

RC-5 und mehr

Eine praktische Einführung
in die Welt der Fernbedienungen

mit Micropython und dem
ESP32-Board TTGO T-Display



von

G. Heinrichs

Mönchengladbach, 16.11.2024

RC-5 und mehr

Einführung

Mit **RC-5** bezeichnet man ein Datenübertragungsprotokoll mit **Infrarot-Licht**, welches u. A. bei Fernsteuerungen von Unterhaltungselektronik wie Fernseher und Stereoanlagen benutzt wird. Entwickelt wurde es von der Firma Philips.

Die Daten werden seriell über ein Rechteck-Signal mit der Frequenz 36 kHz (manchmal auch 38 kHz) übertragen; dazu wird dieses Signal durch einen Manchester-Code moduliert.

In diesem Beitrag werden wir zeigen, wie Daten mit solchen Signalen übertragen werden können. Dabei werden wir zunächst zeigen, wie man Bytes senden und empfangen kann; damit können wir z. B. auch Texte übertragen. Die entsprechenden Programme werden wir dabei aber so konzipieren, dass eine Übertragung auf andere Datenstrukturen (insbesondere auch das RC-5-Protokoll für Fernbedienungen) recht einfach ist.

In einem älteren Beitrag hatte ich schon einmal Programme für RC-5 vorgestellt; hierbei hatte ich aber auf die in BASCOM schon vorhandenen Empfangs- und Sende-Befehle zurückgegriffen. Hier wollen wir nun den Manchester-Code und das RC-5-Protokoll etwas genauer studieren...

1 Der Manchester-Code

In der folgenden Abbildung 1 sehen wir das Diagramm für ein typisches Manchester-Code-Signal: Hier wurden mit den drei Bytes die ASCII-Codes für die Zeichen 'H', 'a' und 'l' übertragen; dabei ist in diesem Diagramm nur das niederfrequente Modulationssignal zu sehen. Zunächst werden wir uns nur um dieses niederfrequente Signal kümmern; erst in dem Abschnitt 4 werden wir das tatsächlich ausgesandte HF-Signal betrachten.

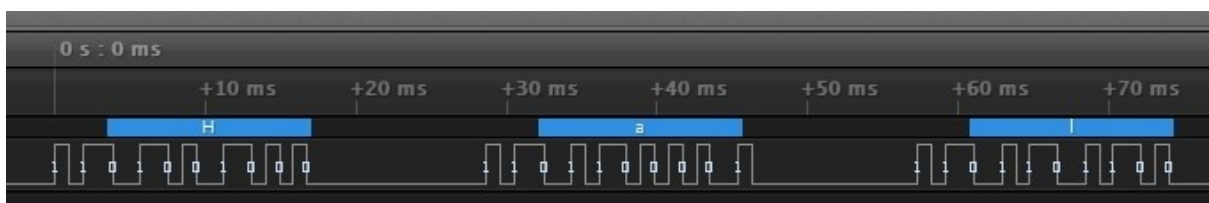


Abb. 1

Das Diagramm in Abb. 1 wurde mit dem Logic Analyzer *Saleae Logic* aufgezeichnet. Dieses Programm kann Manchester-Code-Signale dekodieren. Die Nullen und Einsen befinden sich hier ganz bewusst **auf** den Flanken: Der Manchester-Code ist nämlich Flanken-basiert; eine steigende Flanke steht für eine 1, eine fallende Flanke für 0 (vgl. Abb. 2). Bei RC-5 ist das Bit-Intervall 1,778 μ s lang. Die Flanke, welche den Bitwert festlegt, liegt genau in der Mitte dieses Intervalls.

Die Übertragung beginnt immer mit einer **Präambel**, welche aus zwei 1-Bits besteht. Diese beiden Bits dienen zur **Synchronisierung**: Im Ruhezustand liegt immer ein 0-Signal vor. Der Empfänger wartet nun auf die erste positive Flanke. Registriert er diese, so misst er die Zeitdauer T , die nun verstreicht, bis das Signal wieder 0 wird. Dies geschieht am Ende des ersten 1-Bits; danach sorgt das zweite 1-Bit der Präambel ja zunächst wieder für ein 0-Signal. Für den Empfänger ergibt sich daraus zweierlei:

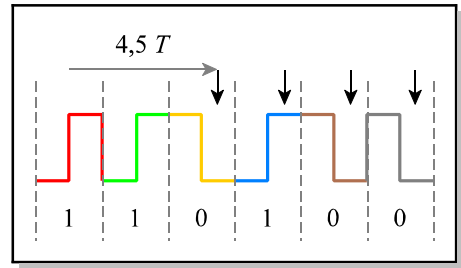


Abb. 2

1. Er kennt nun die Dauer eines Bit-Intervalls: Sie beträgt gerade $2T$. Beim Manchester-Code muss dem Empfänger also gar nicht die Bit-Dauer ($2 \cdot T$) des Senders mitgeteilt werden; er kann sie mit Hilfe der Präambel selbst bestimmen. (Von dieser Möglichkeit werden wir hier aber zunächst keinen Gebrauch machen, sondern nur mit dem oben angegebenen Bit-Intervall von $1,778 \mu\text{s}$ arbeiten.)
2. Das Signal für den Payload (Nutzlast, in unserem Fall ein einziges Byte) beginnt nach einer Zeitspanne von $3T$ nach dem Registrieren des ersten 1-Signals. Die jeweiligen Bitwerte des Payloads ergeben sich für den Empfänger dann einfach aus dem jeweiligen Signalzustand bei den in Abb. 2 mit einem senkrechten Pfeil markierten Zeitpunkten. Diese befinden sich gerade bei $4,5T$; $6,5T$; $8,5T$... nach dem Registrieren der ersten steigenden Flanke (der Präambel).

2 Micropython-Programm für einen Sender

Für eine Zeichenkette soll der entsprechenden Manchestercode über eine rote LED ausgesendet werden. Die LED ist an Pin 25 angeschlossen.

Die Zeichenkette wird im Terminal über die `input`-Funktion entgegengenommen. Diese Zeichenkette wird nun Zeichen für Zeichen übertragen. Dazu werden die ASCII-Codes der einzelnen Buchstaben ermittelt und diese mit der Funktion `to_bitlist` jeweils durch eine Liste von Bits binär dargestellt. Aus dem Buchstaben 'H' mit dem ASCII-Code 72 wird zum Beispiel die Liste `[0, 1, 0, 0, 1, 0, 0, 0]`, und die Liste für die Präambel lautet einfach `[1, 1]`. Nun müssen diese Listen nur noch mit der Funktion `send_bitlist` in ein entsprechendes Signal (für Pin 25) umgesetzt werden; dies geschieht, indem der Reihe nach für jedes Listenelement das zugehörige Signal mit der Funktion `send_symbol` an Pin 25 ausgegeben wird.

Dieser Weg über die Listen liefert vielleicht nicht den schnellsten und elegantesten Programm-Code; aber er hat einige didaktischen Vorteile: So können damit recht einfach und übersichtlich auch andere, komplexere Datenstrukturen (z. B. die von RC-5-Fernbedienungen benutzte Struktur "2 Start-Bits als Präambel + 1 Toggle-Bit + 5 Adress-Bits + 6 Befehlsbits") übertragen werden. Zudem können Zwischenergebnisse (in Form der Listen) leicht mit den von einem Digital Analyzer angezeigten Signalen verglichen und damit die Suche nach möglichen Fehlern erleichtert werden.

