mit einer Frequenz von 36kHz oder 38kHz überlagert werden. Man kann es auch anders ausdrücken, wir müssen die Frequenz von 38kHz mit den Puls-Pause-Längen modulieren. In der HF-Technik, wäre das eine Amplitudenmodulation mit einer Tiefe von 100% - Bei einem Puls wird der Träger von 38kHz gesendet, während Pausen wird nix gesendet – Funkstille.

Da gibt es nur ein Problem. Der ESP32 ist unter MicroPython nicht schnell genug, um die Forderungen via MicroPython-Programm umzusetzen. Aber es gibt eine Lösung und die heißt RMT. Die Klasse **RMT** wohnt im Modul **esp32** und wurde genau für Fälle, wie den unseren, implementiert.

Um komplexe Pulsfolgen zu erzeugen, brauchen wir erst einmal ein RMT-Objekt. Es gibt davon drei Varianten. Wir verwenden Modus 1. Hier wird eine Liste oder ein Tupel erstellt, das Puls-Pause-Paare enthält. Nach jedem Wert wechselt der Pegel am Ausgangspin. Die feinste Auflösung ist 12,5ns (1ns = 1 Milliardstel Sekunde). So genau brauchen wir gar nicht zu dosieren, uns reicht 1µs als Basislänge. Also teilen wir den RMT-Takt von 80MHz durch 80. Ich habe den Logic Analyzer mit Kanal 0 an GPIO27 angeschlossen.

```
>>> from machine import Pin
>>> from esp32 import RMT
>>> rmtPin = Pin(27, Pin.OUT, value = 0)
>>> rmt = RMT(0, pin=rmtPin,
        idle_level=0,
        clock_div=80,
        )
>>> pulses=(500,1000,1500,2000)
```

Ich starte das Sampeln mit der Taste R und schicke dann in Thonny den Befehl ab.

```
>>> rmt.write_pulses(pulses,0)
```

Die erste Kurvenform sieht jetzt so aus. Der Wert 500 wird als Ruhepegel (idle-level), also als Pause ausgegeben und erscheint daher nicht im Plot.

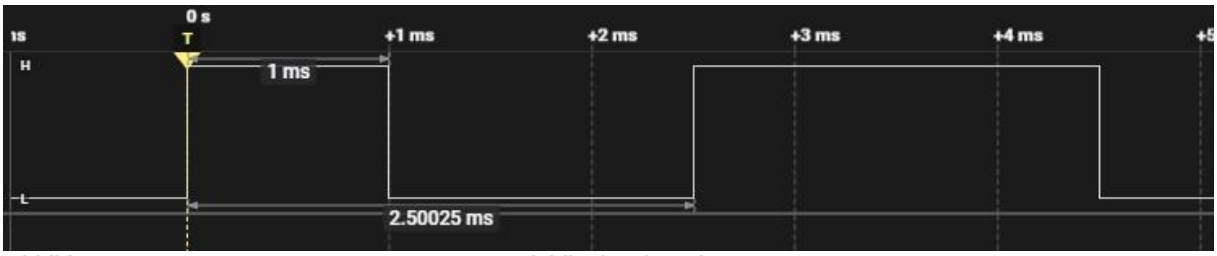

Abbildung 4: 500-1000-1500-2000 an GPIO27 mit idle_level=0+data=0

Abbildung 5: 500-1000-1500-2000 an GPIO27 mit idle_level=0+data=1

Wir starten erneut ein Sampling mit **idle-level**=0, setzen jetzt aber **data** auf 1. Der erste Wert erscheint nun als Puls der Länge 500µs.

