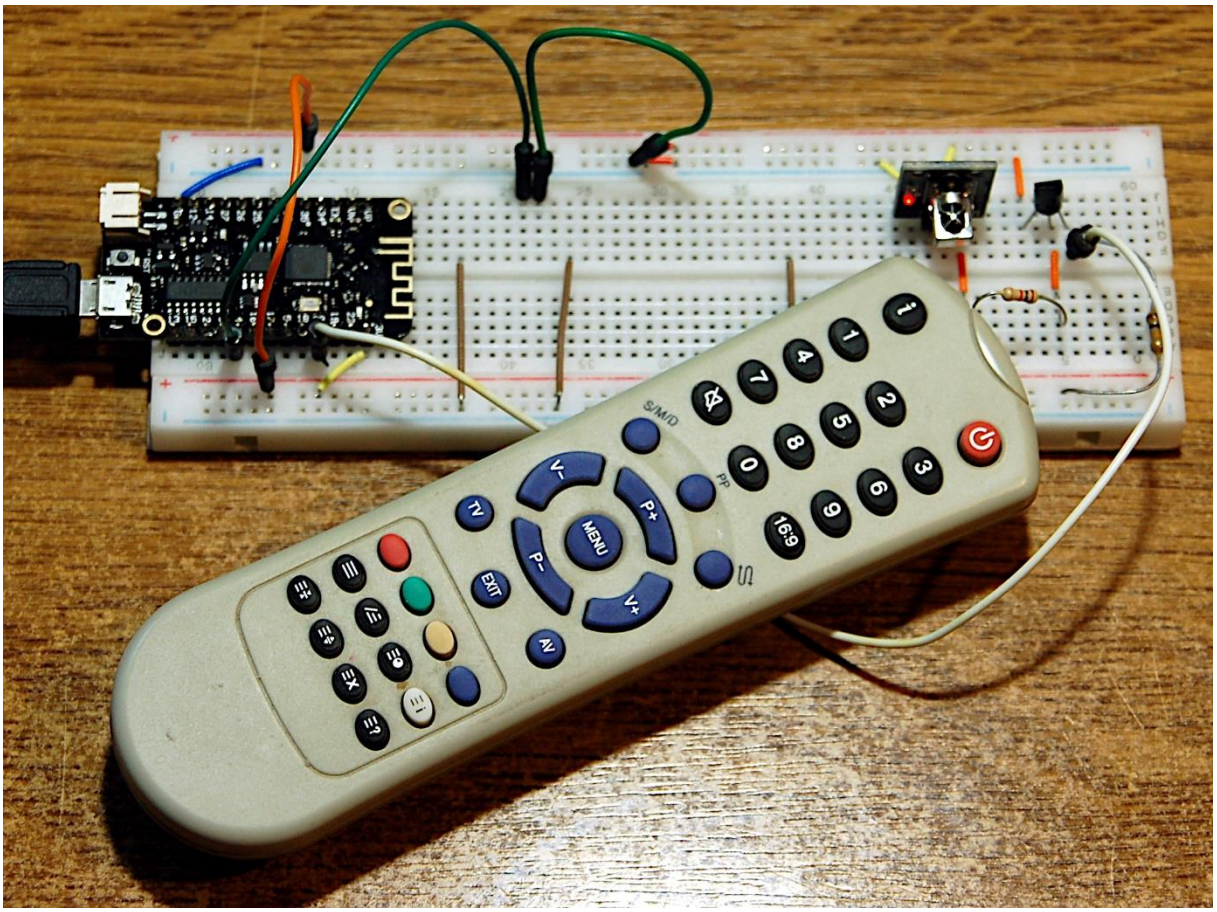

*IR-RC-Recorder*

Diesen Beitrag gibt es auch als PDF-Dokument.

Als Vorbereitung für ein größeres Projekt muss der ESP32 (ESP8266) von einer IR-Fernsteuerung lernen, wie ein Gerät angesprochen werden kann. Dazu muss er den Dialekt, das Protokoll, der RC (Remote Control) verinnerlichen. Er benötigt ein Auge, das Lichtimpulse im Bereich von ca. 950nm wahrnehmen kann, ein Spektralbereich, auf den das menschliche Sehorgan nicht anspricht.

Wir müssen also das, was die RC sendet, irgendwie darstellen und vermessen. Dann stellen wir die Ergebnisse in einer Datei auf dem Controller zusammen. Damit kann er dann Befehle, die er zum Beispiel per Funk von einem Smartphone erhält, an ein Gerät weitergeben, das auf IR-Signale hört.

Wie das im Einzelnen funktioniert, erfahren Sie im ersten Beitrag zu diesem Themenkreis im Rahmen der Reihe

# MicroPython auf dem ESP32 und ESP8266

heute

# Der ESP32 lernt RC5

Wir schauen uns erst einmal die Schaltung an und tragen die Bauteile zusammen. Dann gibt es Informationen zum Datentransfer vom RC-Sender zum ESP32. Wir nehmen den Manchester-Code unter die Lupe, welcher dem Übertragungsprotokoll zu Grunde liegt und besprechen dann das Programm und die beteiligten Module.

## Die Schaltung

Die Schaltung zum ersten Teil des Projekts ist genial einfach und daher besonders für Einsteiger geeignet. Sie besteht nur aus zwei Modulen, einem Transistor und zwei Widerständen. **Allerdings**, die Programmierung ist dagegen schon anspruchsvoller und greift einige Male tief in die MicroPython-Trickkiste.

Aufgebaut ist alles auf einem Breadboard. Ich hatte grade kein kleines zu Hand (30 Kontaktreihen) und habe daher eines mit 62 Kontaktreihen verwendet. Es sollte sich herausstellen, dass dieses ein Glücksgriff war, denn es kamen Stück für Stück diverse Bauteile und Module dazu, sodass das Board schließlich dicht bestückt war. Damit der Controller auch mit einem Board auskommt, habe ich mich für einen ESP32 Lolin entschieden, 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, um sämtliche Pins unterzubringen.

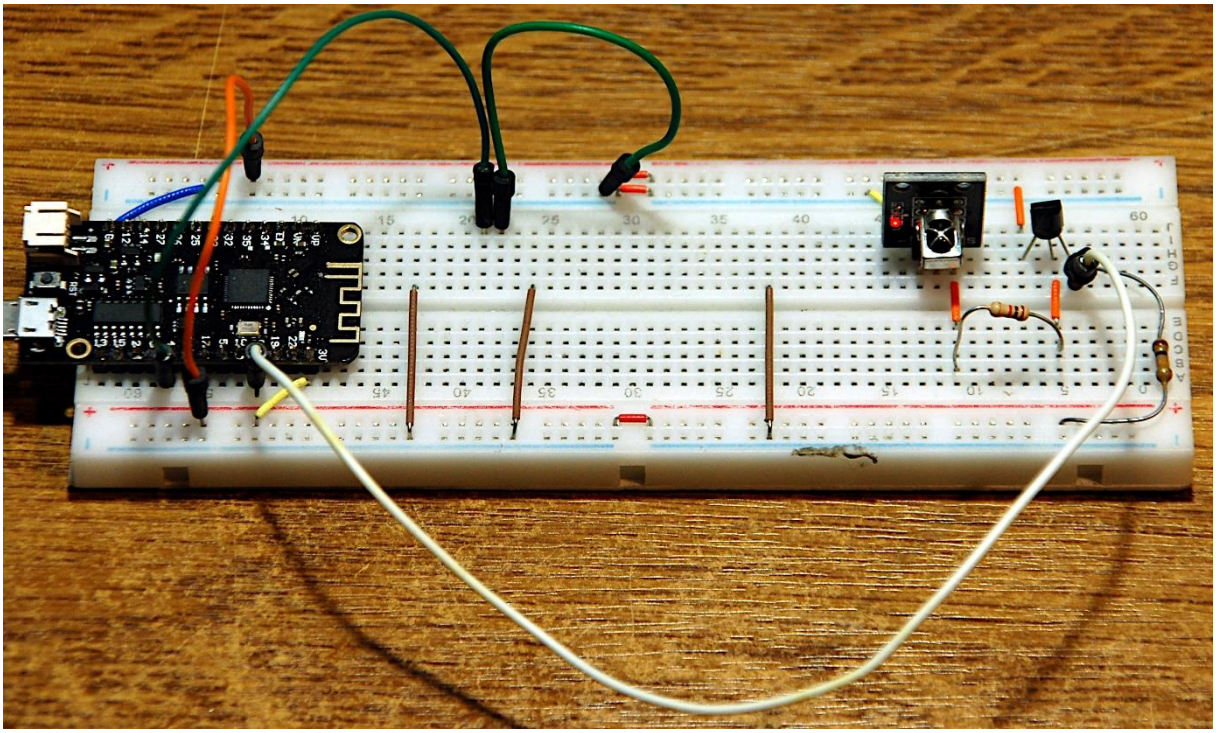

*Abbildung 1: RC-Empfänger - sehr übersichtlicher Aufbau*

Das Schaltbild zeigt die Verdrahtung etwas genauer.