Das zugehörige Programm `RC5_send_1h.py` sieht nun so aus:

```

# RC5_send_1h.py: eine Zeichenkette mit LED im Manchestercode senden
# mit Präambel (zwei 1-Bits) und
# Payload (1 Byte bzw. 8 Bits, 0/1 mit neg./pos. Flanke)
# Halbe Bit-Dauer: real: ca. T = 900 us)

from time import sleep, sleep_us, sleep_ms
from machine import Pin
led = Pin(25, Pin.OUT)
led.value(0) # mit Low-Signal starten

T = 890 # Halbe Bitzeit in us (ergibt ca. 900 us real)

def send_symbol(b): # sendet Signal für 1 Bit b; b ist 1 oder 0
    led.value(b == 0)
    sleep_us(T)
    led.value(b == 1)
    sleep_us(T)

def to_bitlist(anzahl_bits, n): # anzahl_bits: Anzahl der Bits (bei der
                                # Übertragung eines Bytes: 8); n: Zahl
                                # (int), z. B. ASCII-Code eines Zeichens
    bl = []
    for i in range(anzahl_bits):
        j = anzahl_bits - 1 - i
        if n & (1 << j) : # n & (1 << j) > 0 => True
            bl.append(1)
        else:
            bl.append(0)
    # print(bl) # zum Testen
    return bl

def send_bitlist(b_l): # sendet die Signalfolge für die Bit-Liste bl
    for i in b_l:
        send_symbol(i)

# Eingabe der Zeichenkette:
zk = input('Text: ') # liefert Warnung bei non-ASCII-Zeichen
# zk = zk + '\n' # Steuerzeichen für new line anhängen
print('sende: ', zk)
# sleep(3) # spätestens jetzt Digital Analyser/Empfangsprogramm starten

# Zeichenkette senden...
for c in zk:
    pre_list = [1, 1]
    ascii_code = ord(c)
    bit_list = to_bitlist(8, ascii_code)
    # Präambel senden:
    send_bitlist(pre_list)
    # Bitliste für Zeichen senden:
    send_bitlist(bit_list)
    led.value(0) # mit Low-Signal beenden
    sleep_ms(40)

```

3 Sende-Programm mit dem Saleae Digital Analyzer testen

Wir schließen zunächst den Datenausgang Pin 25 an den Kanal 0 des Analyzers (CH 1 bei dem in Abb. 3 gezeigten Analyzer) und die Masse an GND an. Wenn Sie zu Testzwecken an Pin 25 auch eine LED ohne Vorwiderstand angeschlossen haben, sollten Sie diese entfernen.

Öffnen Sie nun das Analyzer-Programm. Über die Pfeiltasten (links neben dem grünen Start-Button) stellen Sie zunächst ein:

- Speed: 8 MS/s
- Duration: 200 ms



Abb. 3

Unter dem Start-Button stellen Sie die Triggerung auf ansteigende Flanke ein. Anschließend müssen wir noch den Analyzer für den Manchester-Code aktivieren und konfigurieren. Dazu betätigen wir am rechten Rand des Programm-Fensters das Plus-Symbol in der Analyzers-Zeile und klicken dann auf die Option "Show more analyzers". Es erscheint nun darunter eine Liste mit Codes; daraus wählen Sie nun "Manchester" aus. Jetzt erscheint ein Fenster, in welchem Sie die Parameter unseres Manchester-Codes wie in Abb. 4 dargestellt eingeben müssen. Schließen Sie Ihre Eingaben mit dem Save-Button ab. Der Manchester-Code wird jetzt als Protokoll aufgeführt; er kann bei Bedarf mit Hilfe des Zahnrad-Symbols neu konfiguriert werden.

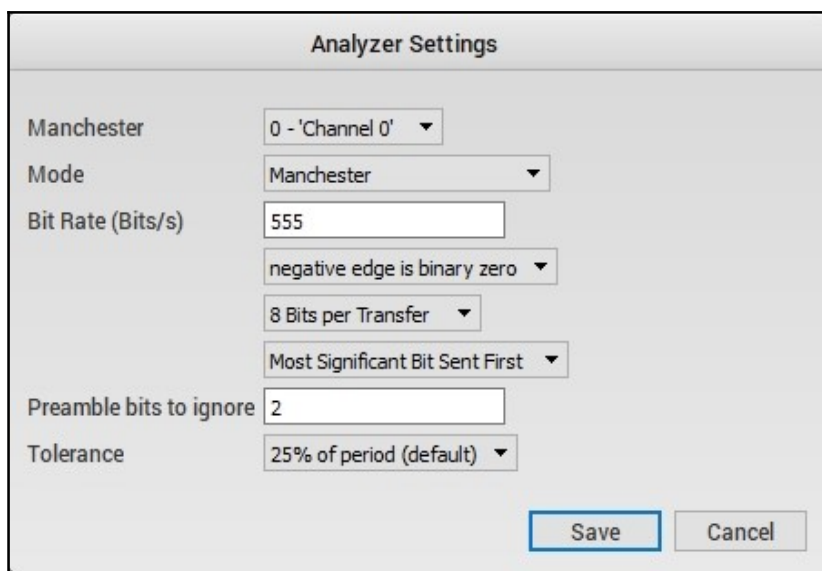


Abb. 4

Wir starten nun unser Sende-Programm und geben eine (kurze) Zeichenkette ein. Gleich nach der Eingabe starten wir die Aufzeichnung beim Analyzer. Innerhalb der im Sendeprogramm angegebenen Wartezeit von 3 s beginnt der Messvorgang und ein Diagramm wie in Abb. 1 sollte (bei entsprechender Vergrößerung) erscheinen.

4 Modulierte HF-Signale mit einer IR-LED senden

Zum Senden von RC-5-Signalen müssen wir ein **Rechtecksignal von 36 kHz als Trägersignal** erzeugen und **dieses mit dem Signal aus Kapitel 2 modulieren**. Dank der modularen Struktur unseres Programms `RC5_send_1h.py` aus Abschnitt 2 müssen wir im Wesentlichen nur an einer Stelle eine Änderung vornehmen, und zwar bei der Funktion `send_symbol`: In den Phasen mit dem Bitwert 1 (True) aktivieren wir ein PWM-Signal mit der Frequenz 36 kHz und in den in den Phasen mit dem Bitwert 0 (False) deaktivieren wir es.

PWM-Signale lassen sich mit Hilfe eines PWM-Objekts an jedem Ausgangs-Pin erzeugen: Die zugehörige **Klasse PWM** müssen wir zunächst aus dem `machine`-Modul importieren:

```
from machine import PWM
```

Mit dem Befehl

```
pwm_led = PWM(led, freq=36_000, duty_u16=0)
```

wird am Pin 25 (das ist der Pin, welcher mit dem bereits definierten `led`-Objekt verbunden ist) ein PWM-Signal mit der Frequenz von 36 kHz erzeugt. Bei einem Duty-Wert von $2^{15} = 32768$ ist die Impulsdauer halb so groß wie die Periodendauer (der maximale `duty_u16`-Wert ist $2^{16}-1$); High- und Low-Zustand einer Periode sind dann gleich groß. Ist der Duty-Wert 0 (wie bei unserer Instanziierung), dann ist die Impulsdauer auch 0, d. h. wir erhalten ein konstantes 0-Signal am Pin 25.