Weiter, **idle-level** wird auf 1 gesetzt.

```
>>> rmt = RMT(0, pin=rmtPin,
        idle_level=1,
        clock_div=80,
        )
>>> rmt.write_pulses(pulses,0)
```

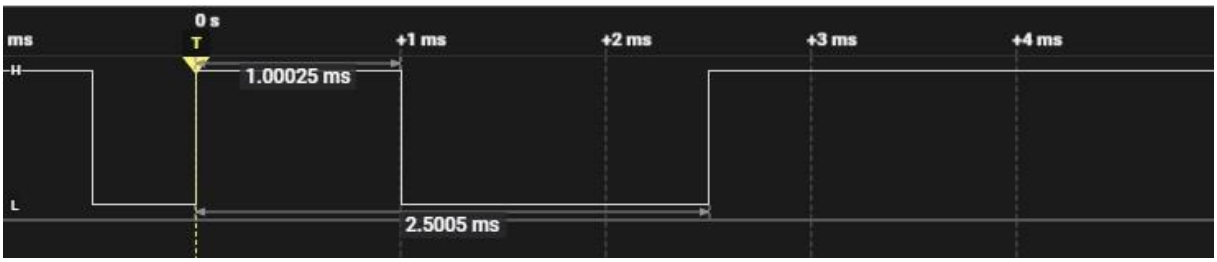

Abbildung 6: 500-1000-1500-2000 an GPIO27 mit idle_level=1+data=0

Und zuletzt beide Werte auf 1, das gibt den invertierten ersten Fall.

```
>>> rmt.write_pulses(pulses,1)
```

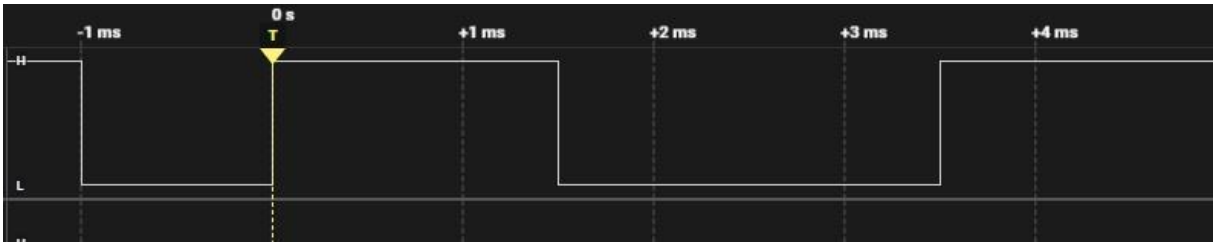

Abbildung 7: 500-1000-1500-2000 an GPIO27 mit idle_level=1+data=1

In Teil 1, IR-Empfänger, habe ich geschrieben, dass jeder RC5-Code mit einem 1-Bit beginnen muss. Das bedeutet, dass eine Pause vorausgehen muss, denn der Pegel in der zweiten Takthälfte entscheidet über eine 0 oder 1. Wir müssen auch bedenken, dass der Transistor den logischen Pegel negiert. Und, dass die IR-LED feuert, wenn der Transistor durchgesteuert wird (Basis auf 3,3V) und die Kathode der LED auf GND zieht. Die LED gibt kein Licht ab, wenn der Transistor sperrt (Basis auf GND). Welches ist nun die richtige Konfiguration?

Ja, das muss wohl Version 2 sein. So sieht es an GPIO27 aus.

