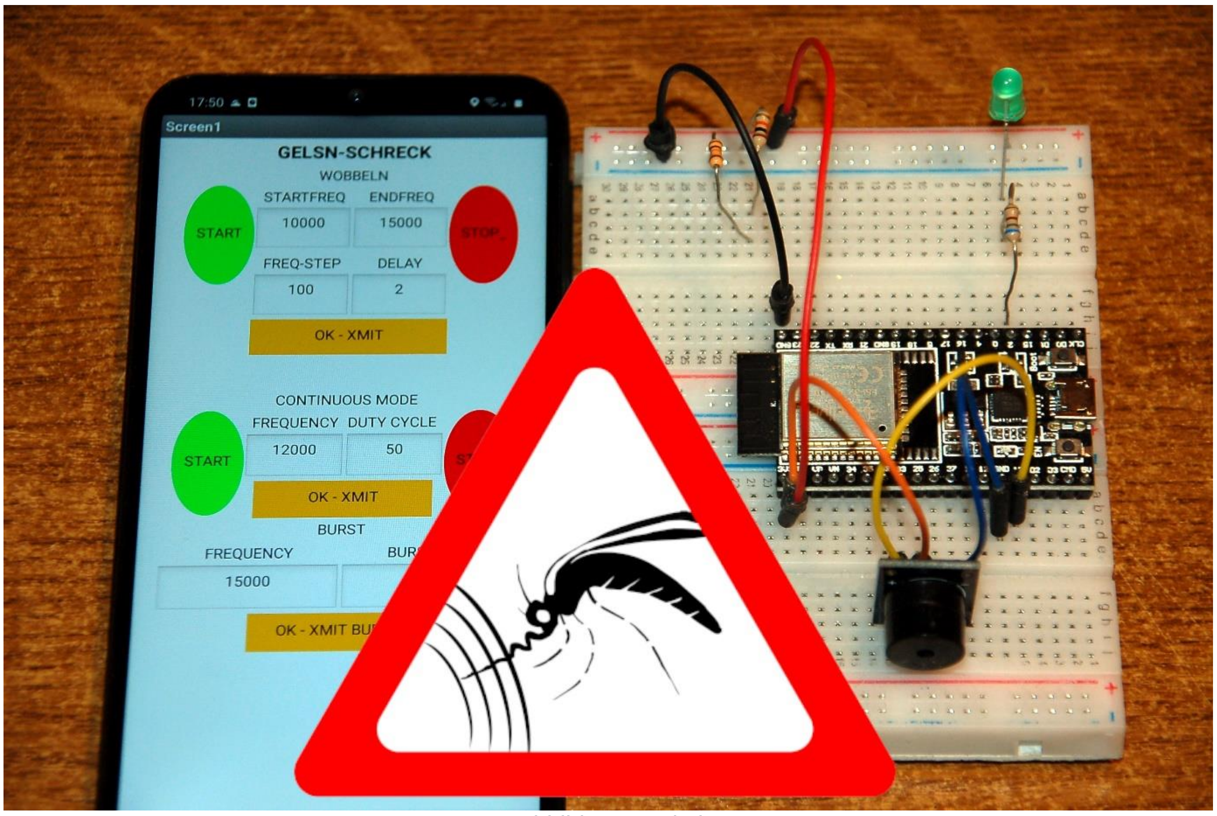


Abbildung 1: Titel

The insect fauna is decreasing more and more, we hear again and again. Well, if I compare the display of my motorcycle helmet after a night ride today with the one 10 years ago, that might be true. Still, it would be nice if you didn't have to scratch 20 or more mosquito bites the next day after an evening outdoors. Apparently there are still far too many of the pests. Some people swear by ultrasound fighting the beasts around the seating area. This refers to frequencies above the human hearing range, i.e. from 20 kHz. The circuit that I am presenting today, together with an app for an Android mobile phone that I will use as a control (next episode), could therefore provide a remedy against the mosquito maffia. The sounds from this circuit can also scare off rodents with long tails, dogs, cats and other animals. Thanks to the few components, the structure is very clear and therefore ideally suited for newcomers. So welcome to the blog with the title

Gelsenschrecker with the ESP32 and MicroPython

Gelsen (or Gelsn pronounced: "Göisn") refers to the name of the mosquito in Lower Bavaria and Austria. The same bastards can really spoil an evening outdoors. You can smell yourself with mosquito repellent, wrap yourself up to your neck in protective clothing or simply use the ultrasonic cannon to protect yourself against the attack. Exactly the latter is the content of this description. Of course, this requires a certain amount of hardware and software. Here is the list of parts for the four possible expansion stages.

Hardware components

	Grundaufbau
1	ESP32 Lolin LOLIN32 oder ESP32 D1 Mini NodeMCU WiFi Modul
1	KY-006 Passiver Piezo Buzzer Alarm Modul
1	LED (grün)
1	Widerstand 680 Ω
2	Widerstand 10 kΩ
2	Mini Breadboard 400 Pin mit 4 Stromschienen für Jumper Kabel
diverse	Jumperkabel
	Ausbaustufe 1
1	MOS40106 sechsfach Inverter
1	Transistor BC550 o. ä.
1	Widerstand 2,2 kΩ
1	Widerstand 10 kΩ
1	12 V-Spannungsquelle
	Ausbaustufe 2
1	Hochtonhorn 105 dB-110 dB, 4000..40000 Hz
	Ausbaustufe 3
1	MT3608 DC-DC Netzteil Adapter Step up Modul
	Verstärker mit H-Brücke aus folgenden Teilen
2	BD135
2	BD136
3	BC550
1	Widerstand 1 kΩ
1	Widerstand 1,5 kΩ
1	Widerstand 10 kΩ
2	Widerstand 2,7 kΩ
2	Widerstand 10 Ω
2	Widerstand 820 Ω
1	Elektrolytkondensator 100 µF / 16 V
1	Elektrolytkondensator 470 µF / 35 V
1	Kühlkörper ca. 30 x 70 mm
4	Silikon- oder Glimmerscheiben
4	Schrauben M3x10 + Muttern
1	Platine 60 x 100mm, einseitig kaschiert oder Lochrasterplatine

Software

For flashing and programming the ESP32:
Thonny oder
µPyCraft
packetsender for testing the ESP32 as a UDP server

Used Firmware:

MicropythonFirmware
Please choose a stable version

MicroPython-Programs:
http://www.grzesina.de/az/gelsenschreck/gelsnschreck_test.py for testing via REPL input request

MicroPython - Language - Modules and Programs

You can find detailed instructions for installing Thonny here. There is also a description of how the Micropython firmware is burned onto the ESP chip.