Die Funktion zum Senden eines Symbols lautet damit:

```
def send_symbol(b): # b ist True (=1) oder False (=0)
    if b:
        pwm_led.duty_u16(0)
        sleep_us(T)
        pwm_led.duty_u16(32768)
        sleep_us(T)
    else:
        pwm_led.duty_u16(32768)
        sleep_us(T)
        pwm_led.duty_u16(0)
        sleep_us(T)
```

Zuletzt nehmen wir noch folgende Änderungen vor:

- Den Befehl `led.value(0)` am Anfang des Programms können wir ersatzlos streichen; das Signal wird schon durch die Initialisierung von PWM mit `pwm_led.duty_u16(0)` auf 0 gesetzt.
- Den Befehl `led.value(0)` am Ende des Programms ersetzen wir durch `pwm_led.duty_u16(0)`
- Für `T` wählen wir jetzt einen kleineren Wert, nämlich 870, weil `send_symbol` für die zusätzlichen Befehle etwas Zeit benötigt.



Abb. 5: Das RC5-Signal für die Zeichenkette "hallo"

In Abb. 5 ist der Anfang eines Signals zu sehen, welches mit diesem Programm `RC5_send_2c.py` erzeugt wurde. Hier wurde die Zeichenkette "hallo" übertragen. In dieser Abbildung sind die PWM-Signale (genauer: die Halbbits mit 1-Signal) wegen der hohen Frequenz nur als graue Blöcke zu sehen. Für die folgende Abbildung wurde das 2. Halbbit des ersten Präambelbits stark vergrößert: Hier können wir deutlich das PWM-Signal erkennen;



Abb. 6: Signal eines Halbbits mit 1-Signal.

Der Digital Analyzer zeigt uns auch folgende Daten an: die Frequenz (36,04 kHz) die Dauer des 1-Halbbits (13,88 μ s) sowie die Periodendauer (27,75 μ s). Das passt gut zu unseren benutzten Programmwerten `freq = 36_000` und `duty_u16 = 215`.

Dieses Signal senden wir nun aus, indem wir an Pin 25 eine IR-LED anschließen. Ich benutze hier dazu das **IR-Transmitter-Modul** aus Abb. 7. Auf diesem kleinen Board gibt es neben einem Vorwiderstand für die IR-LED auch eine LED (im sichtbaren Bereich); diese erlaubt es, den Signalzustand auch mit dem Auge zu kontrollieren. Natürlich kann man ebenso auf Standard-IR-LEDs zurückgreifen (vgl. auch den Beitrag auf meinem Forum: <https://forum.g-heinrichs.de/viewtopic.php?f=12&t=71>).

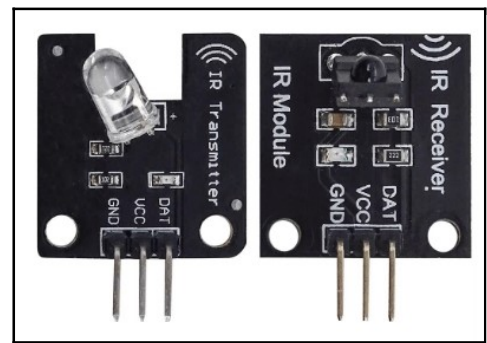


Abb. 7

5 IR-Signal empfangen und mit einem Digital Analyzer dekodieren

Als Empfänger dient der zugehörige **IR-Receiver** aus Abb. 7. Der Baustein am oberen Rand des Moduls empfängt die IR-Signale und **demoduliert** sie; das demodulierte Signal wird optisch durch eine rote LED angezeigt; dies kann z. B. bei einer Fehlersuche ganz hilfreich sein. Statt dieses Moduls kann man auch "einfache" IR-Empfänger" wie z. B. den Baustein TSOP1738 benutzen.

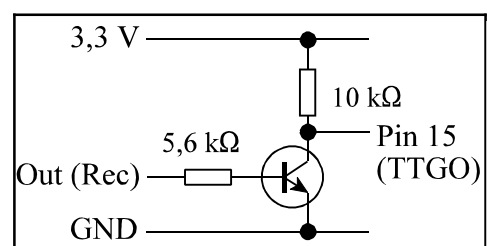


Abb. 8

Leider ist das Ausgangssignal (OUT) des IR-Receivers (ebenso wie beim TSOP1738) invertiert. Damit das Signal wieder die bislang benutzte Polung aufweist, invertieren wir das Ausgangssignal mit einer einfachen Transistor-Schaltung (Abb. 8). Der Einfachheit halber bauen wir die Schaltung auf demselben Breadboard auf wie den TTGO; dann können wir den 3V-Anschluss des TTGO auch als elektrische Quelle (3,3 V) benutzen.

Das Board mit dem Sender und das Board mit dem Empfänger werden jetzt gegenüber aufgestellt. Wenn wir jetzt mit dem Sende-Programm RC5_send_2c (vgl. Abschnitt 2) eine Zeichenkette senden, sollten wir (ggf. nach einer Warte-Zeit von 3 s) am Empfänger-Modul die rote Kontroll-LED kurz aufleuchten sehen.

Nun schließen wir den Ausgang der Transistorschaltung (in Abb. 8 mit Pin 15 TTGO) an Kanal 0 (CH1) des Analyzers an und verbinden auch die beiden GND-Anschlüsse. Die Einstellungen des Analyzers sollten dieselben sein wie in Abschnitt 3. Jetzt starten wir den Analyzer wie in Abschnitt 3 beschrieben und senden wie eben eine Zeichenkette. Der Analyzer zeigt jetzt das demodulierte Signal an, das wie in Abb. 1 aussieht.

6 Demoduliertes IR-Signal mit dem TTGO dekodieren

Das demodulierte Signal entspricht im Wesentlichen dem Signal, welches in den Abschnitten 2 und 3 behandelt worden ist. Dieses Signal gilt es nun zu dekodieren. Dabei folgen wir der am Ende von Abschnitt 1 angegebenen Strategie. Das zugehörige Programm RC5_empf_1a.py sieht folgendermaßen aus:

```
# RC5_empf_1a.py: Zeichenketten über IR-Modul im Manchestercode
# (mit 36 kHz Trägersignal) empfangen
# Präambel: zwei 1-Bits; Payload: 1 Byte, d. h. 8 Bits; T ≈ 900 us (real)
# Sender sollte zwischen zwei Zeichen ca 30 ms warten.
# getestet mit rc5_send_2c.py

from time import sleep, sleep_us, sleep_ms
from machine import Pin, PWM

signal = Pin(25, Pin.IN) # für Empfangssignal

def wait_for_start_bit():
    while not signal.value():
        pass

def get_bit_list(n, t): # n = Anzahl der Bits, t = Zeit für 1 (ganzes) Bit
    global signal
    l = []
    for i in range(n):
        l.append(signal.value())
        sleep_us(t)
    return l

def bit_list_to_nr(l): # Zahl
    n = len(l)
    value = l[0]
    for i in range(1, n):
        value = (value << 1) + l[i] # value = value * 2 + l[i]
    return value

# Hauptprogramm
T = 875 # Halbe Bitzeit in us
```



```

T_2 = T * 2 # Signallänge für 1 Bit
T_4_2 = 4 * T + T // 2 # s. u.
nr_of_bits = 8 # Hier werden Bytes übertragen

print('Warte auf Botschaften...')

# Einzelne Bytes empfangen und auf dem Terminal ausgeben
while True:
    wait_for_start_bit()
    sleep_us(T_4_2) # jetzt in der Mitte der 2. Hälfte des 1. Payload-
                    # Bits (nr_of_startbits = 2)
    bit_list = get_bit_list(nr_of_bits, T_2)
    value = bit_list_to_nr(bit_list)
    if nr_of_bits == 8:
        print(chr(value), end = '')
    sleep_us(T) # jetzt hinter dem letzten Bit

```

Zum Testen schließen wir den TTGO mit dem IR-Receiver an den Empfangsrechner an und starten dieses Programm (RC5_empf_1a.py). Den TTGO mit dem IR-Transmitter schließen wir an den Senderechner an und starten dort das Programm RC5_send_2c.py. Im Terminal des Senderechners geben wir hinter der Eingabe-Aufforderung "Text:" eine Zeichenkette ein, z. B. "Hallo Welt". Diese Zeichenkette sollte danach im Terminal des Empfangsrechner erscheinen.