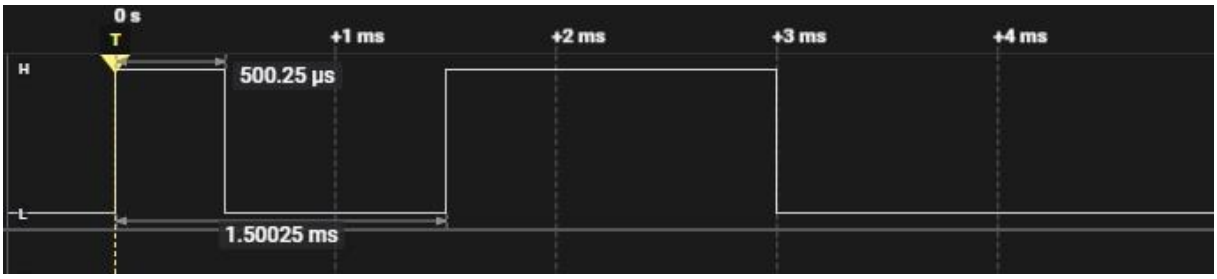

Abbildung 8: 500-1000-1500-2000 an GPIO27 mit idle_level=0+data=1

Gut, aber wir brauchen ja kein Dauerlicht, sondern wir müssen den Puls von 889µs in 32 Einzelpulse von ca. 27µs Periodendauer aufteilen, um auf die geforderte Frequenz von 38kHz zu kommen. Auch das kann RMT. Wir brauchen nur einen weiteren Parameter bei der Instanziierung des RMT-Objekts zu ergänzen. Das Tupel, das wir an **tx_carrier** übergeben, besteht aus der Trägerfrequenz in Hertz, dem Duty Cycle in Prozent und wiederum einer Pegelangabe.

```
>>> cfreq=38000
>>> duty=50
>>> rmt = RMT(0, pin=rmtPin,
        idle_level=0,
        clock_div=80,
        tx_carrier = (cfreq, duty, 1))
```

Am Ausgang GPIO27 erscheint nun ein 38kHz-Burst immer dann, wenn das Rechtecksignal auf LOW-liegt. Das sieht dann so aus. Genau das ist es, was wir haben wollen. Jeder zweite Wert in der Folge soll den Bit-Zustand widerspiegeln. Damit das deutlich wird, habe ich hier nicht die RC5-Abfolge gewählt, sondern Zeitintervalle, die ganz leicht auf die Plots zugeordnet werden können.

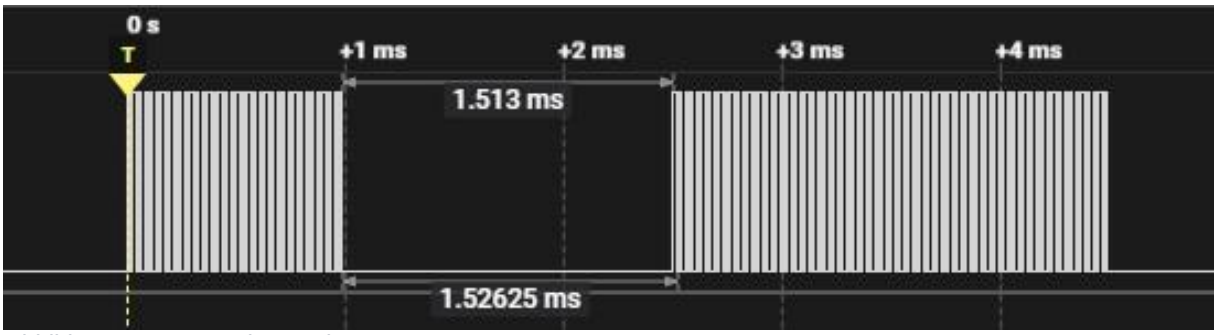

Abbildung 9: RC5-Code gepulst

Nachdem wir unseren Weg grundsätzlich gefunden haben, wenden wir uns dem Modul **irsend.py** mit der Klasse **IRSEND** zu. Sie beinhaltet überschaubare 62 Codezeilen.

Die Klasse IRSEND

Wir holen uns die Klasse **array** für den Aufbau des Tupels mit den Puls-Pause-Zeiten. Das Modul **timeout**() liefert uns die Closure **TimeOutMs**() für einen nicht blockierenden Softwaretimer. Weil viel Datenmüll entsteht, holen wir uns den Müllsammler **collect** aus dem Modul **gc**.

```
from array import array
from timeout import *
from gc import collect
```

Es folgt die Deklaration der Klasse **IRSEND** mit dem Klassen-Attribut **PULS**, wodurch wir die halbe Bit-Länge auf 889µs festlegen.