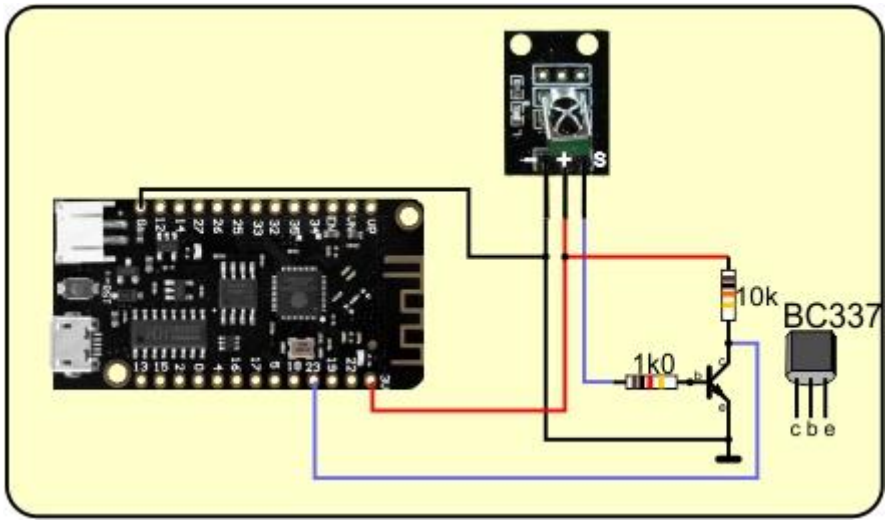

*Abbildung 2: Schaltung mit ESP32 Lolin*

Mit einem ESP8266 D1 mini sieht das so aus.

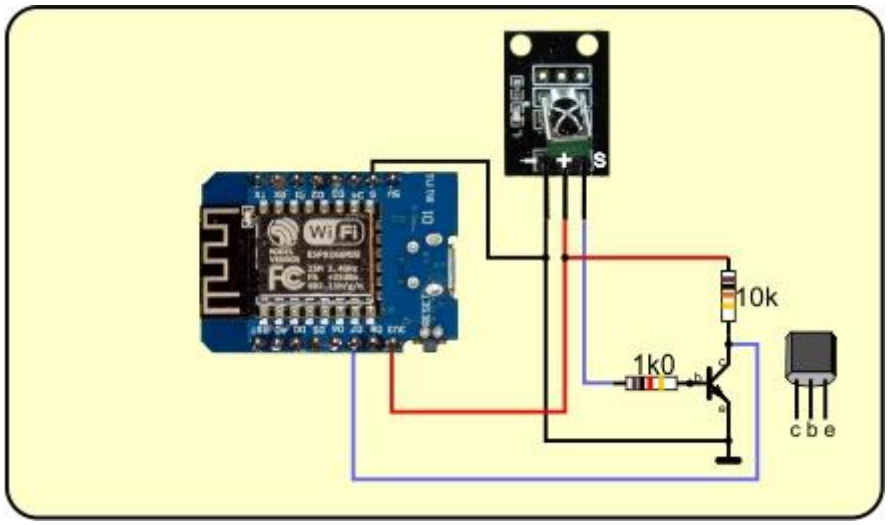

*Abbildung 3: Schaltung mit ESP8266 D1 mini*

Im Prinzip würde sogar ein ESP8266-01 ausreichen, weil wir ja nur eine GPIO-Leitung als Eingang benötigen. Wer die vorliegende Schaltung nur zum Auslesen oder Testen einer RC einsetzen möchte, kann also gut und gern einen ESP8266 verwenden. Wir wollen aber, dass der Controller auch IR-Signale senden kann. Genau das kann aber nur der ESP32. Dem ESP8266 fehlt dazu die Klasse **RMT** aus dem Modul **esp32**.

Der Transistor negiert den am Ausgang **S** des Empfänger-Moduls anliegenden logischen Pegel. Diese Maßnahme war nötig, weil die Programme von Peter Hinch, herzlichen Dank an Peter, mit den Pegeln des Senders arbeiten. Die Pegel des IR-Empfänger-Moduls sind im Vergleich dazu negiert. Statt am Programm etwas zu verändern habe ich mich für die Hardwarelösung mit dem Transistor entschieden.

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

## Hardware

Wie Sie vermuten, ist die Teileliste sehr übersichtlich.

| | |
|---|---|
| 1 | ESP32 Dev Kit C unverlötet<br>oder ESP32 NodeMCU Module WLAN WiFi Development Board<br>oder NodeMCU-ESP-32S-Kit<br>oder NodeMCU Lua Amica Modul V2<br>oder ESP8266 ESP-01S WLAN WiFi Modul<br>oder D1 Mini V3 NodeMCU mit ESP8266-12F |
| 1 | KY-022 Set IR Empfänger |
| 1 | Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord kompatibel mit Arduino und Raspberry Pi - 1x Set |
| diverse | Jumper Wire Kabel 3 x 40 STK |
| 1 | NPN-Transistor BC337 oder ähnlich |
| 1 | Widerstand 1,0 kΩ |
| 1 | Widerstand 10 kΩ |
| Optional | Logic Analyzer |

## Was die RC sendet

Die RC sendet nicht einfach für 1 IR-LED an und für 0 IR-LED aus. Das wäre viel zu störanfällig, denn jede Wärmequelle sendet auch IR-Licht aus. Solche IR-Emitter schalten aber nicht mit 36kHz ein und aus, das kann nur die RC. Je nach Hersteller kann diese Schaltfrequenz zwischen 36kHz und 40kHz liegen. Der Empfänger muss, oder sollte zumindest, auf derselben Frequenz wie der Sender arbeiten, um eine höhere Reichweite und sichere Übertragung zu gewährleisten.

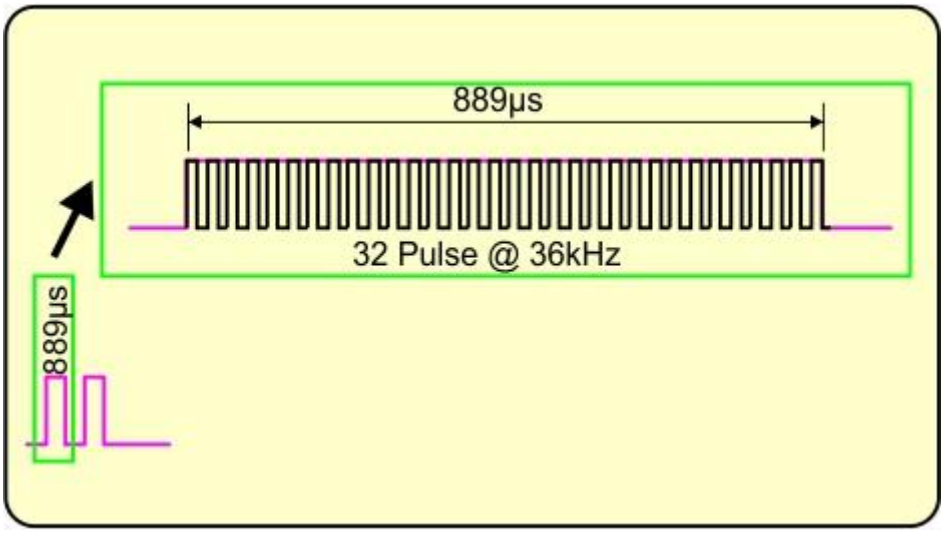


*Abbildung 4: Die RC sendet mit 36kHz*

Eine Pulsreihe (Burst) von 32 36-kHz-Pulsen steht dann für eine logische 1 oder 0, davor oder danach sendet die RC nicht. Die Pausen sind genauso lang wie die 36-kHz-Bursts. Eine logische 1 beginnt mit einer Pause, die logische 0 endet mit einer solchen. Abbildung 5 zeigt das für die Bitfolge 1-1-0.