Für jede neue Übertragung müssen Sie das Sendeprogramm RC5_send_2c.py neu starten; dies können Sie mit Hilfe einer zweiten Schleifen-Struktur vermeiden (s. RC5_send_2d.py). Das Empfangsprogramm muss hingegen nicht neu gestartet werden, da es in einer Endlosschleife auf neue Bytes wartet.

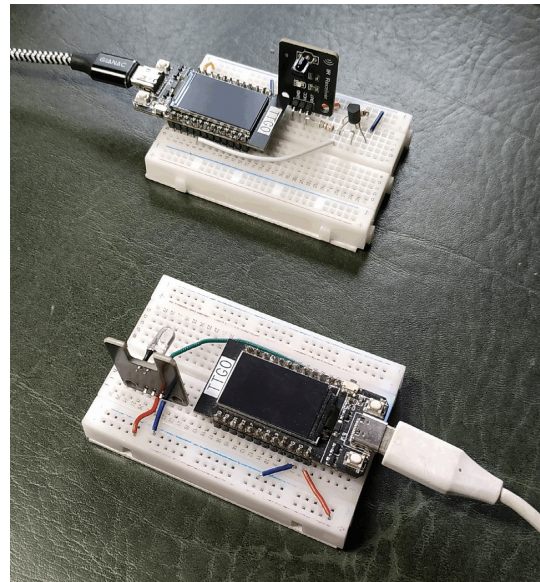


Abb. 9

7 Empfänger für RC-5-Signale einer Fernbedienung programmieren

Das Empfangsprogramm für eine echte(!) RC-5-Fernbedienung unterscheidet sich von unserem Programm RC5_empf_1a.py nur in der Länge der Bit-Liste (bit_list) und deren Auswertung:

Die Anzahl der Bits (nr_of_bits) ist jetzt nicht 8, sondern 12: 1 Toggle-Bit, 5 Adress-Bits (Gerät) und 6 Befehls-Bits.

Die (numerischen) Werte für das Toggle-Bit, die Gerätenummer und den Befehlscode erhalten wir aus den zugehörigen Teillisten:

Toggle-Bit	bit_list[0]
Gerätenummer	bit_list_to_nr(bit_list[1:6])
Befehlscode	bit_list_to_nr(bit_list[6:12])

Das vollständige Programm finden Sie unter dem Dateinamen RC5_Fernbed_empf_1.py.

Wenn Sie das Programm mit einer RC-5-Fernbedienung austesten wollen, sollten Sie beachten: Wo RC-5-draufsteht, muss nicht unbedingt RC-5 drin sein, vgl. auch Abschnitt 10. Falls sie keine (echte) RC-5-Fernbedienung haben, können Sie zum Testen auch das Programm rc5_Fernbed_send_1.py benutzen, welches im nächsten Abschnitt vorgestellt wird.

8 Sender für RC-5-Befehle

Im Wesentlichen entspricht das Programm rc5_Fernbed_send_1.py zum Senden von RC-5-Befehlen dem Programm aus Abschnitt 4. Lediglich das Hauptprogramm muss der neuen Datenstruktur (1 + 5 + 6 Bits) angepasst werden.

```
# Hauptprogramm
while True:
    # Eingabe von Toggle-Bit, Gerätenummer und Befehlscode:
    print('Eingaben:')
    toggle_bit = int(input('Toggle-Bit (0/1): '))
    if toggle_bit < 0 or toggle_bit > 1:
        print('Ungültiger Wert!')
        exit()
    address = int(input('Gerätenummer(0-31): '))
    if address < 0 or address > 31:
        print('Ungültiger Wert!')
        exit()
    command = int(input('Befehlscode(0-63): '))
    if command < 0 or command > 63:
        print('Ungültiger Wert!')
        exit()
    print()
    bit_list = [toggle_bit]
    bit_list = bit_list + to_bitlist(5, address)
    bit_list = bit_list + to_bitlist(6, command)
    # print(bit_list, 'wird gesendet.') # zum Testen
    print()
    # Signal senden
    pre_list = [1,1] # Präambel
    # Präambel senden:
    send_bitlist(pre_list)
    # Bitliste für Zeichen senden:
    send_bitlist(bit_list)
    pwm_led.duty_u16(0) # mit Low-Signal beenden
    sleep_ms(30)
```

Bei der Eingabe der Werte für das Toggle-Bit, die Gerätenummer und den Befehlscode wird jeweils überprüft, ob diese im zulässigen Bereich sind; sollte das nicht der Fall sein, wird das Programm der Einfachheit halber mit der Methode `exit()` abgebrochen; diese muss zu Beginn des Programms mit `from sys import exit` importiert werden.

Nun werden die zugehörigen Bit-Listen gebildet und zu einer einzigen Liste `bit_list` (mit 12 Elementen) zusammengefügt. Schließlich werden die Präambel (`pre_list`) und die gerade erzeugte Bitliste mit der bekannten Funktion `send_bitlist` gesendet.

9 RC-5-Fernbedienungs-App

Es gibt eine Reihe von Apps, mit deren Hilfe ein Handy als IR-Fernbedienung eingesetzt werden kann. Hier möchte ich die App **IR Remote Creator** der Firma **keuwlsoft** (keuwl.com) vorstellen. Mit Ihrer Hilfe können die unterschiedlichsten Fernbedienungen simuliert werden. Dabei kann der Anwender auf recht einfache Weise sowohl das Design (Form der Fernbedienung und ihrer Knöpfe) als auch die Signale festlegen, welche durch Betätigen der Knöpfe ausgesendet werden sollen. Dabei empfiehlt es sich, von einer der bereits implementierten Fernbedienungen auszugehen und diese nach eigenen Wünschen zu modifizieren. Selbst erstellte Fernbedienungen können (als txt-Datei) gespeichert werden.

In Abb. 10 sehen Sie den Screenshot von der Fernbedienung mit dem Namen "Remote 1". Diese kann man als Ausgangspunkt für die Erzeugung einer eigenen RC-5-Fernbedienung wählen. Die Vorgehensweise ist (ähnlich wie bei der App "Bluetooth Electronics" vom gleichen Anbieter, vgl. <https://forum.g-heinrichs.de/viewtopic.php?t=133>) recht intuitiv: In der Menü-Zeile am oberen Rand sind Buttons für die folgenden Rubriken zu sehen:



Abb. 10

- **Werkzeug:** Hier können grundlegende Einstellungen vorgenommen werden. Die wichtigste ist wohl die erste Option, die Einstellung der Betriebsart (Mode): Die Standardeinstellung ist "Creator"; nur bei dieser Einstellung können Sie die Fernbedienung in Aussehen und Funktionsweise ändern.
- **Information:** Hier erhalten Sie Informationen zum Umgang mit der App.
- **Datei laden oder speichern:** Die Einstellungen liegen in Form von txt-Dateien vor; Sie können geänderte oder neu erstellte Fernbedienungen hier speichern oder bereits bestehende erneut laden. Die Dateien befinden sich im Verzeichnis `/storage/emulated/0/Documents/keuwlsoft/ir-remotes`.
- **Editieren:** Hier können Sie eine bestehende Fernbedienung mit Hilfe der Werkzeuge in der linken Leiste editieren. Sie können ihre Form ändern, Knöpfe hinzufügen, entfernen oder abändern. Insbesondere können Sie hier auch – und das ist das Entscheidende – festlegen,

welches Signal durch diese Knöpfe jeweils ausgesendet werden soll (Mehr dazu im nächsten Abschnitt!)