MicroPython is an interpreter language. The main difference to the Arduino IDE, where you always and exclusively flash entire programs, is that you only have to flash the MicroPython firmware once at the beginning on the ESP32 before the controller understands MicroPython instructions. You can use Thonny, µPyCraft or esptool.py for this. I have described the process for Thonny here.

As soon as the firmware is flashed, you can have a casual conversation with your controller, test individual commands and immediately see the answer without first having to compile and transfer an entire program. This is exactly what bothers me about the Arduino IDE. You simply save an enormous amount of time if you can do simple tests of the syntax and hardware through to trying out and refining functions and entire program parts via the command line before you knit a program out of it. For this purpose I also like to create small test programs over and over again. As a kind of macro, they combine recurring commands. From such program fragments, entire applications can develop.

Autostart

If the program is to start autonomously when the controller is switched on, copy the program text into a newly created blank file. Save this file under boot.py in the workspace and upload it to the ESP chip. The program starts automatically the next time it is reset or switched on.

Testing programs

If the program is to start autonomously when the controller is switched on, copy the program text into a newly created blank file. Save this file under boot.py in the workspace and upload it to the ESP chip. The program starts automatically the next time it is reset or switched on.

In between, Arduino IDE again?

If you want to use the controller together with the Arduino IDE again later, simply flash the program in the usual way. However, the ESP32 / ESP8266 then forgot that it ever spoke MicroPython. Conversely, every Espressif chip that contains a compiled program from the Arduino IDE or the AT firmware or LUA or ... can easily be provided with the MicroPython firmware. The process is always as described here.

The construction and the first commissioning

The Baseversion

The basic level is very clearly equipped with just a few parts. The only active component is the ESP32. The possibility of eliciting PWM signals up to (theoretical) 40 MHz from the controller gave me the idea for this project. The targeted control of the output of the circuit can be done via an Android device, or recently also iPhone, with the help of an app that we will build with the App Inventor2 in the next blog episode. For this, too, the ESP32 provides everything we need for radio communication. Incidentally, an ESP8266 does not manage the high frequencies with PWM, so it has to be the big brother.

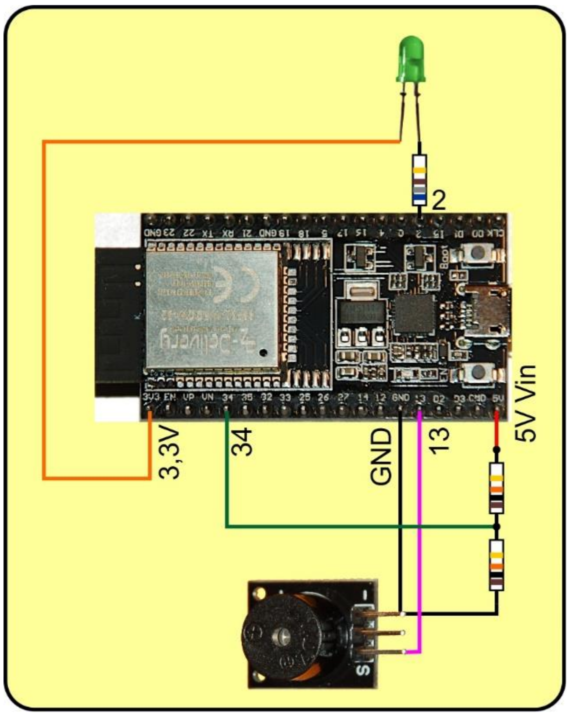


Abbildung 2: Gelsenschreck_Basisschaltung

The basic version of the circuit can be put together very quickly for first attempts. In order to be able to elicit sounds from it at the same time, I have hidden the commands that are responsible for the radio connection in the gelsnschreck.py program. So we don't have to worry about it at first. Instead of the pinched text, two lines have been inserted that enable operation via the USB cable. For the time being, the input command replaces radio transmission. After starting the modified program gelsnschreck_test.py the LED flashes long - briefly - briefly, this is the signal for operational readiness.

Here is the list of possible commands for the three operating modes wobbling, continuous tone and burst mode.

Modus	**Befehl**	**Bemerkung**
Wobbeln	w:f:<Startfrequenz in Hz>	lower frequency limit from which the pitch is increased.
	w:t:<Endfrequenz in Hz>	upper limit of the frequency sweep
	w:s:<Frequenzzuwachs in Hz>	Rate of increase in the throughput frequency
	w:d:<Verweildauer in ms>	Time during which the frequency is maintained
	w:start	switch on Tone
	w:stop	switch off Tone
Dauerton	c:f:<Frequenz in Hz>	Continuous tone frequency
	c:d:<Dutycycle in %>	Pulse to period ratio - of the output signal
	c:start	switch on Tone
	c:stop	switch off Tone
Burst	b:f:<Frequenz>	Frequency of the frequency packet and tone
	b:p:<Pulsdauer in s>	Duration of the frequency package

In the trial version of the program, these commands are entered using the PC keyboard. This enables us to quickly check extensions or changes to the program without having to wait for the wireless connection after each restart.