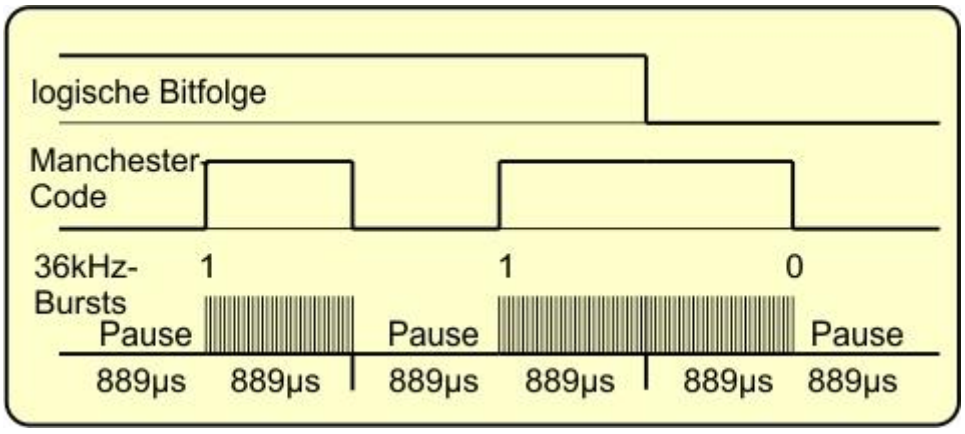

*Abbildung 5: Bitfolge 1-1-0*

Die Übertragung für ein Bit dauert also 1778µs und weil ein Paket bei der Übertragung aus 14 Bits besteht braucht die RC dafür 24,89ms. Danach folgt eine Pause von ca. 89ms, das entspricht 50 Bit-Längen. Nach dieser Zeit wird die Bitfolge von meiner RC wiederholt, auch wenn ich nur ganz kurz eine Taste betätige. Das bringt eine erhöhte Datensicherheit, weil der Empfänger die Folge erst akzeptiert, wenn die beiden Pakete identisch sind. Eine ständige Wiederholung erfolgt, wenn ich auf der Taste bleibe.

Woher weiß denn nun das empfangende Gerät, ob ich die Taste nur kurz oder länger gedrückt habe, wurde nun eine 1 gesendet oder 111. Um das zu beantworten betrachten wir zunächst einmal die 14 übertragenen Bits genauer.

*Abbildung 6: Bedeutung der Bits*

Jeder Bit-Train beginnt grundsätzlich mit einer 1. In der Bitfolge sind beim RC5-Protokoll eine 5-Bit lange Geräteadresse und ein 7-Bit langes Befehlswort codiert. Auch das zweite Bit war ursprünglich eine 1. Um mehr Befehle zur Verfügung zu haben, hat man später im zweiten Bit der Folge das negierte Bit 6 des Befehlswortes codiert. Bei weniger als 64 Befehlen bleibt es also bei einer 1.

Das dritte Bit ist das sogenannte Toggle-Bit. Genau daran kann der Empfänger sehen, ob eine Taste gehalten wird, dann ist T bei jeder Folge gleich, wie in Abbildung 7.

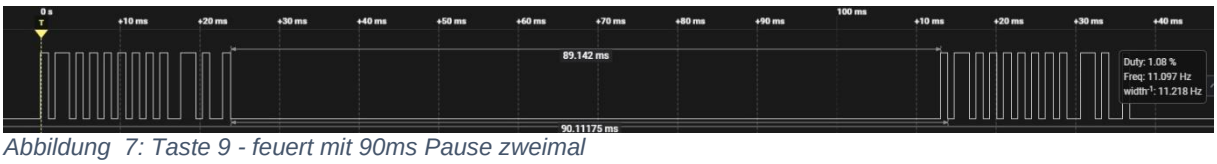

*Abbildung 7: Taste 9 - feuert mit 90ms Pause zweimal*

Wird die Taste zweimal hintereinander gedrückt, dann wechselt das T-Bit seinen Pegel.

*Abbildung 8: Taste 9 - Wiederholte Betätigung*

Besser zu erkennen ist das, wenn man die beiden Plots untereinanderlegt. Die Pulsfolgen sind übrigens mit einem Logic Analyzer in Verbindung mit der kostenlosen Software Logic2 von SALEAE aufgenommen. Wie man mit diesem Tool arbeitet und woher man es bekommt, das verraten die Links (Bernd Albrecht "Logic Analyzer - Teil 1: I2C-Signale sichtbar machen"). Ich hatte den Kanal 1 hier am Collector des Transistors angeschlossen.


*Abbildung 9:; Taste 9 - Vergleich des Pulszuges*

Aus einem 0-Bit im oberen Zug wird ein 1-Bit im unteren. Die wiederholte Folge ist jeweils mit der ersten identisch. Auf der RC habe ich die Taste 9 gedrückt, die als Kommandowort auch eine 9 liefert.

## Der Manchester-Code

Wie entsteht denn jetzt aus der Folge der logischen Bits die Pulsfolge mit der der Sender die IR-LED taktet? Wie Sie in Abbildung 5 sehen, erfolgt der Flankenwechsel stets in der Mitte des logischen Bits. Die erste Hälfte der Bitdauer liegt der Pegel auf dem negierten Bitwert, in der zweiten Hälfte entspricht der Pegel dem tatsächlichen Bitwert.

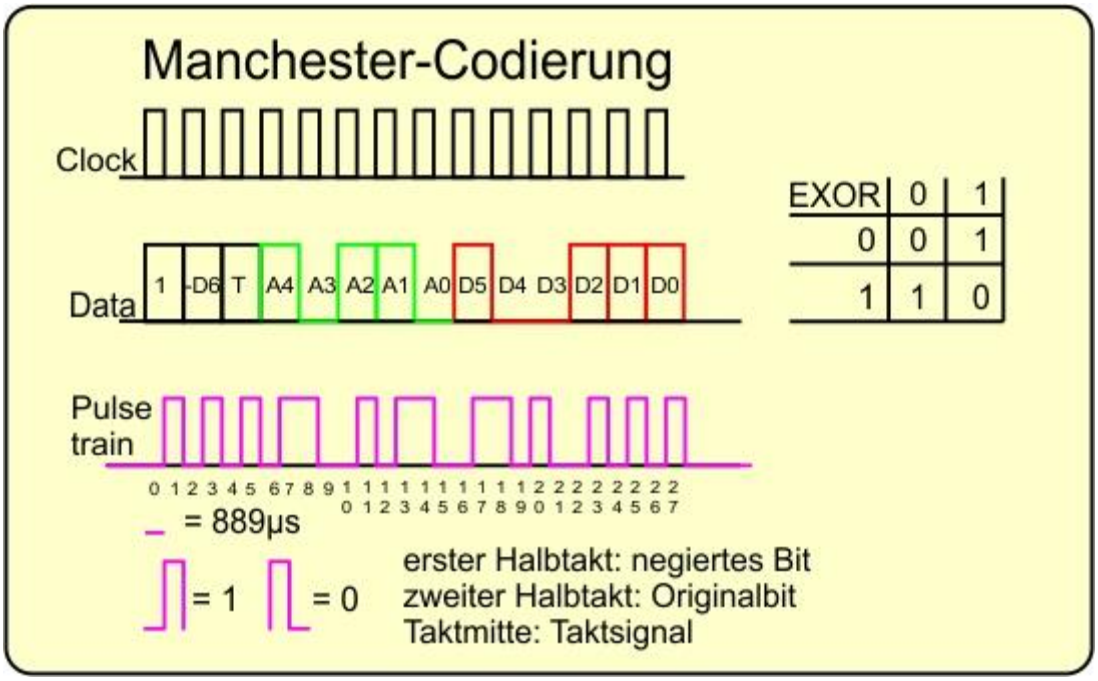


*Abbildung 10: manchester-Codeierung*

Die Frequenz des Bittransfers beträgt 1/1778µs = 562 Hz. Überlagert man die Folge mit einem Takt von der doppelten Frequenz von 1124 Hz und verknüpft die beiden Signale mit einem EXOR-Gatter, dann erhält man die Pulsfolge im Manchester-Code. Während der Pegel dieser Folge auf HIGH liegt, wird die 36kHz-Taktung eingeschaltet, bei LOW wird nix gesendet. Ein Pegelwechsel des Nutzsignals, der eigentlichen Bitfolge (Data), erfolgt genau dann, wenn ein Puls oder eine Pause des **Manchster**-Signals (Puls Train) die Länge von 1778µs hat. Haben Puls und Pause die Länge 889µs bleibt der Pegel des Nutzsignals unverändert.