- **Auswahl:** Hier können Sie aus einer Liste von bestehenden Fernbedienungen die gewünschte auswählen.
- **Starten:** Hierdurch wird die aktuelle Fernbedienung aktiviert. Wenn Sie dann einen der Knöpfe betätigen, wird das zugehörige Signal ausgesendet.

Für unsere neue Fernbedienung benutzen wir als Ausgangspunkt die Vorlage "Remote 1" aus Abb. 10. Zunächst ändern wir das Design ein wenig: Mit Hilfe der Werkzeugleiste entfernen wir als Erstes sämtliche Schaltflächen, welche sich unter den grünen Knöpfen befinden; dazu tippen wir jeweils auf den zu entfernenden Knopf und betätigen das **Mülleimer-Symbol**. Nun geben wir den Auf- und Ab-Tasten mit Hilfe des **Text-Symbols** noch die Beschriftungen "+" und "-". Anschließend verkleinern wir die Fernbedienung auf die Maße $h = 700$ und $b = 512$. Das Ergebnis ist in Abb. 11 zu sehen.



Abb. 11

Die oberen grauen Knöpfe sollen zur Steuerung eines **Fernsehers (TV)**, die darunter liegenden grünen zur Steuerung eines **CD-Players** dienen. Jetzt müssen wir für die einzelnen Schaltflächen festlegen, welche Signale ausgesendet werden sollen.

Wie man dabei vorgeht, erläutern wir an Hand des Knopfes mit der Aufschrift "4": Zunächst tippen wir auf diesen Knopf. Dann betätigen wir die **IR-Schaltfläche** am unteren Rand der Werkzeugleiste. Rechts von der Werkzeugleiste erscheinen dann die aktuellen Parameter für das Signal (Format, Frequenz, Befehl/Command und Adresse; an den beiden darunter befindlichen Werten wollen wir nichts ändern.)

Im Auswahlmenü **Format** wählen wir RC-5. Bei der **Frequenz** aktivieren wir den Wert 36 kHz. Bei dem **Kommando** geben wir den Wert 4 (für 4. Programm) ein. Der Fernseher hat standardmäßig die Adresse 0; diesen Wert tragen wir im Adress-Feld ein.

Oberhalb der Eingabe-Felder sehen wir jetzt auch das zugehörige **Zeit-Diagramm für das IR-Signal** (Abb. 12). Es weist die für RC-5-Signale typischen 12 Rechteck-Signale auf.

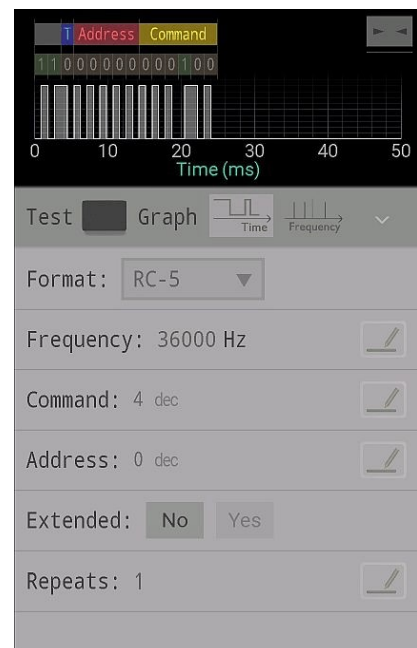


Abb. 12

Wenn Sie nun die Fernbedienung mit dem Starten-Symbol aktivieren, können Sie durch Betätigen des 4-Knopfes das "4"-Signal aussenden. Dies können Sie z. B. mit einem RC-5-Empfänger (vgl. Abschnitt 7) kontrollieren.

Es empfiehlt sich, die neu erzeugten Fernbedienungseinstellungen abzuspeichern.

Für die weiteren Einstellungen können Sie die folgende Tabelle benutzen; die Daten stammen von der Webseite <https://en.wikipedia.org/wiki/RC-5>.

Knopf (Btn-Nr)	Bild	Gerät	Kommando
1	0	0	0
2	1	0	1
...
9	8	0	8
10	9	0	9
11	Ein	0	10 (Einschalten)
12	▲	0	13 (Vol +)
13	▼	0	14 (Vol -)
14	<<	20	33 (zurück)
15		20	48 (anhalten)
16	□	20	54 (stoppen)
17	⬢	20	53 (spielen)
18	>>	20	32 (vorwärts)

Wer die Daten nicht alle selbst eingeben möchte kann auf folgende txt-Datei zurückgreifen:

Keuwlsoft IR Remote File (Version 1)
2024-10-02 19:39:13

```

new_remote(My_RC-5 ,512,700,3)
  title_color(255,255,255)
  add_button(196,432,102,0,2,0)
    button_text(0,41,,,255,255,255)
    transmit_rc5(36000,0x00,0x00,0,1,0)
  add_button(96,192,102,0,2,0)
    button_text(1,41,,,255,255,255)
    transmit_rc5(36000,0x01,0x00,0,1,0)
  add_button(196,192,102,0,2,0)
    button_text(2,41,,,255,255,255)
    transmit_rc5(36000,0x02,0x00,0,1,0)
  add_button(296,192,102,0,2,0)
    button_text(3,41,,,255,255,255)
    transmit_rc5(36000,0x03,0x00,0,1,0)
  add_button(96,272,102,0,2,0)
    button_text(4,41,,,255,255,255)
    transmit_rc5(36000,0x04,0x00,0,1,0)
  add_button(196,272,102,0,2,0)
    button_text(5,41,,,255,255,255)
    transmit_rc5(36000,0x05,0x00,0,1,0)
  add_button(296,272,102,0,2,0)
    button_text(6,41,,,255,255,255)
    transmit_rc5(36000,0x06,0x00,0,1,0)
  add_button(96,352,102,0,2,0)
    button_text(7,41,,,255,255,255)
    transmit_rc5(36000,0x07,0x00,0,1,0)
    add_button(196,352,102,0,2,0)
      button_text(8,41,,,255,255,255)
      transmit_rc5(36000,0x08,0x00,0,1,0)
    add_button(296,352,102,0,2,0)
      button_text(9,41,,,255,255,255)
      transmit_rc5(36000,0x09,0x00,0,1,0)
    add_button(96,92,86,3,6,2)
      transmit_rc5(36000,0x0A,0x00,0,1,0)
    add_button(416,212,88,7,2,0)
      button_text(+,36,,,255,255,255)
      transmit_rc5(36000,0x0D,0x00,0,1,0)
    add_button(416,332,88,8,2,0)
      button_text(-,36,,,255,255,255)
      transmit_rc5(36000,0x0E,0x00,0,1,0)
    add_button(96,532,92,1,17,10)
      transmit_rc5(36000,0x21,0x14,0,1,0)
    add_button(176,532,92,1,17,5)
      transmit_rc5(36000,0x30,0x14,0,1,0)
    add_button(256,532,92,1,17,6)
      transmit_rc5(36000,0x36,0x14,0,1,0)
    add_button(336,532,92,1,17,7)
      transmit_rc5(36000,0x35,0x14,0,1,0)
    add_button(416,532,92,1,17,11)
      transmit_rc5(36000,0x20,0x14,0,1,0)

```

Sie finden diese Datei unter dem Namen `Remote_0001c.txt`. Wenn Sie diese in das oben angegebene Verzeichnis Ihres Handys kopieren, können Sie sie in die Remote App laden.