When the program starts, the following parameters are set to their start values.

```
startFreq=10000  # Startfrequenz Wobbeln
endFreq=20000    # Endfrequenz Wobbeln
step=10          # Zeitstufe Wobbeln in ms
delta=10         # Frequenzstufe beim Wobbeln
repeat=True      # Wobbeln wiederholen

freq=startFreq   # Continuous Freq.
duty=512         # Dutycycle in 1/1024; 512 entspricht 50%
pulse=5          # Burstdauer in s
```

The following start message appears in the terminal. The commands can then be entered.

```
ADC Initialized:  0
Socket established, waiting...
Kommando:b:p:1
from 10.0.1.230
Content = b:p:1
```

```
B-P:1
Kommando:b:f:4000
from 10.0.1.230
Content = b:f:4000
B-F:4000
Kommando:
```

These commands generate a frequency pulse of 4000Hz and a duration of one second.

Expansion stage 1

That's all? You hardly hear that. That's right, it wasn't very loud. That's why there is also the expansion stage 1. With a 3.3V control signal, the buzzer is only tickled a little on the stomach. And that's why we are now giving the circuit a small amplifier addition in the form of the six inverters in the CMOS component CD40106. It is operated with 12V and also forms the rather blurred curve into a clean square-wave signal through the Schmitt trigger functionality of the inverters.

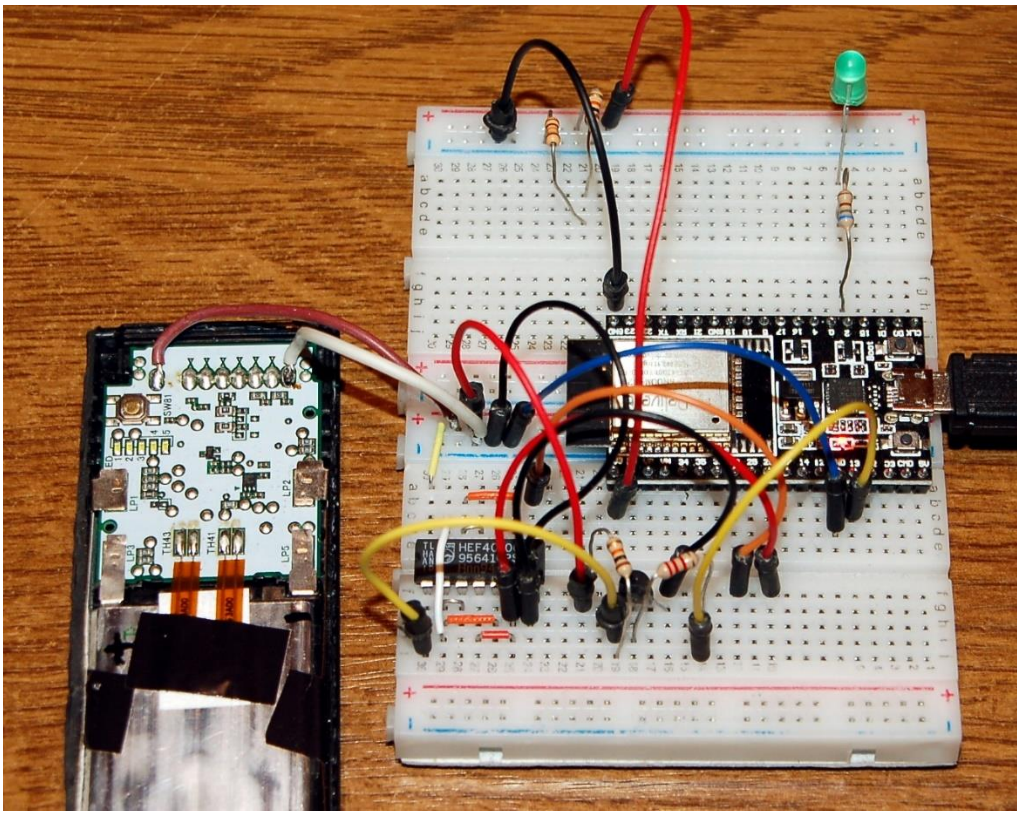


Abbildung 3: Ausbaustufe2 mit 12V-Li-Akku

This is what it looks like now, and that's behind it.

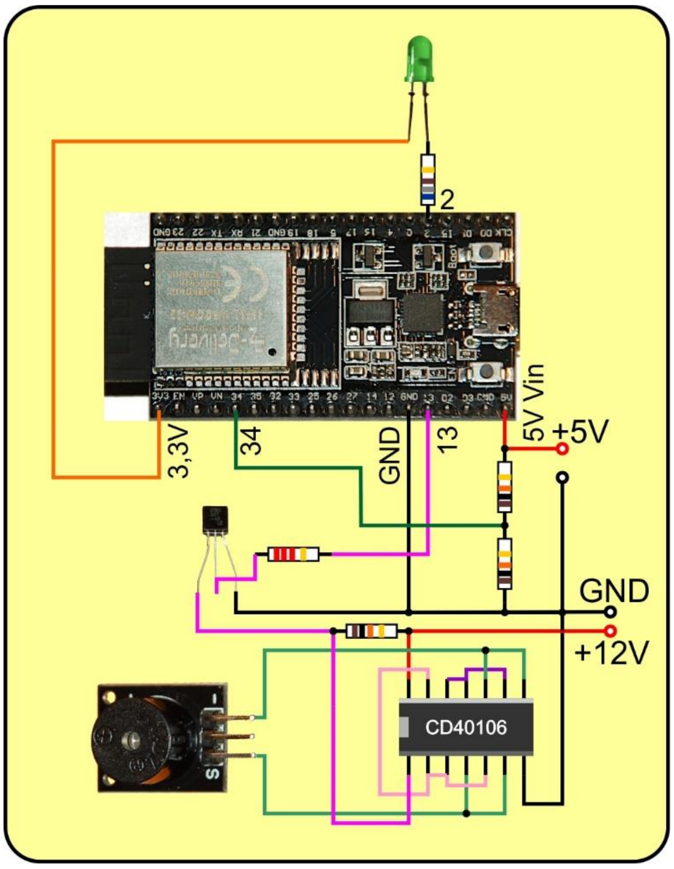


Abbildung 4: Ausbauversion 1

You want more input - please, welcome. Each of the triangles symbolizes one of the six Schmitt trigger levels. The point at the output means that the input signal is inverted. A 1 at the input becomes a 0 at the output and vice versa. We specifically use this function once. The unit 13/12 reverses the signal that comes from 1/2 so that the subsequent "output stages" are driven in push-pull.