Weil das erste Bit stets eine 1 ist, sind die Pegel der folgenden Bits eindeutig bestimmt. Das kann man nutzen, um aus Manchester-Code das Nutzsignal zurückzugewinnen. Folgt die zweite Flanke nach dem Start nach 889µs, dann ist das nachfolgende Bit auch eine 1 und so weiter. Der vierte Puls umfasst die Takte 7 und 8. Das bedeutet, dass von A4 zu A3 ein Pegelwechsel stattfinden muss, A3 also LOW oder 0 ist. Auch die nachfolgende Pause ist lang, somit muss erneut ein Pegelwechsel erfolgen, was A2 zu HIGH oder 1 macht. Ein folgender kurzer Puls belässt A1 auch auf 1. Jetzt können Sie sicher den Rest selbst entschlüsseln.

Wenn Sie nicht sicher sind, ob Ihre RC auch richtig funkt(ioniert), dann testen Sie das einfach mit der Handykamera, die kann Infrarotlicht sehen.

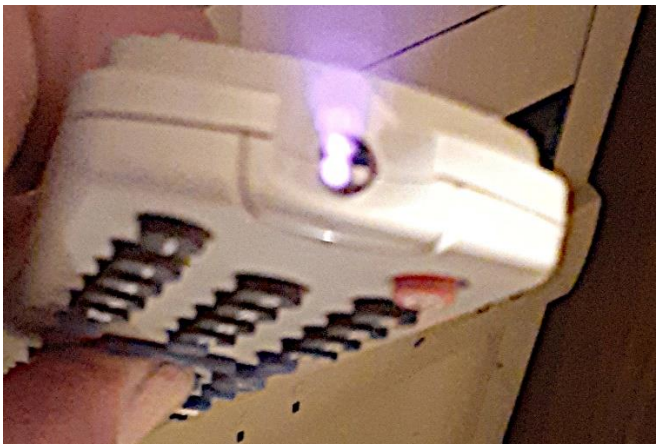

*Abbildung 11: IR-Nachweis*

## 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:

lern.py Hauptprogramm
IR-MicroPython-Paket von Peter Hinch und
das was wir davon brauchen abgespeckte Version

## 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 und seine etwas kryptische Umgebung

Das Paket von Peter Hinch enthält die Module für verschiedene Hersteller und RC-Untertypen und ist grob in Empfänger und Sender eingeteilt. Die Pakete sind so geschnürt, dass nach dem Import des Programms **test.py** aus dem Paket **ir_rx test.py** eine Bedienerinfo in REPL ausgegeben wird.

**>>> from ir_rx.test import test**

Ich habe das Paket für den Zweck dieses Beitrags umgebaut und schlanker gemacht, wodurch es ein wenig überschaubarer wird. Kopieren Sie bitte den Ordner

**ir_rx** nach dem Download und dem Entpacken des Zip-Archivs in Ihr Arbeitsverzeichnis und von dort in den Flash-Speicher des Controllers.

Einige Informationen zum Thema MicroPython-Paket erleichtert das Verständnis des Programmablaufs. Ein Paket oder engl. Package ist ein Verzeichnis, in dem mehrere Module wohnen können. Auch weitere Unterpakete können in diese WG einziehen. Ein Paket bezieht seinen Namen vom Namen des Verzeichnisses. Eine genaue Auswahl der Module ist durch die Punktnotation möglich, wie im obigen Beispiel.

Im Paket kann es eine Datei mit dem Namen **__init__.py** geben. Diese Datei wird immer dann ausgeführt, wenn das Paket oder ein darin befindliches Modul importiert wird. **__init__.py** stellt Initialisierungscode, Basis-Objekte und/oder Basis-Klassen bereit, die von den anderen Modulen des Pakets genutzt werden können. Die enthaltenen Deklarationen werden durch die Schreibweise der Anweisung in den globalen Namensraum eingebunden, so wie es auch bei der Vererbung geschieht. Später importierte Module können somit problemlos darauf zugreifen. Die folgende REPL-Zeile bewirkt die Ausführung der **__init__.py**, die ich um eine print-Anweisung erweitert habe, um die Ausführung der Datei zu belegen.

```
>>> from ir_rx.philips import RC5_IR
__init__.py wurde ausgeführt
```

Die Objekte von **RC5_IR** liegen im globalen Namensraum.

```
>>> from machine import Pin
>>> rc = RC5_IR(Pin(23),lambda _ : 0)
>>> rc
<RC5_IR object at 3ffee3c0>

Reset!
```

Aber auch der Import des gesamten Pakets führt zum Abarbeiten der Datei **__init__.py**.

```
>>> import ir_rx
__init__.py wurde ausgeführt

Reset!
```

```
>>> import ir_rx.acquire
__init__.py wurde ausgeführt
>>> import ir_rx.philips
>>> from machine import Pin
>>> rc=ir_rx.philips.RC5_IR(Pin(23),lambda _ : 0)
```

Diese Zeilen zeigen zweierlei.
**Erstens**: **__init__.py** wird nur beim ersten Import eines Moduls aus dem Paket ausgeführt.
**Zweitens**: Die Klasse **RC5_IR** befindet sich jetzt in einem eigenen Namensraum, der durch die Punktnotation erreicht wird.

Das soll an Informationen zu Paketen genügen und reicht zum Verständnis der Arbeitsweise des Programms aus, das wir nun besprechen. Zum Schluss gibt es noch eine Grafik, die das komplexe Zusammenwirken der Komponenten zeigt.

Wir haben das Hauptprogramm **lern.py**, die **Philips-RC5**-Komponente mit der Klasse **RC5_IR** und die Klasse **IR_RX** in der Datei **__init__.py** die als Schnittstelle zur Hardware dient, also quasi den Treiber für den IR-Empfänger darstellt.

Wir beginnen mit dem Hauptprogramm und da, wie üblich, mit dem Importgeschäft.

```
from sys import platform, exit
from time import sleep
from gc import collect
from machine import Pin, freq
from ir_rx.philips import RC5_IR
```

Mit **platform** findet das Programm heraus zu welcher Familie der Controller gehört, und die Funktion **exit**() ermöglicht einen geordneten Ausstieg aus dem Programm. Dann holen wir aus dem Modul **time** die Funktion **sleep** für Denkpausen. Anfallenden Datenmüll beseitigt **collect()**. Die Klasse Pin brauchen wir für den Eingang an GPIO23 und mit der Funktion **freq**() wird bei einem ESP8266 der Takt vom Defaultwert 80MHz auf 160MHz erhöht. Das Kernstück dieses Projektteils ist der Import der Klasse **RC5_IR**. Aus den Informationen zu Paketen wissen wir, dass neben dem Import der Klasse auch die Datei **__init__.py** gestartet wird. Die Variante mit **from** bindet die Deklarationen, die in der Klasse stattfinden, in den globalen Namensraum ein. Den zeitlichen Ablauf der Aktionen zeigt die Ausgabe in REPL nach dem Start von **lern.py**. Wir kommen noch darauf zurück, wenn wir die anderen Programmteile besprechen.

Nach dem Start von **lern.py** wird folgendes in REPL ausgegeben. Die Reihenfolge der Aktionen ist wichtig für das Zusammenspiel der Komponenten.

**>>> %Run -c $EDITOR_CONTENT**
*__init__.py wurde ausgeführt*
*RC5_IR wurde importiert*
*constructor von RC5_IR; decode <bound_method>*
*Constructor von IR_RX*

```
if platform == "esp8266":
    freq(160000000)
    p = Pin(13, Pin.IN)
elif platform == "esp32":
    p = Pin(23, Pin.IN)
else:
    print("unbekannter Port")
    exit()
```

Jetzt stellen wir den Controllertyp fest und richten den entsprechenden GPIO-Pin als Eingang ein.

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

**keypressed** ist ein Flag, mit dem die Hauptschleife erfährt, dass auf der RC eine Taste gedrückt wurde. **Data**, **Addr** und **Ctrl** nehmen später die Werte für den entsprechenden Teil der IR-Nachricht auf. **rcCode** wird als leere Liste deklariert und nimmt alle Werte auf, die in einem Puls Train übermittelt wurden. Das ist nötig, weil die RC auf einen Tastendruck mindestens zwei Pakete sendet, wir aber nur eines brauchen können.