10 Ein Rekorder für Pseudo-RC-5-Fernbedienungen

Als ich jüngst im Internet nach alten RC-5-Fernbedienungen suchte, stieß ich auf zahlreiche Angebote. Es stellte sich aber schnell heraus, dass es sich hierbei um Universal-Fernbedienungen handelte. Vermutlich arbeiten diese wie eine "echte" RC-5-Fernbedienung auch mit einer IR-LED und einer Trägerfrequenz von 36 kHz (oder auch 38 kHz). Aber die Kodierung dieser Fernbedienungen ist anders: Wie eine Untersuchung mit unserem Digital Analyzer zeigte, waren deren Signale häufig deutlich komplexer und länger als die (echten) RC-5-Signale. Eine solche vom Standard-Code abweichende Fernbedienung bezeichne ich hier als "Pseudo-RC-5-Fernbedienung". Für diese habe ich ein **Rekorder-Programm** für den TTGO entwickelt: Dieses Programm kann das Signal für einen Befehl (mit Hilfe eines IR-Receivers) aufzeichnen und (mit Hilfe eines IR-Transmitters) auch wieder abspielen. Derartige Fernbedienungen sind im Handel auch unter der Bezeichnung "lernfähige Fernbedienung" erhältlich. (Beachten Sie: Universal-Fernbedienungen sind nicht unbedingt lernfähig; häufig besitzen sie nur einen fertigen Vorrat an Codes von gängigen Geräten.)

Ein solcher Recorder besitzt im Wesentlichen zwei Funktionen:

- **Aufnahme des Signals:** Von einer bestehenden Fernbedienung empfängt er das Signal für ein bestimmtes Kommando, z. B. mittels eines IR-Receivers (vgl. Abb. 7). Dabei wartet er zunächst auf das erste High-Signal; in dem Augenblick, wo das Signal nun von 0 auf 1 wechselt, startet er einen Timer. Nun wartet er, bis das Signal wieder auf 0 geht. Ist dies der Fall, speichert er mit Hilfe des Timers die Dauer der High-Phase ab. Dann wartet er auf die nächste High-Phase und speichert deren Dauer bei Erreichen der nächsten Low-Phase ebenfalls ab. So geht es immer weiter; auf diese Weise erhalten wir eine Liste mit den Zeitintervallen der einzelnen High- und Low-Phasen. Der Vorgang wird schließlich abgebrochen, wenn die Gesamtzeit einen bestimmten Wert überschritten hat (bei RC-5 z. B. 30 ms). Die Liste dieser Zeitintervalle wird schließlich in einer Text-Datei abgespeichert.
- **Wiedergabe des Signals:** Die Zeitdauern der einzelnen High- und Low-Phasen werden aus der Datei gelesen. Der Reihe nach werden nun PWM-Signale mit entsprechenden Dauern und Duty-Werten erzeugt (ganz ähnlich wie bei der Funktion `send_symbol` aus dem Abschnitt 4); der letzte Duty-Wert ist dabei immer 0.

Ich verzichte darauf, das entsprechende Programm an dieser Stelle wiederzugeben. Sie finden es unter dem Namen `RC5_Recorder_3a.py`. Ich habe es mit einer *majority*-Fernbedienung (mit $T_{\max} = 80000$) erfolgreich getestet.

Interessant für den Anwender ist vielleicht noch die folgende Bemerkung: Vom letzten Zeitintervall abgesehen, sollten die Zeitintervalle bei unserer Fernbedienung aus dem Abschnitt 9 alle etwa 1,8 ms bzw. 0,9 ms lang sein. Ein Blick auf die Liste zeigte aber erheblich Schwankungen. Zunächst hatte ich vermutet, dass Micropython den zeitkritischen Anforderungen nicht gewachsen gewesen sei. Eine Kontrolle des von der App erzeugten Signals mit Hilfe eines Digital Analysators zeigte jedoch eine recht gute Übereinstimmung mit den Zeitwerten aus der Liste. Die Schwankungen sind also zum größten Teil auf die App (oder mein Handy) zurückzuführen.

11 Ein Empfänger für RC-5-Signale mit unbekanntem Bit-Intervall

Am Ende von Abschnitt 1 hatte ich schon darauf hingewiesen, dass der Empfänger bei einem (echten) RC-5-Signal die Länge des Bit-Intervalls aus der Präambel ableiten kann. Wie in den ersten Abschnitten möchte ich mich hier auf die Übertragung von Zeichenketten beschränken. Als Sende-Programm können wir einfach unser Programm `RC5_send_2c.py` bzw. `RC5_send_2d.py` mit einem anderen Wert für die halbe Bitzeit T benutzen.

Wie sieht nun ein entsprechendes Empfangsprogramm aus? Im Wesentlichen können wir auf das bereits vorgestellte Programm `RC5_empf_1a.py` zurückgreifen. Wir ergänzen es zunächst um die folgende Funktion

```
def getBitzeitT(): # liefert halbe Bitzeit
    t0 = ticks_us()
    while signal.value(): # auf pos. Flanke des ersten Präambel-Bits warten
        pass
    while not signal.value(): # auf nächste neg. Flanke warten
        pass
    return ticks_diff(ticks_us() - t0) // 2
```

Dabei müssen die Funktionen `ticks_us` und `ticks_diff` zuvor aus dem Modul `time` importiert werden. Dann entfernen wir die Zuweisungen

```
T = 875 # Halbe Bitzeit in us
T_2 = T * 2
T_4_2 = 4 * T + T // 2 # s. u.
```

am Anfang des Hauptprogramms; die entsprechenden angepassten Informationen fügen wir jetzt in die `while`-Schleife des Hauptprogramms ein:

```
while True:
    wait_for_start_bit()
    T = getBitzeitT() # jetzt am Ende des 1. Präambel-Bits
    T_2 = T * 2
    T_2_2 = 2 * T + T // 2
    sleep_us(T_2_2) # jetzt in der Mitte der 2. Hälfte des 1. Payload-Bits
    bit_list = get_bit_list(nr_of_bits, T_2)
    value = bit_list_to_nr(bit_list)
    if nr_of_bits == 8:
        print(chr(value), end = '')
    sleep_us(T) # jetzt hinter dem letzten Bit
```

Das vollständige Programm finden Sie unter dem Dateinamen `RC5_empf_2a.py`.

12 Ein RC-5-Empfänger mit einstellbarer Toleranz

Die in Abschnitt 1 beschriebene Strategie zur Auswertung von RC-5-Signalen stößt an ihre Grenzen, wenn die Länge der High- und Low-Phasen des RC-5-Signals mehr oder weniger von T =

0,889 μs bzw. $2T = 1,778 \mu\text{s}$ abweichen. Nehmen wir einmal an, dass 10 aufeinander folgende High- und Low-Phasen mit der Länge T jeweils um 10 % zu groß sind. Dann wird z. B. bei der 9. Messung nicht der Wert des neunten, sondern der des achten Halb-Bits gemessen (vgl. Abb. 13).