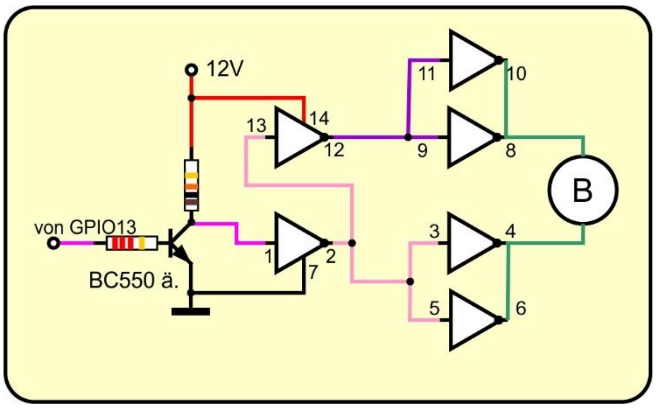


Abbildung 5: Gelsnschreck_CMOS-Verstärkerschaltung

And so that you can also assign the numbers to the connections, a look inside the CD40106 now follows.

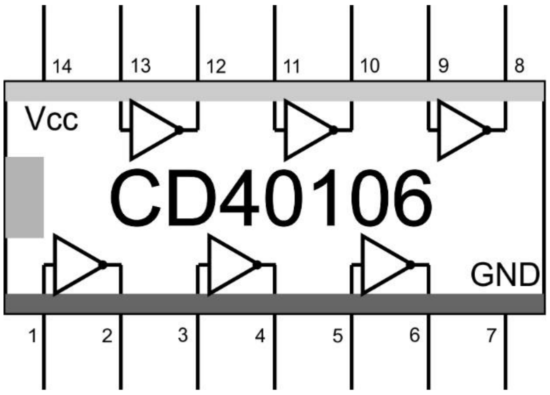


Abbildung 6: CMOS40106_Innenleben

The schematic representation of an inverter reveals that its output stage works in push-pull. If you switch two of these units against each other, controlled with opposite polarity, then this becomes a full bridge.

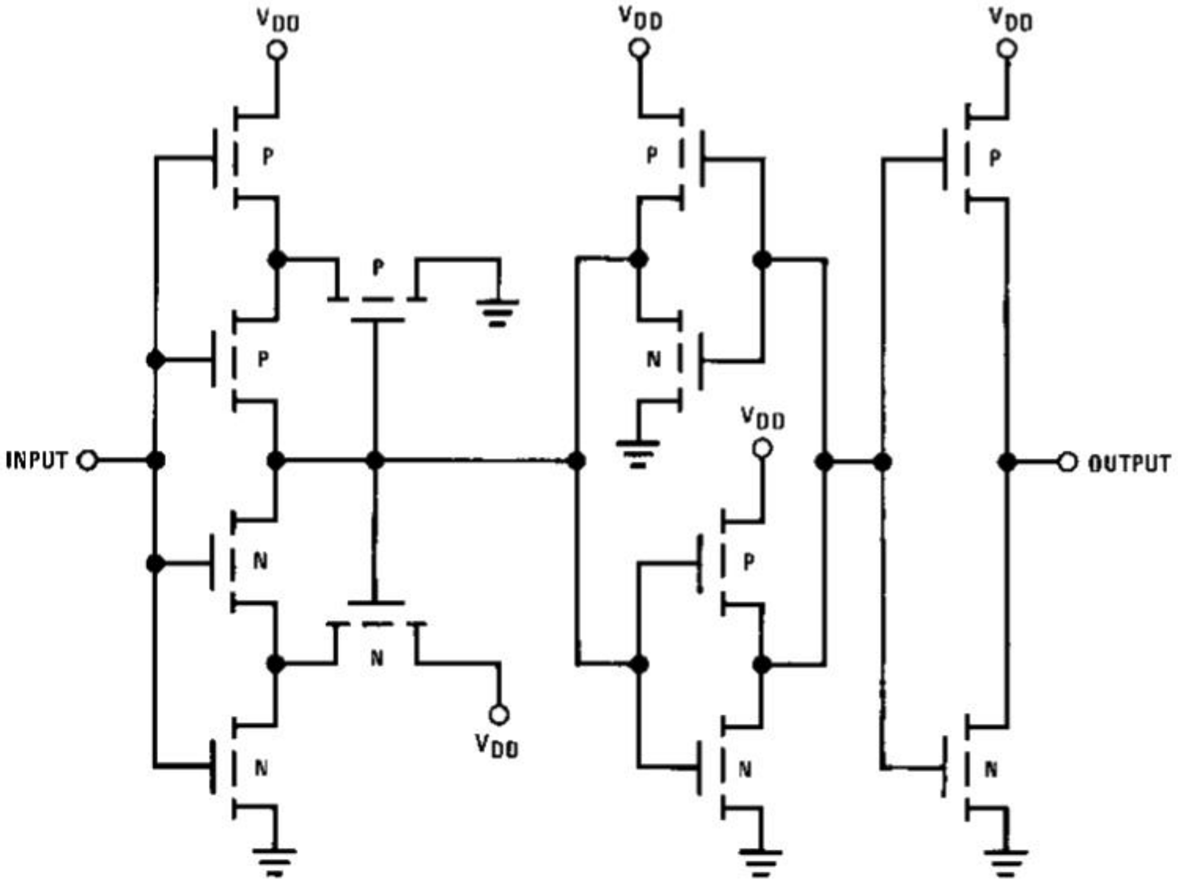


Abbildung 7: Inverterstufe

Let's look at the new parts of the circuit and clarify their function. The ESP32 delivers a maximum of 3.3V signal peaks. That would be sufficient as a logic level for a 1 (ULH = 2.9V) if we were to operate the 40106 at Vcc with 5V. But it is not enough for 12V operating voltage (ULH> 5.9V). That's why we need the transistor for level adjustment. The BC550 is satisfied with the 3.3V. Sufficient base current then flows through the 2.2 kΩ resistor so that the transistor switches through and pulls the 10 kΩ resistor to GND potential. If, on the other hand, 0V is applied to the base from the ESP32, then the BC550 blocks and the 12V of the operating voltage is applied to the collector.