Die Funktion **cb**() ist die Callback-Funktion, die letztlich gerufen wird, wenn ein Pulse Train Paket eingefahren ist. Hier wird entschieden, was damit passieren soll. Das kann von Anwendung zu Anwendung unterschiedlich sein. Deswegen steht diese Funktion auch im Hauptprogramm, für das der User zuständig ist. Stünde der Code in einem Modul, dann müsste neben dem Hauptprogramm auch dieses Modul geändert und hochgeladen werden.

```
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))
```

Wurde ein gültiges Datenpaket empfangen, dann setzen wir **keypressed** auf True und hängen Befehlscode, Geräteadresse und den Status des Toggle-Bits als Tupel an die Liste **rcCode** an. Auf diese Weise ist sichergestellt, dass alle Pakete aus dem Empfangspuffer entfernt werden und das erste gesichert ist.

Jetzt ist **cb**() definiert und wir können ein RC5-Objekt instanziieren, indem wir dem Konstruktor das GPIO-Objekt **p** und die Callback-Routine **cb** übergeben.

```
ir = RC5_IR(p, cb)
```

Bevor wir die Mainloop betreten, öffnen wir noch die Datei **befehle.cfg** zum Schreiben.

```
f=open("befehle.cfg","w")
```

```
while True:
    t=input("Tasten-Name auf der RC -> ")
    if t=="q":
        f.close()
        ir.close()
        exit()
```

Die Aufforderung, den Namen einer Taste auf der RC einzugeben erscheint in REPL, der Cursor blinkt. Die Eingabe speichern wir in **t**.

War es ein "**q**", dann schließen wir die Datei, schalten die Interruptquellen aus und verlassen das Programm.

Sonst erhalten wir die Aufforderung, jetzt die Taste auf der RC zu betätigen. Die while-Schleife wartet, bis das geschehen ist. Dem Controller gönnen wir noch eine kurze Rast und verlassen die Schleife mit break.

```
    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"
        f.write(line)
        keypressed=False

    gc.collect()
```

Im ersten Listenelement steht jetzt das Tupel mit dem Commando-Wort, der Geräteadresse und dem Toggle-Bit-Status bereit. Dieses Tupel entpacken wir in die entsprechenden drei Variablen und entfernen dann die Elemente aus der Liste **rcCode**.

Wir lassen uns die Werte anzeigen, dann bauen wir die Zeile für die Ausgabe in die Datei zusammen und schreiben sie in die Datei.

Jetzt fragen Sie sicher, was an dem kleinen Prográmmchen, mit grade mal 54 Zeilen, so mystisch ist. Nun das liegt an der Funktion **cb**(), besser gesagt, nicht an der Funktion selbst, sondern an dem Mechanismus ihres Aufrufs, der um einige Ecken geht. Außer dem Hauptprogramm haben wir es ja noch mit zwei weiteren Dateien zu tun, die uns zwei Klassen bescheren.

Da ist die Klasse **RC5_IR** im Modul **philips**, die wir selbst importiert haben und die uns die Methode **decode**() schenkt. Ich möchte hier nur einmal den Namen erwähnt haben, damit Sie wissen, woher der Bezeichner stammt. Besprechen werden wir das Listing später.

Wir fangen jetzt nämlich ganz hinten, im Verborgenen an, mit **__init__.py**. Aber eigentlich ist die Ausführung dieses Programms das, was als Erstes nach dem Programmstart passiert, sehen Sie selbst

**>>> %Run -c $EDITOR_CONTENT**
*__init__.py wurde ausgeführt*

*RC5_IR wurde importiert*
*constructor von RC5_IR decode <bound_method>*
*Constructor von IR_RX*

Bereits mit dem Import der Klasse **RC5_IR** stoßen wir die Ausführung der Datei **__init__.py** an und damit stellen wir den Programmtext der Klasse **IR_RX** bereit. Von dieser wird aber keine Instanz erzeugt, vielmehr erbt **RC5_IR** von **IR_RX** und das bedeutet, dass deren Bezeichner von **RC5_IR** aus so referenzierbar sind, als wären sie in **RC5_IR** selbst deklariert worden. Das ist der Hintergedanke, die Philosophie, von Peter Hinch, eine spezielle Klasse für ein spezielles RC-Protokoll mit den grundlegenden Feld-Wald-Wiesen-Eigenschaften auszustatten, die für jede RC-Anwendung gebraucht werden. Natürlich könnte man die Basiseigenschaften auch in **RC5_IR**, **RC6_IR**, **NEC8**, **NEC16** und weiteren Klassen einfügen, dann müsste man Änderungen im Basisbereich aber auch in all diesen Klassen durchführen. Das ist unnötige Arbeit und Zeitverschwendung und erzeugt Redundanzen. So genügt es, wenn man die Änderungen nur in **IR_RX** durchführt, das gilt dann für alle speziellen Klassen von RC-Typen. Außerdem hält das Vorgehen den Speicherbedarf niedrig, weil nur das importiert werden muss, was auch gebraucht wird. Das wird noch deutlicher, wenn Sie die Original Dateien von Peter studieren.

**RC5_IR** ist also speziell auf das **Decodieren** eines RC5-Signals zugeschnitten. Die Hauptaufgabe der Klasse **IR_RX** dagegen ist das Erfassen der **zeitlichen Abstände** zwischen den Flanken des Signals, das vom IR-Empfangsmodul, negiert über den Transistor, zum Eingangs-Pin **GPIO23** gelangt. Diese Funktionen braucht man bei jeder RC. Die Empfängerdiode sieht das Signal so, wie es der Sender, also die RC, wegschickt. Der Chip dahinter filtert den 36-kHz-Träger heraus und gibt eine saubere Pulsfolge an seinem Ausgang **S** ab, leider in negierter Form. Der Transistor invertiert das Signal, sodass es vom Decoder in der Klasse **RC5_IR** auch verstanden wird.

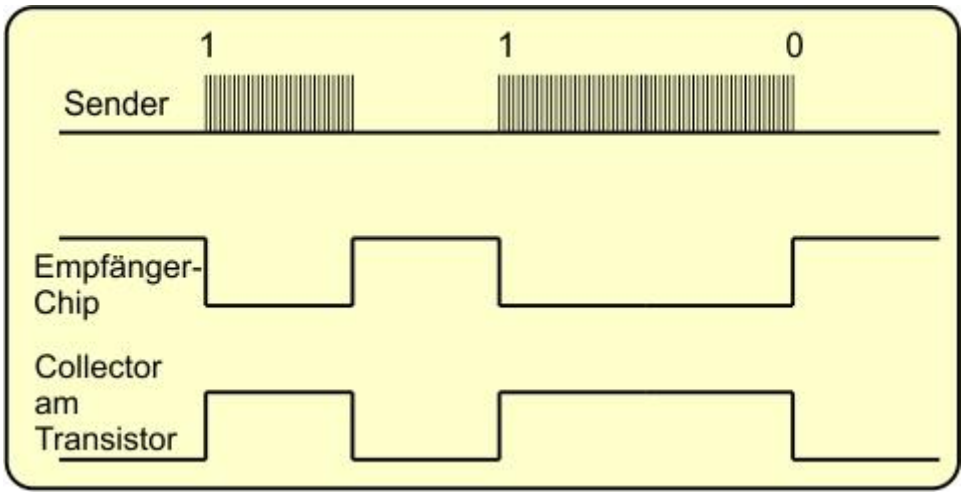

*Abbildung 12: Vom Sender zum ESP32-Eingang*

## __init__.py:

```
from machine import Timer, Pin
from array import array
from utime import ticks_us
```

Wir brauchen einen Hardware-Timer und ein Pin-Objekt, also importieren wir die entsprechenden Klassen vom Modul **machine**. Die Zeitintervalle landen in einem Array für 64-Bit-Zahlen. Die Auflösung der Zeiterfassung erfolgt in Microsekunden.