Der Konstruktor nimmt eine ganze Reihe von Parametern, drei davon sind optionale Schlüsselwortparameter. Dem Positionsparameter **rmt** müssen wir ein RMT-Objekt übergeben, das im aufrufenden Programm instanziiert werden muss. **pulsmax** setzt die maximale Anzahl von Pulsflanken in einem RC5-Wort fest, **tpackage** die maximale Zeitdauer des Pakets. Mit **rdelay** definieren wir die Sendepause zwischen den zu sendenden Paketen. Über **debug** steuern wir die Ausgabe von Meldungen zur Laufzeit. Wenn die Werte von den eingetragenen Defaultwerten nicht abweichen, müssen sie beim Erzeugen der Instanz nicht angegeben werden.

```
    def __init__(self,rmt,
                 pulsmax=28,
                 tpackage=30,
                 rdelay=90,
                 debug=False):
        self.rmt=rmt
        self.tpackage=tpackage
        self.rdelay=rdelay
        self.arr = array('H', 0 for _ in range(pulsmax))
        self.aptr=0
        self.pegel=False
        self.toggle=False
        self.debug=debug
```

Diese Argument-Werte landen dann in Instanzattributen. Wir erzeugen ein Array der maximalen Größe für die Aufnahme von 16-Bit -Werten. Ein Pointer (Index) ins Array wird gebraucht, **.aptr**, und auf 0 gesetzt. **.pegel** steuert den Flankenwechsel und **.toggle** das Toggelbit.

Die Methode **append**() habe ich mir aus dem Paket von Peter Hinch ausgeliehen. Sie erledigt das Einfügen eines Intervalls oder mehrerer Intervalle in das Array **arr**. Bei jedem Aufruf von **.append**() werden die in der Liste **times** enthaltenen Intervallzeiten ins das Array eingetragen. Dabei wird jedes Mal der Index erhöht und der Pegel umgeschaltet.

```
    def append(self, *times):
        for t in times:
            self.arr[self.aptr] = t
            self.aptr += 1
            self.pegel = not self.pegel
```

In gewissen Fällen muss die Dauer eines Intervalls verlängert werden, Das ist der Fall, wenn der Pegel des Nutzsignals wechselt. In diesem Fall ist es nötig, die Dauer des vorhergehenden Intervalls um 889µs zu verlängern. Das macht die Methode **add**(), die ich mir auch von Peter ausgeliehen habe.

```
    def add(self, t):
        assert t > 0
        self.arr[self.aptr - 1] += t
```

Die Wirkungsweise der beiden Methoden habe ich im Tabellenblatt Codiertabelle.ods dargestellt. Dazu kommen wir gleich.

Die Reihenfolge der Intervallzeiten wird in der Methode **codeIt**() festgelegt. Die Adresse der RC und das Datenwort werden als Positionsparameter übergeben. Dann setzen wir den Array-Index **aptr** auf 0 und das Attribut **.pegel** auf False. **.pegel** kennzeichnet den logischen Zustand des Manchester-Signals. Geräteadresse und Kommandowort setzen wir an die, ihnen zugeordneten Bitpositionen. Eine Sonderstellung hat dabei das siebte Bit des Datenworts, das invertiert an Position 12 zu befördern ist. Das Toggle-Bit landet an Position 11. Wir lassen uns den zu sendenden Code anzeigen und stellen die Maske für den Transfer auf 0x2000.

```
    def codeIt(self, addr, data):
        self.aptr=0
        self.carrier=False
        d = (data & 0x3f) | ((addr & 0x1f) << 6) | \
            (((data & 0x40) ^ 0x40) << 6) | \
            ((int(self.toggle) & 1) << 11 )
        print(bin(d))
        mask = 0x2000
```

Das Maskenbit wird nun mit jedem Durchlauf der while-Schleife um eine Position nach rechts verschoben. Das Procedere endet, wenn der Wert 0 erreicht wird, was als **False** interpretiert wird. Dadurch bricht dann der Schleifendurchlauf ab.

```
        while mask:
            if mask == 0x2000:
                self.append(PULS)
            else:
                bit = bool(d & mask)
                if bit ^ self.pegel:
                    self.add(PULS)
                    self.append(PULS)
                else:
                    self.append(PULS, PULS)
            mask >>= 1
```

Gleich beim ersten Lauf muss ein Puls angefügt werden, das 1-Bit. Danach picken wir durch Undieren des Datenworts **d** mit der Maske **mask** jeweils ein Bit heraus und wandeln das Ergebnis, 0 oder 1, in einen boolschen Wert, False oder True, um. Das Ergebnis wird dann mit dem momentanen Zielzustand in **.pegel** exoderiert. Die Wahrheitstabelle dieser Verknüpfung finden Sie in Abbildung 10.