The inverters work as amplifiers and formers, whereby the signal is inverted with each pass. This side effect does not bother us, however, it is necessary in one case, as I have already noted above. The first stage cleans the signal. The two following units in the same row are connected in parallel, which gives us more power.

The signal from the first stage is first intentionally inverted on the upper floor and then also amplified by the two parallel stages. At pins 6 and 8 we get a voltage that is not only switched on and off like with the ESP32, but here the polarity is reversed. The buzzer is therefore at the center taps of the half bridges in the output stages of the inverters.

Expansion state 2

Unfortunately, that didn't make the buzzer's signal overwhelmingly louder either. That might be enough for the immediate area. Nevertheless, we can get a lot more sound out of it if we engage a full-blown din instead of the buzzer. What is meant is a tweeter horn that works like the buzzer on a piezo basis, but emits significantly more phon.

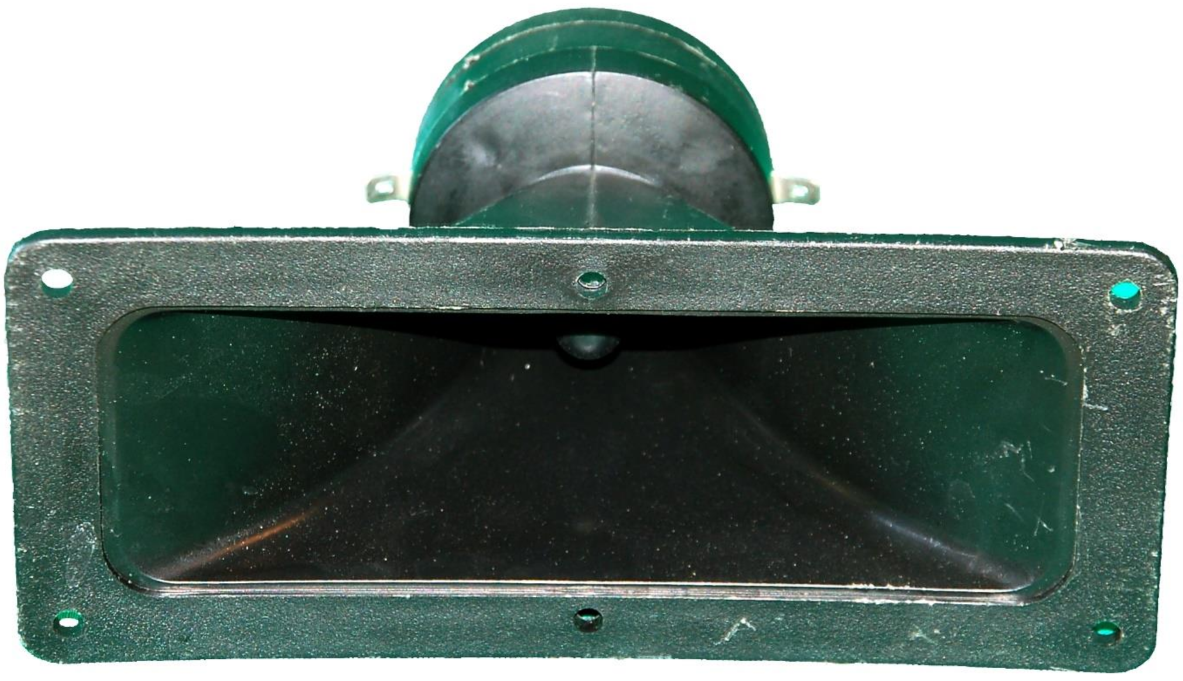


Abbildung 8: Hochtonhorn

Nothing needs to be changed on the previous circuit itself. We just connect the horn instead of the buzzer.

Did I promise too much? Just test it out on your neighbor's Wauzi when he decorates your front yard with his stepmines. Just play a little with the frequency. The horn can easily cope with up to 40 kHz, as does the ESP32, of course. By the way, 40kHz is also the communication band of bats. To locate flying insects and obstacles, they call out short bursts of a few hundredths of a second in a sequence of 5 to 10 packets. Bat detectors are used to make these calls audible to our ears. I will briefly talk about these devices at the end of this episode. The presence of these fast flying artists also helps us to decimate the unloved mosquitoes and moths.

Now that the function of the hardware is clear, let's examine the program more closely.