Dann deklarieren wir die Basisklasse **IR_RX** und einige Klassen-Attribute.

```
class IR_RX():
    # Result/error codes
    # Repeat button code
    REPEAT = -1
    # Error codes
    BADSTART = -2
    BADBLOCK = -3
    BADREP = -4
    OVERRUN = -5
    BADDATA = -6
    BADADDR = -7
```

Es folgt der die Deklaration des Konstruktors __**init**__().

```
    def __init__(self, pin, nedges, tblock, callback, *args):
        # Optional args for callback
        self._pin = pin
        self._nedges = nedges
        self._tblock = tblock
        self.callback = callback
        self.args = args
        self._errf = lambda _ : None
        self.verbose = False

        self._times = array('i',  (0 for _ in range(nedges +
1)))  # +1 for overrun
        pin.irq(handler = self._cb_pin, trigger =
(Pin.IRQ_FALLING | Pin.IRQ_RISING))
        self.edge = 0
        self.tim = Timer(0)
        self.cbck = self.decode
        print("Constructor von IR_RX")
```

Ein RC5_IR-Objekt muss wissen an welchem Pin das IR-Signal hereinkommt, wie viele Flanken das Signal maximal aufweist, wie viele Millisekunden der Pulse-Train dauert und welche Funktion aufgerufen werden soll, wenn er zu Ende ist. All diese Informationen werden auf Instanz-Attribute übertragen. Eine Funktion **.errf()** wird leer vordefiniert, **.verbose** liefert Ausgaben in REPL, wenn wir **True** zuweisen.

Das Array **._times** besteht aus **.nedges** +1 Zellen, die vorzeichenbehaftete ('i') Ganzzahlen aufnehmen können.

Für das Pin-Objekt **.pin** wird der IRQ-Handler **.cb_pin**() festgelegt und für steigende und fallende Flanken scharfgemacht.

Der Index in das Array **._times** ist **.edge**, er wird mit 0 initiiert. Als Timer verwenden wir den Hardwaretimer Timer(0).

Schließlich weisen wir **.cbck** die Methode **RC5_IR.decode** zu. Das funktioniert deswegen, weil zum Zeitpunkt, an dem wir den Konstruktor von **IR_RX()** in **RC5_IR** aufrufen, der Konstruktor von **RC5_IR** die Methode **decode**() bereits angelegt hat und deren Bezeichner somit bekannt ist. Wir werden auf den Sachverhalt bei der Besprechung der Klasse **RC5_IR** noch einmal zurückkommen. Den kryptischen Zusammenhang werden Sie verstehen, wenn wir mit der Besprechung am Ende sind. Entstanden ist die Mystik durch die Philosophie von Peter Hinch, dass die Decoder-Module einfach austauschbar sein sollen und Daten zwischen den verschiedenen Ebenen transportiert werden müssen. Vertrauen Sie mir einfach, es wird am Ende alles perfekt laufen. Nach dem Start von **lern.py** wird folgendes in REPL ausgegeben. Daran sehen Sie noch einmal die Reihenfolge der Aktionen

**>>> %Run -c $EDITOR_CONTENT**
*__init__.py wurde ausgeführt*
*RC5_IR wurde importiert*
*constructor von RC5_IR; decode <bound_method>*
*Constructor von IR_RX*

```
    def _cb_pin(self, line):
        t = ticks_us()
        if self.edge <= self._nedges:  # Allow 1 extra pulse
            if not self.edge:  # First edge received
                self.tim.init(period=self._tblock ,\
                      mode=Timer.ONE_SHOT, callback=self.cbck)
            self._times[self.edge] = t
            self.edge += 1
```

Das ist die Methode **_cb_pin**(). Sie ist der IRQ-Handler des Pin-Objekts und wird gerufen, wenn sich der Pegel an GPIO23 ändert. Als IRQ-Handler muss die Routine aus verschiedenen Gründen so kurz wie möglich gehalten werden. Daher werden nur die Zeiten festgehalten und die umfangreichere Decodierung erst dann durchgeführt, wenn das gesamte Paket gelesen wurde.

Wir merken uns die Zeit in **t** und prüfen, ob der Index **.edge** im gültigen Bereich liegt. Wenn **.edge** den Startwert 0 hat, ist **not .edge** wahr, und wir starten den Timer(0) für einen einmaligen Schuss. Nach dem Ablauf von **tblock** Millisekunden wird die Funktion gerufen, deren Referenz wir in **.cbck** abgelegt haben, **.decode**().

Der Zeitwert wandert ins Array, und wir erhöhen den Index.

```
    def do_callback(self, cmd, addr, ext, thresh=0):
        self.edge = 0
        if cmd >= thresh:
            self.callback(cmd, addr, ext, *self.args)
        else:
```

```
            self._errf(cmd)
```

Die Methode **do_callback**() liegt in der Rufkette des Timers. Bei der Initialisierung erhält **.callback** die Referenz auf die Funktion **cb()** aus dem Hauptprogramm. Dadurch kann **do_callback**() von der untersten Ebene auf die höchste Ebene zugreifen.

Bei abgelaufenem Timer muss der Index **.edge** wieder auf 0 gesetzt werden. Falls der Kommando-Code aus dem IR-Signal mindestens den Wert von **thresh** hat, wird **cb**() via **.callback**() gerufen, andernfalls liegt ein Fehler vor, der durch die Funktion **._errf**() gemeldet werden kann, wenn dafür eine Funktion angegeben wurde, was ein Aufruf der nächsten Methode erledigen kann.

```
    def error_function(self, func):
        self._errf = func
```

```
    def close(self):
        self._pin.irq(handler = None)
        self.tim.deinit()

print ("__init__.py wurde ausgeführt")
```

Vor dem Programmende müssen die noch aktiven IRQ-Quellen ausgeschaltet werden, weil sich der ESP32 sonst merkwürdig verhält. **Close**() erledigt das. Abschließend erhalten wir eine Meldung, dass **__init__.py** ausgeführt wurde.

Jetzt fehlt nur noch die Klasse **RC5_IR**. Die einzige Funktion, die sie zur Verfügung stellt, ermittelt aus den Flankenabständen in **._times** die Bitwerte des Signals. Weil die Klasse von **IR_RX** erbt, hat **decode**() Zugriff auf alle in **IR_RX** deklarierten Objekte, speziell auch auf die Instanz-Attribute, also auch auf das Array **._times**.

Zur Berechnung von Zeitdifferenzen importieren wir die Funktion **ticks_diff**(). Dann holen wir die Basis-Klasse **IR_RX** ins Boot.

```
from time import ticks_diff
from ir_rx import IR_RX

class RC5_IR(IR_RX):
    def __init__(self, pin, callback, *args):
        # Block lasts <= 30ms and has <= 28 edges
        print("constructor von RC5_IR; decode", self.decode)
        super().__init__(pin, 28, 30, callback, *args)
```

Bei der Deklarierung der Klasse **RC5_IR** geben wir an, dass wir von **IR_RX** erben wollen. Der Konstruktor fordert das GPIO-Pin-Objekt und die Callback-Funktion für den Timer an. Ein Pulsblock darf höchstens 30 ms dauern, tatsächlich sind es 25 ms. Darin treten bis zu 28 Flanken auf.

Der print-Befehl informiert uns darüber, dass der Konstruktor aufgerufen wurde und dass jetzt die Funktion **decode**() bekannt ist.

Erst jetzt ruft **super().__init__**() den Konstruktor von **IR_RX** auf und gibt das Pin-Objekt, die Flankenanzahl, die Blockdauer und die Callback-Funktion an diesen weiter. Jetzt weiß **IR_RX._cb_pin**() an welchem Anschluss gehorcht werden muss und **IR_RX.do_callback**() weiß, welche Funktion im Hauptprogramm eine Ahnung davon hat, was mit den empfangenen Daten passieren soll.

Wie **._cb_pin**() ist **decode**() ein Interrupt-Handler. Weil mehrere GPIO-Pins IRQ-fähig sind, kann es für die Handler-Routine nützlich sein, zu erfahren, welcher Anschluss den IRQ ausgelöst hat. deswegen hat die Zeile

```
def _cb_pin(self, line):
```

den Parameter **line** in der Parameterliste, in welchem eine Referenz auf das Pinobjekt hereinkommt. Analog sieht es bei den Timern aus. Auch hier wird gemeldet, welcher Timer gefeuert hat. Weil wir aber an der Nummer gar nicht interessiert sind, steht hier nur der Unterstrich als Dummy-Variable. Natürlich könnten wir aus demselben Grund **line** auch durch den Unterstrich ersetzen. Wird aber kein Parameter angegeben, bekommen wir eine Fehlermeldung.

```
    def decode(self, _ ):
        try:
            nedges = self.edge  # No. of edges detected
            if not 14 <= nedges <= 28:
                raise RuntimeError\
                (self.OVERRUN if nedges>28 else self.BADSTART)
            # Regenerate bitstream
            bits = 1
            bit = 1
            v = 1  # 14 bit bitstream, MSB always 1
            x = 0
```

Die Decodierung läuft in einem **try – except**- Konstrukt. Wir schaufeln **.edge** nach **nedges** um. Die lokale Variable **nedges** hat übrigens nichts mit dem Parameter **nedges** in der Parameterliste des Konstruktors **IR_RX**() zu tun.