Ist das Ergebnis True, dann wird infolge ein Pegelwechsel herbeigeführt, indem die Intervallzeit des vorigen Pegels verlängert und eine normale Intervallzeit von 889µs angehängt wird. Abbildung 11 zeigt das Geschehen in den einzelnen Durchgängen der while-Schleife. Das dazugehörende Tabellenblatt können Sie herunterladen.

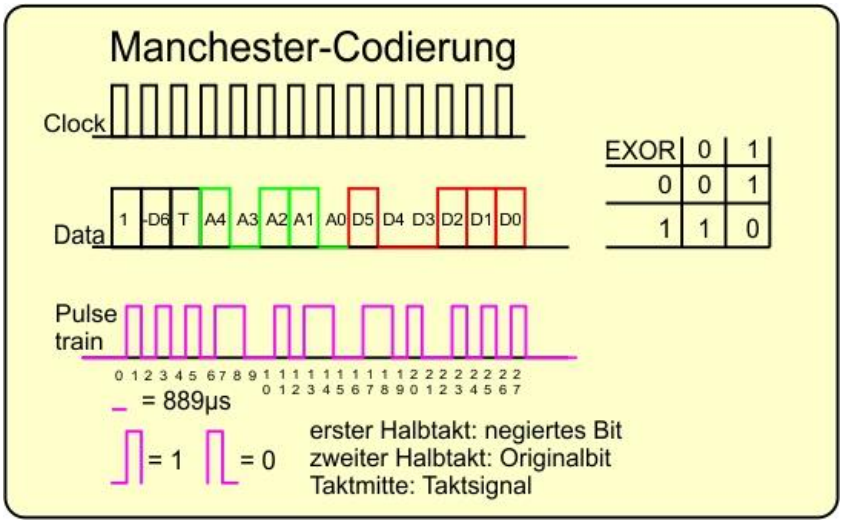

Abbildung 10: Manchester-Codeierung

Codiertabelle

data		1	1	1	0	0	0	0	0	0	0	0	0	1	0	0														
mask		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	Das Maskenbit wird von links nach rechts durchgeschoben													
Bit = data & mask			1	1	0	0	0	0	0	0	0	0	0	1	0	0														
pegel		0	1	1	1	0	0	0	0	0	0	0	0	0	1	0														
Bit ^ pegel		0	0	0	1	0	0	0	0	0	0	0	0	1	1	0														
aptr		0	1	3	5	6	8	10	12	14	16	18	20	21	22	24														
		ap	apap	apap	+ap	apap	apap	apap	apap	apap	apap	apap	+ap	+ap	apap		+ap	vorige Zeit verlängern und Pulszeit anfügen												
																	apap	zwei Pulszeiten anfügen												
	arr																													
Durchgang		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
	0	889	x																											
	1		889	889	x						x	Position des Index: aptr																		
	2				889	889	x																							
	3					1778	889	x																						
	4							889	889	x																				
	5									889	889	x																		
	6											889	889	x																
	7													889	889	x														
	8															889	889	x												
	9																	889	889	x										
	10																			889	889	x								
	11																				1778	889	x							
	12																					1778	889	x						
	13																							889	889	x				
	14	Schicht im Schacht																												
Impulsschema																														

Abbildung 11: Codiertabelle

Nachdem die Zeiten gesetzt sind, folgt am Ende jedes Durchlaufs das Schieben des Maskenbits.

Die Methode **transmit**() fasst das Codieren, das Senden und die Verwaltung des Toggle-Bits zusammen. Sie nimmt die Geräteadresse, den Kommando-Code und die Anzahl von Sendewiederholungen mit 2 als Defaultwert. **.codeIt**() stellt dann das

Array mit den Intervallzeiten bereit. Die for-Schleife sorgt für die Wiederholungen, zwischen denen eine Pause von **.rdelay** Millisekunden liegen muss.