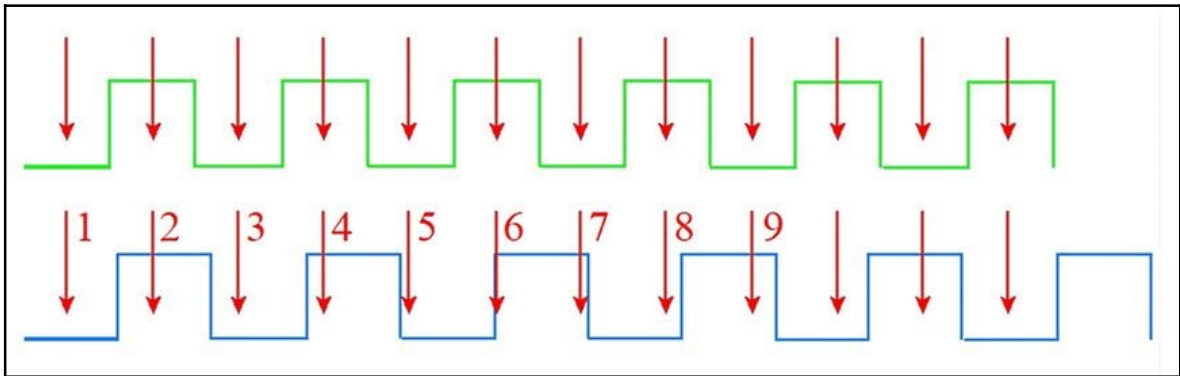


Abb. 13: Bei dem unteren Signal ist die Halb-Bit-Dauer um 10 % größer als bei dem oberen.

Für eine Verbesserung bietet sich folgende Vorgehensweise an: In der **Messphase** werden zunächst die Zeitdauern ΔT der einzelnen High- und Low-Phasen gemessen und in einer Liste `deltaT_list` gespeichert. Diese Werte werden nun in zwei aufeinander folgenden Phasen ausgewertet.

In der **ersten Auswertungsphase** werden den Zeitintervallen 1 oder 2 Halb-Bits zugeordnet; die Signalwerte 0 bzw. 1 dieser Halb-Bits werden hier mit `False` bzw. `True` gekennzeichnet. Bei der Zuordnung wird nun mit einem Toleranzwert (z. B. 20 %) gearbeitet: Ist ΔT klein (d. h. befindet sich der Wert zwischen $T \cdot (1 - \text{Toleranzwert})$ und $T \cdot (1 + \text{Toleranzwert})$), dann wird ihm der negierte Wahrheitswert des vorangegangenen Halb-Bits zugeordnet: Hatte dieses z. B. den Wert `False`, dann wird unserem Zeitintervall nun der Wert `True` zugewiesen. Ist ΔT groß (d. h. befindet sich der Wert zwischen $2 \cdot T \cdot (1 - \text{Toleranzwert})$ und $2 \cdot T \cdot (1 + \text{Toleranzwert})$), dann werden ihm gleich zwei Halb-Bits zugeordnet: das erste Halb-Bit erhält dabei denselben Wert wie das vorangegangene, das zweite Halb-Bit den negierten Wert. So erhalten wir eine Folge von Wahrheitswerten für die Halb-Bits; diese wird als Liste `HB_list` abgespeichert. Dabei müssen das erste und letzte Halb-Bit gesondert betrachtet werden.

Würden wir die Werte 1 (für `True`) und 0 (für `False`) für diese Halb-Bits jetzt in ein Zeit-Diagramm mit einer konstanten Einheit für T (z. B. 1 cm) eintragen, so erhielten wir eine Rekonstruktion des ursprünglichen RC-5-Signals. Mit anderen Worten: Unsere Halb-Bit-Liste stellt das (von den tolerierten Ungenauigkeiten bereinigte) RC-5-Signal dar. Diese Liste muss nun noch in die gesuchte Bit-Liste übersetzt werden. In dieser **zweiten Auswertungsphase** betrachten wir die Elemente von `HB_list` paarweise:

- Dem Paar [`False`, `True`] wird der Bit-Wert 1 zugeordnet.
- Dem Paar [`True`, `False`] wird der Bit-Wert 0 zugeordnet.

Zuletzt müssen wir dieser Bit-Liste die entsprechenden Werte für das Toggle-Bit, die Geräte-Nummer und die Befehlsnummer gewinnen. Diese erfolgt genauso wie in Abschnitt 7 beschrieben.

Weitere Einzelheiten entnehme man den Kommentaren des Programms `RC5_empf_3d.py`.

13 Ein weiteres Fernbedienungsprotokoll: NEC

Neben dem RC-5-Protokoll gibt es zahlreiche weitere Protokolle für Fernbedienungen. Viele Hersteller haben für die von ihnen vertriebenen elektronischen Geräte eigene Protokolle entwickelt. Das NEC-Protokoll wird indes nicht nur bei NEC-Geräten eingesetzt, sondern auch von zahlreichen anderen Firmen, z. B. von Yamaha, Canon, Tevion, Harman/Kardon, Hitachi, JVC, Pioneer und Toshiba. Deswegen soll es hier etwas genauer dargestellt werden. Das NEC-Protokoll ist m. E. für diesen Beitrag auch insofern interessant, als es sich hinsichtlich seiner Struktur recht stark von RC-5 unterscheidet. Auf den ersten Blick scheint das NEC-Protokoll deutlich komplizierter zu sein als das RC-5-Protokoll (s. Abb. 14). Es wird sich aber zeigen, dass bei NEC ein passendes Empfangsprogramm mit einstellbarer Toleranz deutlich einfacher herzustellen ist.

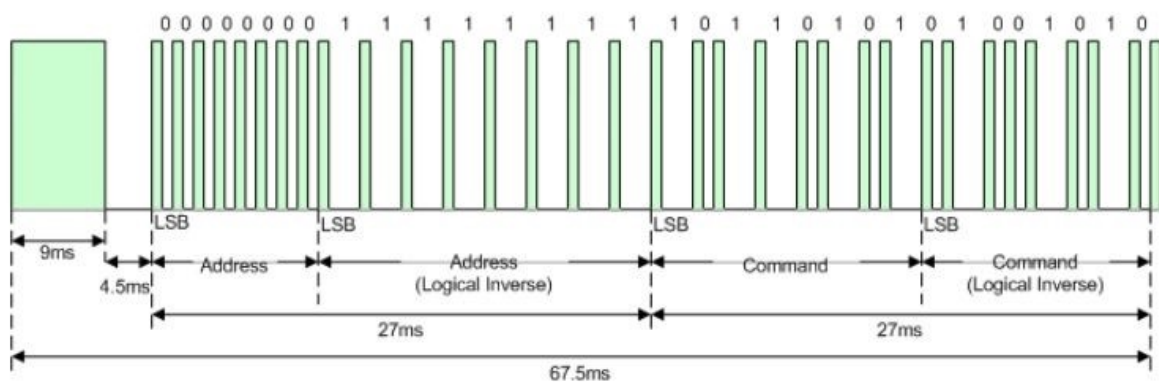


Abb. 14: aus: <https://techdocs.altium.com/display/FPGA/NEC+Infrared+Transmission+Protocol>

Wie bei RC-5 wird auch bei NEC ein hochfrequentes Träger-Signal von 36 kHz bzw. 38 kHz benutzt. (Nebenbei: Mein IR-Empfänger ist bezüglich der HF-Frequenz nicht sehr wählerisch: Er kann bei beiden Trägerfrequenzen zuverlässig eine Demodulation durchführen.) Für unser Empfangsprogramm können wir deswegen das demodulierte Signal aus Abb. 14 als Grundlage nehmen.