Liegt die tatsächliche Flankenanzahl zwischen 14 und 28 inclusive der Grenzen, dann handelt es sich um ein gültiges Paket. Sonst gibt es eine Fehlermeldung.

Erinnern Sie sich?

- Das erste Bit eines Pakets ist stets eine 1
- Bei langen Intervallen hat das folgende Bit den negierten Wert des aktuellen bit-Zustands.

**bits** ist der Bit-Zähler, **bit** der momentane Zustand, **v** der Wert des IR-Worts und **x** der Index in das Array **._times**.

Eine BADBLOCK Exception kann hier, laut Peter, nur beim ESP8266 auftreten, weil der IRQ lange Verzögerungen zwischen dem Auftreten der Flanke und der Ausführung des Handlers aufweist.

```
            while bits < 14:
                if x > nedges - 2:
                    print('Bad block 1 edges', nedges, 'x', x)
                    raise RuntimeError(self.BADBLOCK)
```

Wir wissen, dass ein Intervall zwischen zwei Flanken nominell 889µs beziehungsweise 1778µs betragen kann. Real werden die Zeiten aber davon abweichen, sie sollten dennoch beim RC5-Protokoll nicht kürzer als 500µs und nicht länger als 2100µs sein. **x** ist im Moment 0 wir berechnen also für **width** den Versatz zwischen der ersten und zweiten Flanke und prüfen nach, ob der Wert im gültigen Bereich liegt.

```
                # width is 889/1778 nominal
                width = ticks_diff\
                        (self._times[x + 1], self._times[x])
                if not 500 < width < 2100:
                    self.verbose and \
                    print('Bad block 3 Width', width, 'x', x)
                    raise RuntimeError(self.BADBLOCK)
```

Interessant ist die elegante Einbindung der Debug-Meldung. Man könnte dafür natürlich auch eine if-Konstruktion verwenden. Wie funktioniert das?

**>>> width=400; x=1; verbose=False**
**>>> verbose and print('Bad block 3 Width', width, 'x', x)**
*False*
**>>> width=400; x=1; verbose=True**
**>>> verbose and print('Bad block 3 Width', width, 'x', x)**
*Bad block 3 Width 400 x 1*

MicroPython wertet Ausdrücke von links nach rechts aus. Die Auswertung bei der Verknüpfung mit **and** wird abgebrochen, sobald einer der beteiligten Teilausdrücke um das **and** herum **False** wird. Der **Wert dieses Ausdrucks** wird zurückgegeben. Dieses Verhalten ist auch in LUA etabliert. Was in MicroPython als False gilt unterscheidet sich allerding von dem in LUA. Dort werden nur die Terme false und nil zu false ausgewertet. In MicroPython werden folgende Terme als False ausgewertet: 0, False, None, leere Listen [], Dictionarys {} und Tupel(()). Alles andere wird als **True** evaluiert. Die folgenden Beispiele sprechen für sich.

**>>> 0 and 4**
*0*
**>>> 3 and 5**
*5*
**>>> 7 and 0**
*0*
**>>> 1 and 4 and (4 < 7)**
*True*
Dabei ist *True* nicht der boolsche Wert des Gesamtausdrucks sondern nur der der letzten Klammer.

**>>> 1 and 4 and (4 > 7)**

*False*
**>>> 1 and None**
**>>> 1 and ""**
*''*
**>>> 1 and "" and 4**
*''*
**>>> r=[]**
**>>> 1 and r and 5**
*[]*
**>>>**
**>>> r={}**
**>>> 1 and r and 5**
*{}*

Für das Verständnis ist wichtig, dass bei 1 and 7 nicht ein boolscher Wert, True oder False, herauskommt sondern eben 7, was letztlich in if-Konstrukten auch als **True** gewertet wird, weil 7 eben nicht gleich 0 ist.

**>>> 1 and 7**
*7*
**>>> if 1 and 7: print("Test")**

*Test*

Jetzt haben Sie wieder ein Stück MicroPython-Mystik kennengelernt, aber zurück zum Programm. **short** wird wahr, wenn **width** < 1334 ist, das ist der Mittelwert von 889 und 1778. Ein kurzes Intervall konserviert den Bitstatus, während bei einem langen Intervall der Bitstatus wechselt, aus 1 wird 0, aus 0 eine 1. Diesen Wechsel realisieren wir, indem wir **bit** mit 1 exoderieren, Langform: **bit = bit ^ 1**. Die Wahrheitstabelle des EXOR-Operators finden Sie in Abbildung 10.

```
short = width < 1334
if not short:
    bit ^= 1
v <<= 1
v |= bit
bits += 1
x += 1 + int(short)
```

Wir schieben nun das bisherige Ergebnis um eine Position logisch nach links und oderieren dann mit dem Bitstatus – Langform: v = v<<1 und v = v | bit. Der Bitzähler wird inkrementiert und der nächste Index in das Array berechnet. Der wächst bei langen Intervallen um 1, bei kurzen um 2. **int(short)** wird 0, wenn **short** den Wert **False** hat und 1, falls **short True** ist. Nachdem **bits** den Wert 14 erreicht hat, wird die while-Schleife beendet.

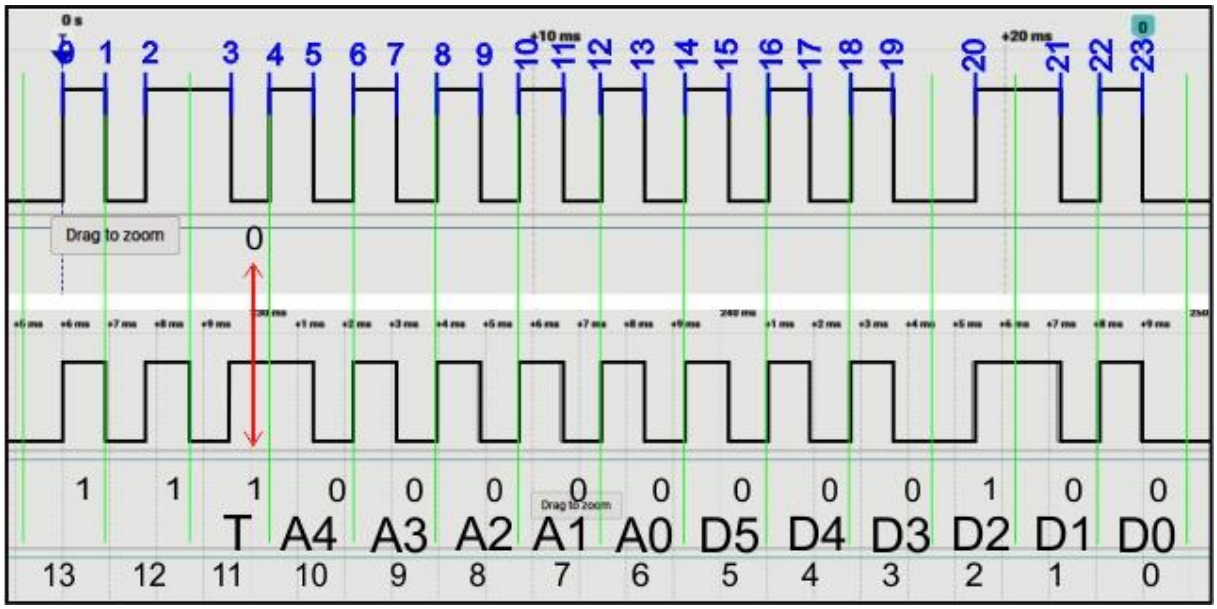


*Abbildung 13: Pulse Train decodieren*

Anhand der Abbildungen 13 und 14 können Sie den Decodier-Algorithmus nachvollziehen und ebenso die Aufspaltung des IR-Worts. Die Taktflanken sind in blau durchnummeriert, die Bitpositionen am unteren Rand in schwarz.

| | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| 1 | **Bits** | **X+1** | **X** | **bit** | **short** | **int(short)** | **x+1+int(short)** | **v** |
| 2 | 1 | | | | | | | 1 |
| 3 | 1 | 1 | 0 | 1 | T | 1 | 2 | 11 |
| 4 | 2 | 3 | 2 | 1 > 0 | F | 0 | 3 | 110 |
| 5 | 3 | 4 | 3 | 0 | T | 1 | 5 | 1100 |
| 6 | 4 | 6 | 5 | 0 | T | 1 | 7 | 11000 |
| 7 | 5 | 8 | 7 | 0 | T | 1 | 9 | 110000 |
| 8 | 6 | 10 | 9 | 0 | T | 1 | 11 | 1100000 |
| 9 | 7 | 12 | 11 | 0 | T | 1 | 13 | 11000000 |
| 10 | 8 | 14 | 13 | 0 | T | 1 | 15 | 110000000 |
| 11 | 9 | 16 | 15 | 0 | T | 1 | 17 | 1100000000 |
| 12 | 10 | 18 | 17 | 0 | T | 1 | 19 | 11000000000 |
| 13 | 11 | 20 | 19 | 0 > 1 | F | 0 | 20 | 110000000001 |
| 14 | 12 | 21 | 20 | 1 > 0 | F | 0 | 21 | 1100000000010 |
| 15 | 13 | 22 | 21 | 0 | T | 1 | 23 | 11000000000100 |
| 16 | 14 | | | | | | | |

*Abbildung 14: Decodiertabelle halbautomatisch*

```
            self.verbose and print(bin(v))
            # Split into fields (val, addr, ctrl)
            val =(v & 0x3f) | (0 if ((v >> 12) & 1) else 0x40)
            addr = (v >> 6) & 0x1f
            ctrl = (v >> 11) & 1
```

Wenn das Debugging eingeschaltet ist, kriegen wir das Ergebnis in binärer Schreibweise angezeigt. Dann teilen wir das IR-Wort in seine Anteile auf. **val** enthält das Kommandowort von Bitposition 5 herunter bis 0. Das siebte Bit steht an Position 12. Wir schieben es an Position 0 undieren mit 1 und stellen so fest, ob es gesetzt ist. Ist das der Fall, oderieren wir den bisherigen Wert von **val** mit 0 sonst mit 0x40. Das Bit 12 in **v** ist ja negiert.

Die Adresse finden wir von Position 10 bis 6. Zum Normieren schieben wir die maskierten Bits um sechs Positionen nach rechts und maskieren mit 0x1f = 0b00011111.

Das Toggle-Bit erhalten wir aus Position 11. Wir schieben um 11 Positionen nach rechts und maskieren mit 1.

Tritt im **try**-Block ein Fehler auf, dann fangen wir diesen durch den **except**-Block ab. **val** erhält dann einen der negativen Werte, die in **IR_RX** als Klassen-Attribute deklariert sind.

```
        except RuntimeError as e:
            val, addr, ctrl = e.args[0], 0, 0
        # Set up for new data burst and run user callback
        self.do_callback(val, addr, ctrl)
```

die ermittelten Werte gehen dann an die Funktion **.do_callback**(), die sie an die Funktion **cb**() weiterreicht.

Sobald die Datei **philips.py** abgearbeitet ist, erhalten wir darüber eine Meldung in REPL.

```
print("RC5_IR wurde importiert")
```

Damit ist der Kreis geschlossen und der Rundgang beendet. In Abbildung 15 können Sie noch die ganze Mystik des Programms inhalieren.

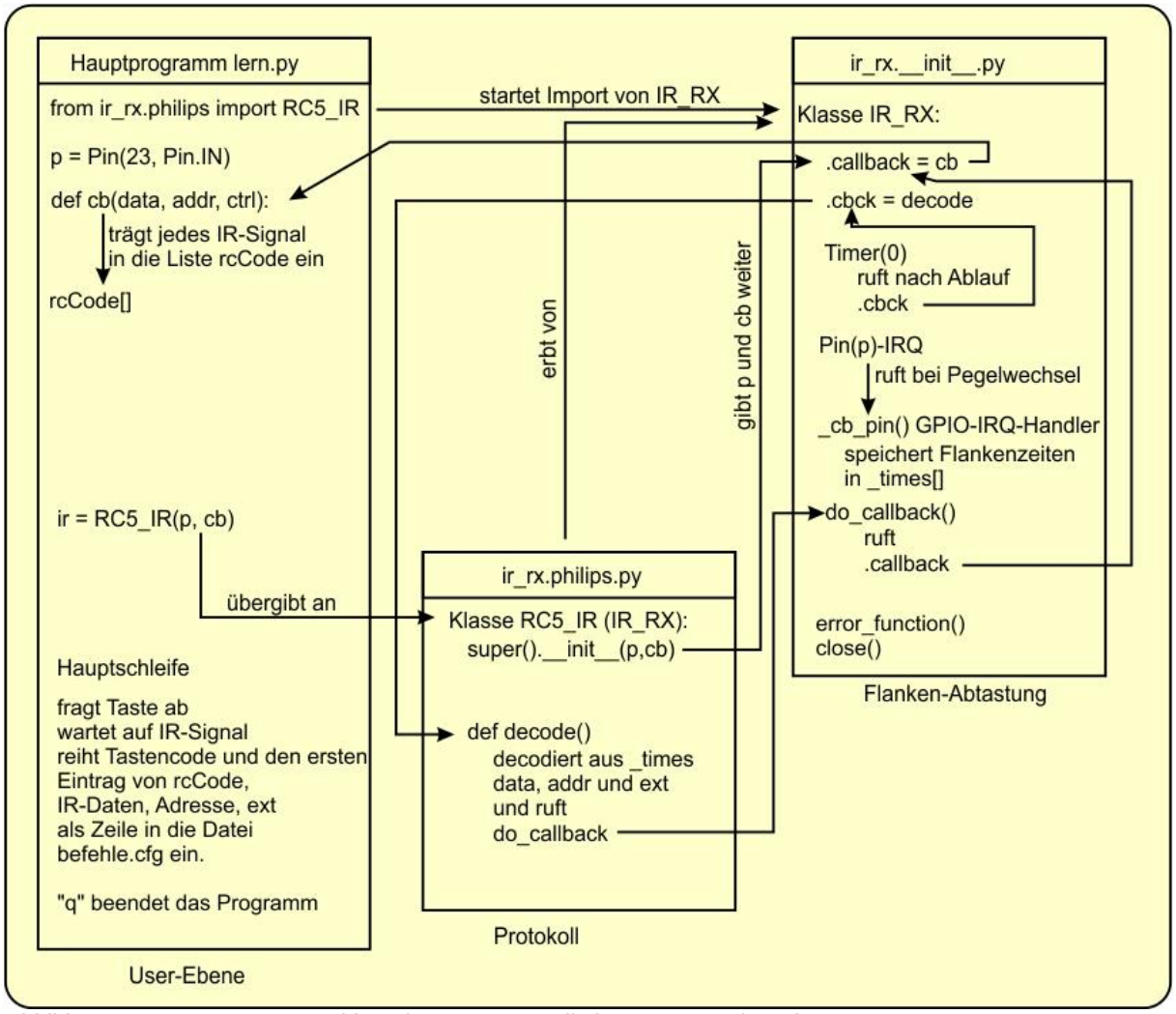


*Abbildung 15: Das Zusammenwirken der Programmteile ist etwas verwirrend*

Im nächsten Beitrag verwandeln wir dann den ESP32 in einen IR-Sender. Bleiben Sie dran!

Halt – ein Mirakel habe ich noch zu bieten. Die folgende Zeile schmeißt das ganze Programm und liefert eine Riesenmenge an Fehlermeldungen. Dabei ist es doch nur eine Eingabeaufforderung mit eine Prompt-Zeile.

```
>>> t=input("taste an der RC >>> ")
```

Es hat enorm lang gedauert, bis ich die Ursache dafür gefunden habe, denn ursprünglich war diese Zeile in eine while-Schleife integriert. Ich musste also herausfinden, welche Anweisung den Programmabsturz bewirkt hat.

Schließlich habe ich den Prompt-String und da speziell diesen Teil ">>>" dafür herausgefiltert. Die drei Größer-Zeichen sind ja auch der normale Prompt von REPL. Warum ein Teil meines Strings den Ablauf des Interpreters so total durcheinanderbringt, konnte ich leider nicht ergründen.