Den Timer **over** stellen wir auf die Zeitdauer eines Manchester-Pakets ein, **tpackage** Millisekunden. **over** ist ein nicht blockierender Software-Timer, während dessen Ablaufs andere Dinge erledigt werden können. Möglich wird das durch eine sogenannte Closure. Folgen Sie dem Link, um mehr darüber zu erfahren.

```
    def transmit(self, addr,data,repeat=2):
        self.codeIt(addr,data)
        for _ in range(repeat):
            over=TimeOutMs(self.tpackage)
            self.rmt.write_pulses(tuple(self.arr),1)
            while not over():
                pass
            delayed=TimeOutMs(self.rdelay)
            while not delayed():
                pass
        self.toggle= not self.toggle
        collect()
```

Mit **.rmt.write_pulses**() lassen wir die Pulsfolge am Pin 27 ausgeben und warten dann auf den Ablauf des Timers. Das tritt ein, wenn die Funktion **over**() **True** zurückgibt. Dann können wir sicher sein, dass der Transfer des Blocks beendet ist.

Ein weiterer Timer wird auf die Pausenzeit zwischen den Paketen eingestellt. Wir warten, bis er abgelaufen ist und wiederholen den Transfer. Die wiederholten Pakete enthalten denselben Wert des Toggle-Bits.

Ist alles erledigt, dann negieren wir das Toggle-Bit für den nächsten Aufruf und räumen den Datenmüll weg.

Das Programm

Das Programm **send**() vereint das Programm **lern.py** aus dem ersten Teil, das wir als Funktion **learn**() einbauen mit den Vorbereitungen, die für das Senden von IR-Codes nötig sind. Tatsächlich gesendet wird vorerst von Hand. Nachdem **send**() einmal gestartet wurde.

Wir haben eine ganze Latte an Importen.

```
from esp32 import RMT
from machine import Pin,SoftI2C
from irsend import IRSEND
from buttons import waitForTouch, waitForRelease, Buttons
from time import sleep
from oled import OLED
from gc import collect
from sys import exit
collect()
```

Rahmendaten für den IR-Transfer werden deklariert und definiert wie in Teil 1.

```
data=0x04
addr=0x00
cfreq=38000
duty=50
```

Dann instanziieren wir ein Tastenobjekt an Pin 13. Damit verbunden ist eine LED an GPIO19, die immer dann leuchtet, wenn auf die Taste gewartet wird.

```
taste=Buttons(13,invert=True,
              pull=True,
              name="lern",
              ledPin=19,
              active=1)
```

Jetzt legen wir den RMT-Pin fest und erzeugen das **RMT**-Objekt **rmt**. Damit instanziieren wir das IRSEND-Objekt **ir**.

```
rmtPin = Pin(27, Pin.OUT, value = 0)
rmt = RMT(0, pin=rmtPin,
          idle_level=0,
          clock_div=80,
          tx_carrier = (cfreq, duty, 1))
ir=IRSEND(rmt)
```

Das I2C-Objekt brauchen wir für die OLED-Instanz.

```
i2c=SoftI2C(scl=Pin(22), sda=Pin(23), freq=100000)
d=OLED(i2c,heightw=32)
```

An GPIO5 speisen wir das Signal vom IR-Empfänger-Modul ein.

```
pLearn = Pin(5, Pin.IN) # IR-Eingangspin
```

Die Variablen für den Lernprozess kennen wir aus dem ersten Teil.

```
keypressed=False
Data,Addr,Ctrl=0,0,0
rcCode=[]
```