Das Signal gliedert sich in 6 Phasen:

Phase	Bezeichnung	Erläuterung
1	header	Das Signal beginnt mit einem 9 ms langen High-Signal (leading pulse burst) gefolgt von einem 4,5 ms langen Low-Signal (space).
2	address	8 Pulse, welche die 8 Bits des Adress-Bytes darstellen. Jeder Puls besteht aus einem $T = 562,5 \mu\text{s}$ langen High-Signal, gefolgt von einem Low-Signal. Letzteres ist $T = 562,5 \mu\text{s}$ lang bei einem Bitwert von 0 und $3T = 1687,5 \mu\text{s}$ lang bei einem Bitwert von 1. Der Bitwert hängt also nur von der Länge der Low-Phase ab. Die Bitfolge beginnt mit dem niederwertigsten Bit (Least Significant Bit). Die nächsten 3 Bytes werden auf dieselbe Art kodiert.
3	address complement	8 Pulse, welche das 1-Komplement (1 und 0 vertauscht) von der Adresse darstellen. (s. Fehlererkennung weiter unten)

4	command	8 Pulse, welche das Befehls-Byte darstellen.
5	command complement	8 Pulse, welche das 1-Komplement (1 und 0 vertauscht) des Befehls darstellen. (s. Fehlererkennung weiter unten)
6	end pulse burst	Dieser Puls beendet das Signal; er besteht lediglich aus einem High-Signal der Länge $T = 562,5 \mu\text{s}$. Ohne dieses Signal könnte der Empfänger die Länge des Low-Signals vom allerletzten Daten-Bit (und damit seinen Bit-Wert) nicht feststellen.

Die Bits mit dem Wert 1 sind jeweils zweimal so lang wie beim Wert 0. Trotzdem ist die Gesamtlänge eines vollständigen NEC-Signals immer gleich lang (nämlich 67,5 ms ohne den letzten Puls). Das liegt daran, dass es zu jedem Bit des Original-Bytes jeweils ein inverses Bit aus dem zugehörigen Komplement-Byte gibt. Original-Byte und Komplement-Byte können nach dem Empfang auch zu einer Fehlerüberprüfung ausgewertet werden.

Es folgt mein Empfangsprogramm NEC_empf_1.py; seine Funktionsweise sollte sich aus der ausführlichen Kommentierung ergeben.

```
# from time import ticks_us, sleep_ms, ticks_diff
from machine import Pin
from sys import exit

signal_in = Pin(25, Pin.IN) # für Empfangssignal (Aufnahme); IR-Receiver
# Toleranzen können auch kleiner sein z. B. +/- 500 us
leading_burst_0 = 9_000 # us
leading_burst_min = 8_000
leading_burst_max = 10_000
leading_space = 4_500
leading_space_min = 3_500
leading_space_max = 5_500

T1 = 1125 # us

address = [0,0,0,0,0,0,0,0]
address_inv = [0,0,0,0,0,0,0,0]
command = [0,0,0,0,0,0,0,0]
command_inv = [0,0,0,0,0,0,0,0]

##### Funktionen #####

def wait_for_high():
    while signal_in.value() == 0:
        pass
    T = ticks_us()
    return T

def wait_for_low():
    while signal_in.value() == 1:
        pass
    T = ticks_us()
    return T
```

```

def wait_for_leading_pulse(): # Zu Beginn liegt Low-Signal vor
    Ta = wait_for_high()
    Te = wait_for_low()
    leading_burst = ticks_diff(Te, Ta)
    print('lburst', leading_burst)
    if leading_burst_min < leading_burst < leading_burst_max:
        # Jetzt liegt High-Signal vor
        Ta = wait_for_low()
        Te = wait_for_high()
        leading_space = ticks_diff(Te, Ta)
        print('lspace', leading_space)
        if leading_space_min < leading_space < leading_space_max:
            return True # Jetzt liegt High-Signal vor
        else:
            return False
    else:
        return False

def get_byte(byte_list): # zu Beginn High-Signal
    for i in range(8):
        Ta = wait_for_low()
        Te = wait_for_high() # Jetzt am Ende des Bits
        deltaT = ticks_diff(Te, Ta)
        if deltaT < T1:
            byte_list[i] = 0
        else:
            byte_list[i] = 1
    return byte_list # Am Ende liegt High-Signal vor

def inv_test(list1, list2):
    result = True
    for i in range(8):
        result = result and (list1[i] != list2[i])
    return result

def bit_list_to_nr(l): # Bit-Liste in Zahl konvertieren (LSB ... LSB)
    n = len(l)
    value = l[n-1]
    for i in range(1,n):
        value = value * 2 + l[n-1-i]
    return value

##### Hauptprogramm #####

print('Programm gestartet... (Abbrechen mit Strg-C)')

while True:
    # Empfang eines NEC-Signals
    # Zu Beginn liegt Low-Signal vor
    if wait_for_leading_pulse() == False:
        print('leading pulse error')
        exit()
    # Jetzt liegt High-Signal vor
    address = get_byte(address) # Liste
    address_inv = get_byte(address_inv) # Liste
    command = get_byte(command) # Liste
    command_inv = get_byte(command_inv) # Liste

```

```

# Jetzt liegt High-Signal vor (Anfang vom Ende-Burst)
Ta = ticks_us()
Te = wait_for_low() # Ende des Bursts und Ende des gesamten NEC-Signals
end_burst = ticks_diff(Te, Ta)
# Jetzt liegt wieder ein Low-Signal vor, bis ein neues NEC-Signal empfangen wird

# Ausgabe der Werte:
print('End-Burst', end_burst)
print('Adresse:', address, '->', bit_list_to_nr(address))
print('Komplement der Adresse:', address_inv)
print('Adress-Test:', inv_test(address, address_inv))
print('Befehl:', command, '->', bit_list_to_nr(command))
print('Komplement des Befehls:', command_inv)
print('Befehls-Test:', inv_test(command, command_inv))
print()
sleep_ms(200) # falls Finger nicht schnell genug von der Taste weg...
# Jetzt bereit für ein nächstes NEC-Signal

```

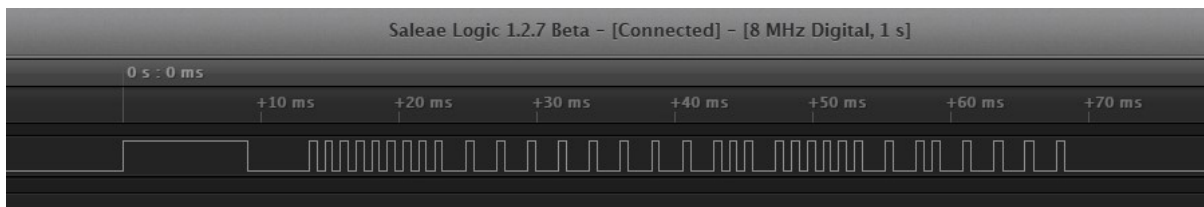


Abb. 15: NEC-Signal für Adresse = 0 und Befehl = 9

Hinweis

Sämtliche im Text angegebenen Programme (RC5_send_1h.py, RC5_send_2c.py, RC5_empf_1a.py, RC5_send_2d.py, RC5_recorder_3a.py, RC5_empf_2a.py, RC5_empf_3d.py, NEC_empf_1.py) finden Sie in der Anlage RC5_Programme.zip.