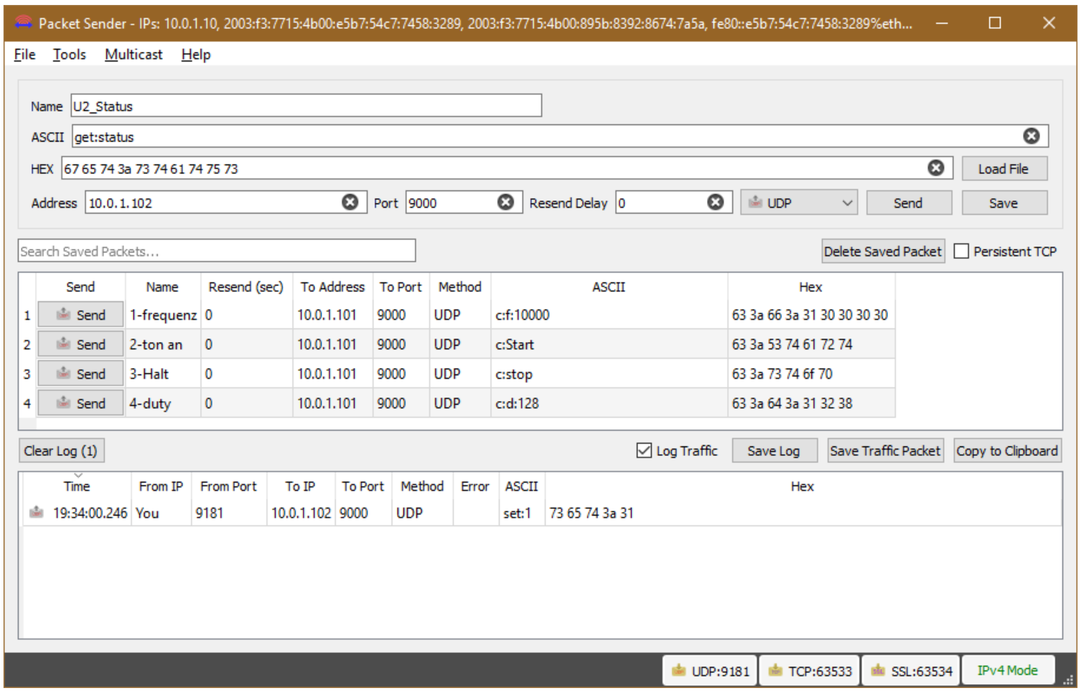
How the tones are created

The program is essentially divided into two parts. On the one hand there is the radio link via UDP, which receives commands from a client, and on the other hand the parser, which checks the syntax and content and initiates execution. Optionally, the UDP server also sends messages back to the client.

The radio connection can be established via an existing WLAN access point as well as via the in-house access point of the ESP32. The latter is advantageous where there is no local network.

The packetsender.exe tool can be helpful for testing the UDP connection. The program is installed on the Windows PC and serves there as a UDP client. With it we can send commands to the server on the ESP32, as we did at the very beginning via the terminal input. The feedback from the ESP32 is also displayed here. Using the program is not difficult and is intuitive. In the following figure only the IP address of the ESP32 has to be adapted to your home net, you can keep the port. The entry UDP: 9181 in the status line (at the bottom) is also important. Instead of the port number 9181, you can of course choose any other port number between 1024 and 65535.

Commands entered in the upper third of the window can be saved. This means that they are listed in the middle section and can be called up again using send. In the lower third you can see what you have sent and what the ESP32 is responding to.



Packet Sender - IPs: 10.0.1.10, 2003:f3:7715:4b00:e5b7:54c7:7458:3289, 2003:f3:7715:4b00:895b:8392:8674:7a5a, fe80::e5b7:54c7:7458:3289%eth...

Name: U2_Status
ASCII: get:status
HEX: 67 65 74 3a 73 74 61 74 75 73
Address: 10.0.1.102 Port: 9000 Resend Delay: 0 UDP

	Send	Name	Resend (sec)	To Address	To Port	Method	ASCII	Hex
1	Send	1-frequenz	0	10.0.1.101	9000	UDP	c:f:10000	63 3a 66 3a 31 30 30 30 30
2	Send	2-ton an	0	10.0.1.101	9000	UDP	c:Start	63 3a 53 74 61 72 74
3	Send	3-Halt	0	10.0.1.101	9000	UDP	c:stop	63 3a 73 74 6f 70
4	Send	4-duty	0	10.0.1.101	9000	UDP	c:d:128	63 3a 64 3a 31 32 38

Clear Log (1) ☑ Log Traffic

Time	From IP	From Port	To IP	To Port	Method	Error	ASCII	Hex
19:34:00.246	You	9181	10.0.1.102	9000	UDP		set:1	73 65 74 3a 31