Die Funktion **learn**() war im ersten Teil das Hauptprogramm. Eine genaue Beschreibung finden Sie dort.

```
def learn(p):
    global keypressed,rcCode
    def cb(data, addr, ctrl):
        global keypressed,rcCode
        if data < 0:  # NEC protocol sends repeat codes.
            print("Repeat code.")
        else:
            keypressed=True
            rcCode.append((data, addr, ctrl))

    ir = RC5_IR(p, cb)  # Instantiate receiver

    f=open("befehle.cfg","w")
    while True:
        t=input("Tasten-Name auf der RC -> ")
        t=(t.strip("\n\r"))
        if t=="q":
            f.close()
            ir.close()
            return
        else:
            print(t,"-Taste an der RC kurz drücken")
            while 1:
                if keypressed:
                    sleep(0.2)
                    break
            Data,Addr,Ctrl=rcCode[0]
            rcCode=[]
            print(t,"->>",Data,Addr,Ctrl)

line=t+","+str(Data)+","+str(Addr)+","+str(Ctrl)+"\n" # oder
            #
line=",".join([t,str(Data),str(Addr),str(Ctrl),"\n"])
            f.write(line)
            keypressed=False

        gc.collect()
```

readData() liest die von **learn**() erstellte Datei **befehle.cfg** mit den Paaren aus RC-Tastenname und IR-Code. Zunächst deklarieren wir ein leeres Dictionary, **codes**. Das Display informiert uns darüber, was zu tun ist. Mit **try – except** fangen wir eventuell auftretende Datei-Fehler ab.

```
def readData():
    codes={}
    d.clearAll()
    d.writeAt("TRY READING",0,0,False)
    d.writeAt("befehle.cfg",0,1)
    try:
        with open("befehle.cfg","r") as f:
            for line in f:
                key,data,addr,ctrl=line.split(",")
```

```
                codes[key]=(data,addr,ctrl)
        d.writeAt("GOT KEY-CODES",0,1)
        waitForTouch(taste,3)
        d.clearAll()
        return codes
    except OSError as e:
        d.writeAt("NOT FOUND",0,2)
        waitForTouch(taste,3)
        d.clearAll()
        return None
```

Mit **with open** erzeugen wir ein Dateihandle zum Lesen aus der Datei. Zeile für Zeile wird gelesen und an den Kommas aufgeteilt. Die entstehende Liste entpacken wir in die Variablen **key**, **data**, **addr** und **ctrl**. **key** ist der Schlüssel für das Dictionary **codes** unter dem das Tupel (**data, addr, ctrl**) angelegt wird.

Die Erfolgsmeldung wird drei Sekunden lang angezeigt, dann löschen wir das Display und geben das Dictionary zurück ans Hauptprogramm. Beim Verlassen des with-Blocks wird die Datei automatisch geschlossen.

Except fängt eventuelle Fehler ab und meldet sie im Display.

Im Hauptprogramm starten wir mit dem Angebot, eine RC auszulesen und eine Codetabelle anzulegen. Wird die Taste innerhalb von 3 Sekunden gedrückt, startet die Lernphase.

```
d.writeAt("LEARN >>> Taste",0,0)
if waitForTouch(taste,3):
    from ir_rx.philips import RC5_IR
    learn(pLearn)
    d.clearAll()
    d.writeAt("CODES SAVED IN",0,0,False)
    d.writeAt("befehle.cfg",0,1)
    d.writeAt("weiter >> Taste",0,2)
    waitForTouch(taste,3)
```

Nach dem Löschen des Displays beenden wir erst mal das Programm. In der folgenden while-Schleife können dann Transferbefehle erteilt werden. Das tun wir dieses Mal aber nur manuell von REPL aus.

Die Schaltung steht? Die externen Module sind alle in den Flashspeicher des ESP32 hochgeladen? Dann kann's losgehen.

Wir starten **send.py** aus dem Editorfenster mit F5.

>>> data
4
>>> addr
0
>>> ir.transmit(addr,data,1)
>>> ir.transmit(addr,data,1)

Jetzt können wir den Logic Analyzer am Collector des Treibertransistors anschließen, Start mit Taste "R" in Logic2.

>>> ir.transmit(addr,data,1)

Je nach Stand des Toggle-Bits bekommen wir einen der folgenden Plots.

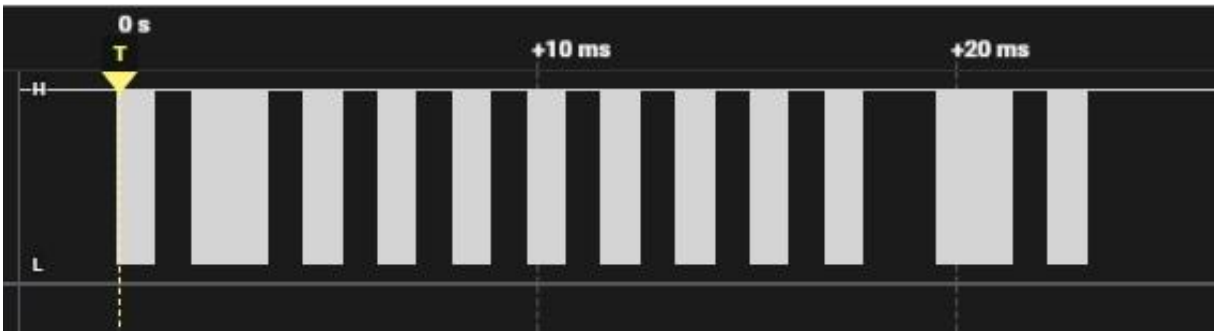

Abbildung 12: Toggle-Bit 0

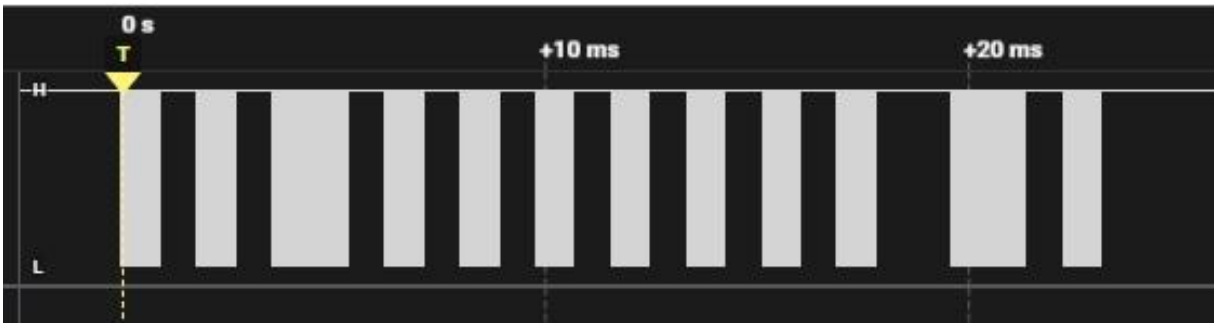

Abbildung 13: Toggle-Bit 1

Und wenn man in einen der Bursts hineinzoomt, erkennt man die 38kHz Feinstruktur.

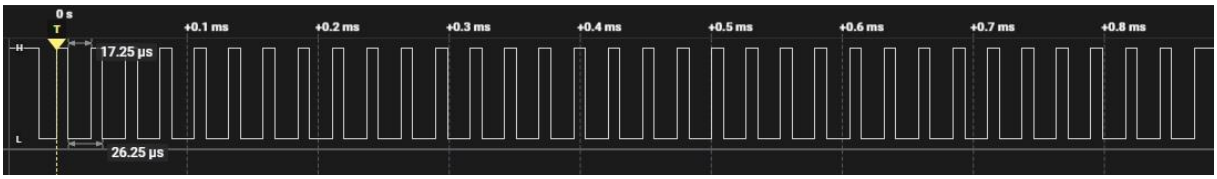

Abbildung 14: Burst vom Transfer von 0-4-1 Detail

Als Nächstes werden wir eine PS/2-Tastatur an den ESP32 anschließen und den Controller dazu bringen, Tastencodes in ASCII-Codes umzuwandeln. Dann können wir auch ohne PC die IR-Codetabellen auf dem ESP32 erstellen. Wobei eine große Tastatur aber auch noch diverse weitere Türen zu neuen Projekten öffnen kann.

Bis dann!