UDP:9181 TCP:63533 SSL:63534 IPv4 Mode

Abbildung 9: Packetsender

But now to the program gelsnchreck_test.py.

```
#gelsnschreck_test.py
# ********************** Importgeschaeft ********************
import esp
esp.osdebug(None)
import os
import gc                       # Platz fuer Variablen schaffen
gc.collect()
try:
  import usocket as socket
except:
  import socket
import ubinascii
import network
from machine import ADC,Pin,I2C,PWM,Timer
from time import sleep,time,sleep_ms
import sys
#from bmp280 import BMP280
#from i2cbus import I2CBus
UbatPin=34
Ubat=ADC(Pin(UbatPin))
print("ADC Initialized: ",Ubat.read())
taste=Pin(0,Pin.IN)
blinkLed=Pin(2,Pin.OUT)
request = bytearray(160)
act=bytearray(30)
response=""
# Pintranslator fÃ¼r ESP8266-Boards
# LUA-Pins     D0 D1 D2 D3 D4 D5 D6 D7 D8
# ESP8266 Pins 16  5  4  0  2 14 12 13 15
#                  SC SD  FL L
i2c=I2C(-1,scl=Pin(21),sda=Pin(22))
#b=BMP280(i2c)

p = PWM(Pin(13))      # create PWM object from a pin
p.deinit()

startFreq=10000  # Startfrequenz Wobbeln
endFreq=20000    # Endfrequenz Wobbeln
step=10          # Zeitstufe Wobbeln in ms
delta=10         # Frequenzstufe beim Wobbeln
repeat=True      # Wobbeln wiederholen

freq=startFreq   # Continuous Freq.
duty=512         # Dutycycle in x/1023
pulse=5           # Burstdauer in s

T=Timer(0)

#*******************Variablen deklarieren ******************
# Die Dictionarystruktur (dict) erlaubt  die Klartextausgabe
# des Verbindungsstatus anstelle der Zahlencodes
connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "GOT_IP"
    }

flag = 0
#*****************Funktionen deklarieren *****************
def hexMac(byteMac):
  """
  Die Funktion hexMAC nimmt die MAC-Adresse im Bytecode
  entgegen und bildet daraus einen String fuer die Rueckgabe
  """
  macString =""
  for i in range(0,len(byteMac)):     # Fuer alle Bytewerte
    macString += hex(byteMac[i])[2:]  # ab Position 2 bis Ende
    if i <len(byteMac)-1 :            # Trennzeichen
      macString +="-"
  return macString

def blink(pulse,wait,inverted=False):
    if inverted:
        blinkLed.off()
        sleep(pulse)
        blinkLed.on()
        sleep(wait)
    else:
        blinkLed.on()
        sleep(pulse)
        blinkLed.off()
        sleep(wait)

def increment(tim):
    global freq,p
    freq=(freq+step if freq <= endFreq else startFreq)
    p.freq(freq)
    print(freq)

def shutOff(tim):
    p.deinit()

def parse(action):
    global p, startFreq,endFreq,step,duty,freq,repeat
    global delta,pulse
    art="E"
    act="_"
    value=-1
    action=action.upper()
    if action.find(":",1) !=-1 :
        art,rest=action.split(":",1)
        if art in ["W","C","B",]:
            if rest.find(":") != -1: # Befehl mit Parameter
                act,value=rest.split(":")
                value=int(value)
                if art=="W": # wobbeln
                    if act=="F": # Startfrequenz setzen
                        startFreq=value
                    elif act=="T": # Endfrequenz setzen
                        endFreq=value
                    elif act=="S": # Stufenzeit setzen
                        step=value
                    elif act=="D": # Stufenzeit setzen
                        delta=value
                elif art=="C":  # Continuous mode
                    if act=="F": # Frequenz setzen
                        freq=value
                        p.freq(freq)
                    elif act=="D": # Dutycycle setzen
                        duty=value
                        p.duty(duty)
                elif art=="B":
                    if act=="F": # Frequenz setzen und Start
                        freq=value
                        p.init()
                        p.duty(duty)
                        p.freq(freq)
                        T.init(mode=Timer.ONE_SHOT,period=\
                               pulse*1000,callback=shutOff)
                    elif act=="P": # Bustdauer setzen + Start
                        p.init()
                        p.duty(duty)
                        p.freq(freq)
                        pulse=value
                        T.init(mode=Timer.ONE_SHOT,period=\
                               pulse*1000,callback=shutOff)
            else: # Befehle ohne Parameter
                act=rest
                value=0
                if art=="W":
                    if act=="START": # wobbeln starten
                        p.init()
                        p.duty(duty)
                        p.freq(startFreq)
                        T.init(mode=Timer.PERIODIC,period=\
                               delta,callback=increment)
                    elif act=="STOP":
                        T.deinit()
                        p.deinit()
                elif art=="C":
                    if act=="START": # Dauerlauf starten
                        p.init()
                        p.duty(duty)
                        p.freq(freq)
                    elif act=="STOP":
                        p.deinit()
                else:
                    act="INVALID ACTION"
                    value=-3
        else:
            art=art+": ERROR"
            act="UNKNOWN COMMAND"
            value=-4
    else:
        if action=="E":
            print("ABGEBROCHEN DURCH USER")
            p.deinit()
            T.deinit()
            sys.exit()
        art=art+": ERROR"
        act="ARGUMENT MISSING"
        value=-5
    return(art,act,value)

# # *****************  Setup accesspoint **************
#
# nic = network.WLAN(network.AP_IF)
# nic.active(True)
# ssid="unit1"
# passwd="uranium238"
#
```

```
# # Start als Accesspoint
#
nic.ifconfig(("10.0.2.101","255.255.255.0","10.0.2.101","10.0.
2.101"))
#
# print(nic.ifconfig())
#
# # Authentifizierungsmodi ausser 0 werden nicht unterstuetzt
# nic.config(authmode=0)
#
# MAC=nic.config("mac") # liefert ein Bytes-Objekt
# # umwandeln in zweistellige Hexzahlen ohne Prefix und in
String decodieren
# MAC=ubinascii.hexlify(MAC,"-").decode("utf-8")
# print(MAC)
# nic.config(essid=ssid, password=passwd)
#
# while not nic.active():
#   print(".",end="")
#   sleep(0.5)
#
# print("Unit1 listening")
# # *****************  Setup accesspoint end ***************

# # ****************  Setup Router connection ***************
# mySid = mySid = 'YOUR_SSID'; myPass = "YOUR_PASSWORD"
# nic = network.WLAN(network.STA_IF)  #  erzeuge WiFi-Objekt
nic
# nic.active(True)  # Objekt nic einschalten
# #
# MAC = nic.config('mac')    # # binaere MAC-Adresse abrufen
und
# myMac=hexMac(MAC)          # in eine Hexziffernfolge
umgewandelt
# print("STATION MAC: \t"+myMac+"\n") # ausgeben
# # Verbindung mit AP im lokalen Netzwerk aufnehmen,
# # falls noch nicht verbunden
# # connect to LAN-AP
# if not nic.isconnected():
#   # Geben Sie hier Ihre eigenen Zugangsdaten an
#   # Zum AP im lokalen Netz verbinden und Status anzeigen
#   nic.connect(mySid, myPass)
#   # warten bis die Verbindung zum Accesspoint steht
#   print("connection status: ", nic.isconnected())
#   while not nic.isconnected():
#     blink(0.8,0.2,True)
#     print("{}.".format(nic.status()),end='')
#     sleep(1)
# # Wenn bereits verbunden, zeige Verbindungsstatus & Config-
Daten
# print("\nconnected: ",nic.isconnected())
# print("\nVerbindungsstatus: ",connectStatus[nic.status()])
# print("Weise neue IP zu:","10.0.1.101")
# nic.ifconfig(("10.0.1.101","255.255.255.0","10.0.1.20", \
#               "10.0.1.100"))
# STAconf = nic.ifconfig()
# print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",\
#       STAconf[1],"\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
#
# # *************  Setup Router connection end ***************


# ----------------- Server starten --------------------------
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('', 9000))
print("Socket established, waiting...")
s.settimeout(2.0) # timeout, damit 'while True' durchläuft
blink(2,0.5,True)
blink(0.3,0.8,True)
blink(0.3,0.8,True)
blink(0.3,1.5,True)
# ------------------- Serverschleife ------------------------
while True:
    try:
        r=""
        # recvfrom wird nach 2 Sec. mit einer Exception
abgebrochen
        # Zwischenzeitlich eintreffende Zeichen bleiben bis
zum
        # nächsten Durchlauf im Empfangsbuffer und werden dann
        # abgeholt.
        #
#         request, addr = s.recvfrom(256)
#         r=request.decode("utf8")
        r=input("Kommando:")
        addr="10.0.1.230"
        print('from {}\nContent = {}'.format(addr,r))
        if r!= "":
            job=parse(r)
            response=job[0]+"-"+job[1]+":"+str(job[2])
            print(response)
#             s.sendto(response,addr)
    except OSError as e:
        #print(e.args[0])
        pass
    if taste.value()==0:
        print("Mit Flashtaste abgebrochen")
        sys.exit()
    blink(0.1,0.9, inverted=True)
    #sleep(1)
```

After a few module imports and the definition of some global variables, a few functions are declared. The blink () function enables blinking to be called up easily via the LED.

The time control is done by the two timer service routines, increment () and shutoff (), for wobbling (aka automatic passage through a frequency band) and for burst (aka short pulse with a certain frequency).

The most comprehensive part of the program is the parse () function. We are checking here for the three range indicators W, C or B for wobble, continuous mode and burst mode. Then there are commands that send a parameter value and those that are supposed to start or end a process. If a valid command is found, the parser also takes care of its execution.

Two departments follow, one of which establishes the connection via the house network. For this part we have to know and enter the access data for the access point. The other department provides its own access point via the ESP32. As a network address you can choose one of the free ranges 10.X.X.X or 192.168.X.X. However, only one of the two options can be active at any one time. The selection is made by marking the block and commenting (Alt + 3) or uncommenting (Alt + 4). At the moment both blocks are commented out because we are talking about the input command with our ESP32.

The receive loop of the UDP server is set with a timeout of 2 seconds. This means that after this time the server loop continues, which means that any other commands can be executed while the server remains active. This is used to call the parser, if necessary, to send a message back to the client and to end the program by pressing the flash button on the ESP32. The latter option is important if the program runs autonomously as boot.py. At the moment the receive commands and the return of a message are commented out. The following two lines have been inserted for this purpose.

```
r=input("Kommando:")
addr="10.0.1.230"
```

On the subject of bats, there is another interesting approach with the ESP32. Because the controller can generate tones of a precisely known, stable frequency, the setup is also suitable as a generator for a bat detector. These devices record the calls of the animals via a suitable microphone. The signal is amplified and mixed with a second oscillation. The calls become audible to us as a kind of bugger when the mixed frequency, as the difference between the two frequencies, is within our audible range.

It wakes the dead

What is meant is the ultimate HF power amplifier, because you have certainly been waiting for expansion stage 3. Here is the blueprint without further comment. The supply voltage for the output stage can be up to 30 V. The construction includes the expansion stage 1 completely. So you can connect the ESP32 directly to the input. The PDF file with layout and assembly plan is available for download here. You should definitely treat the power transistors to a heat sink. The silicone mats, which you can see in the picture if you are using a continuous heat sink, are then important, otherwise there will be short circuits and the power supply will not like that at all. This is because the collector connections of both transistor types are also on the back of the housing. The collector of the BD135 (NPN) is at 30V, that of the BD136 (PNP) is at GND. Do you know the saying "Plus on mass, that bangs class"? Therefore the silicone mats as insulation or mica washers.

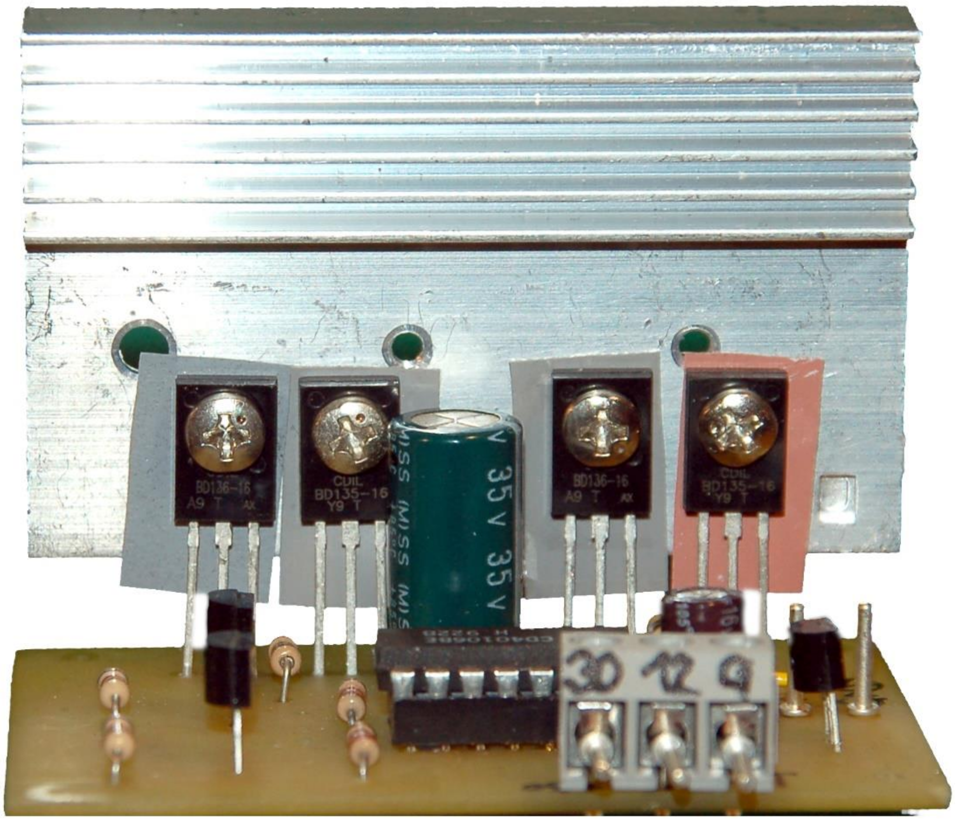


Abbildung 10: Verstärker

Ich wünsche schon mal viel Vergnügen beim Basteln und Programmieren mit den ungewöhnlichen Tönen und Tonlagen und freuen Sie sich auf Special-App zum Fernsteuern des Mückenschreckers.