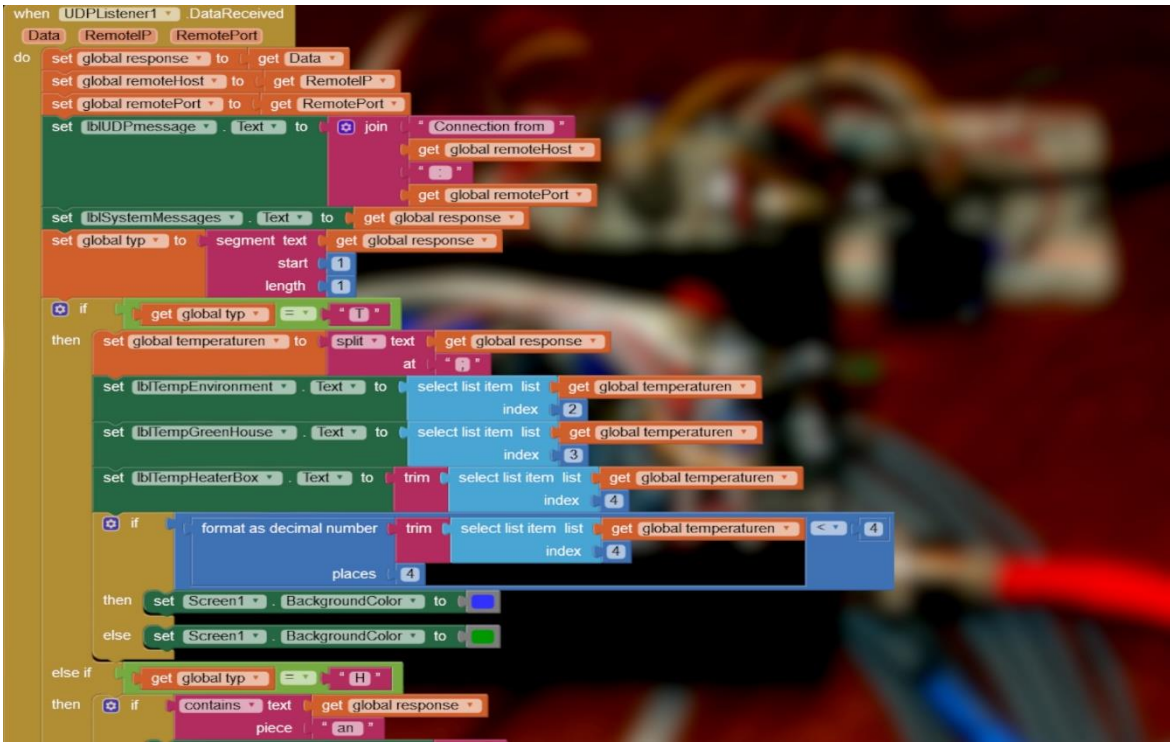


Abbildung 1: Android-App für die Steuerung

Diesen Beitrag gibt es auch als PDF-Dokument in deutsch und englisch
This episode is also available as pdf document in english and german.

Heute werden wir eine Android-App für die Steuerung des Freeze Guardian im Nahbereich um den WLAN-Router bauen. Für den Fernzugriff gibt es einen Webserver, der von (fast) jedem Browser angefragt werden kann. "fast" deswegen, weil Firefox verschiedentlich Mucken macht, wenn die Anfrage an einen Server erfolgt, der nicht mit dem HTTPS-Layer arbeitet, wie Ubuntu16.04 LTS auf meiner 32-Bit-Maschine. Ab Ubuntu 18 werden nur noch 64-Bitsysteme unterstützt. Aber, so wichtig sind unsere Daten nun auch wieder nicht, dass man die verschlüsselt übermitteln müsste. Damit willkommen beim 3. Teil der Reihe um den

Freeze Guardian – Wächter-App und WWW-Zugriff

Um einen grundlegenden Überblick über die angestrebte Netzwerkstruktur zu bekommen, starte ich mit einer Grafik. Im vorangegangenen Beitrag hatte der Linuxrechner bereits Konkakt mit dem ESP8266 via UDP aufgenommen. Auch der Windows-PC konnte sich mit dem ESP8266 bereits über das Programm Packetsender.exe unterhalten, ebenfalls über UDP. Meldungen vom ESP8266 wurden an beide PCs geschickt. Der ESP8266 läuft im Station-Modus und ist daher nicht selbst direkt als Accesspoint ansprechbar. Der ganze Traffic läuft also über den WLAN-Accesspoint – was den Funk angeht. In der Grafik sind das die hellgrünen Pfeile. Ein anderer Teil der Unterhaltungen passiert hauptsächlich oder ausschließlich über Kabel – die Pfeile in mittlerem Grün. Dunkelgrün und auch gemischt ist der Verkehr zwischen dem Smartphone und dem Webserver auf dem Linux. Der kleine, blassgrüne Pfeil steht für die Interprozesskommunikation zwischen dem UDP-Client und dem TCP-Webserver auf der Linuxkiste. Und genau mit Letzterem fangen wir heute an.

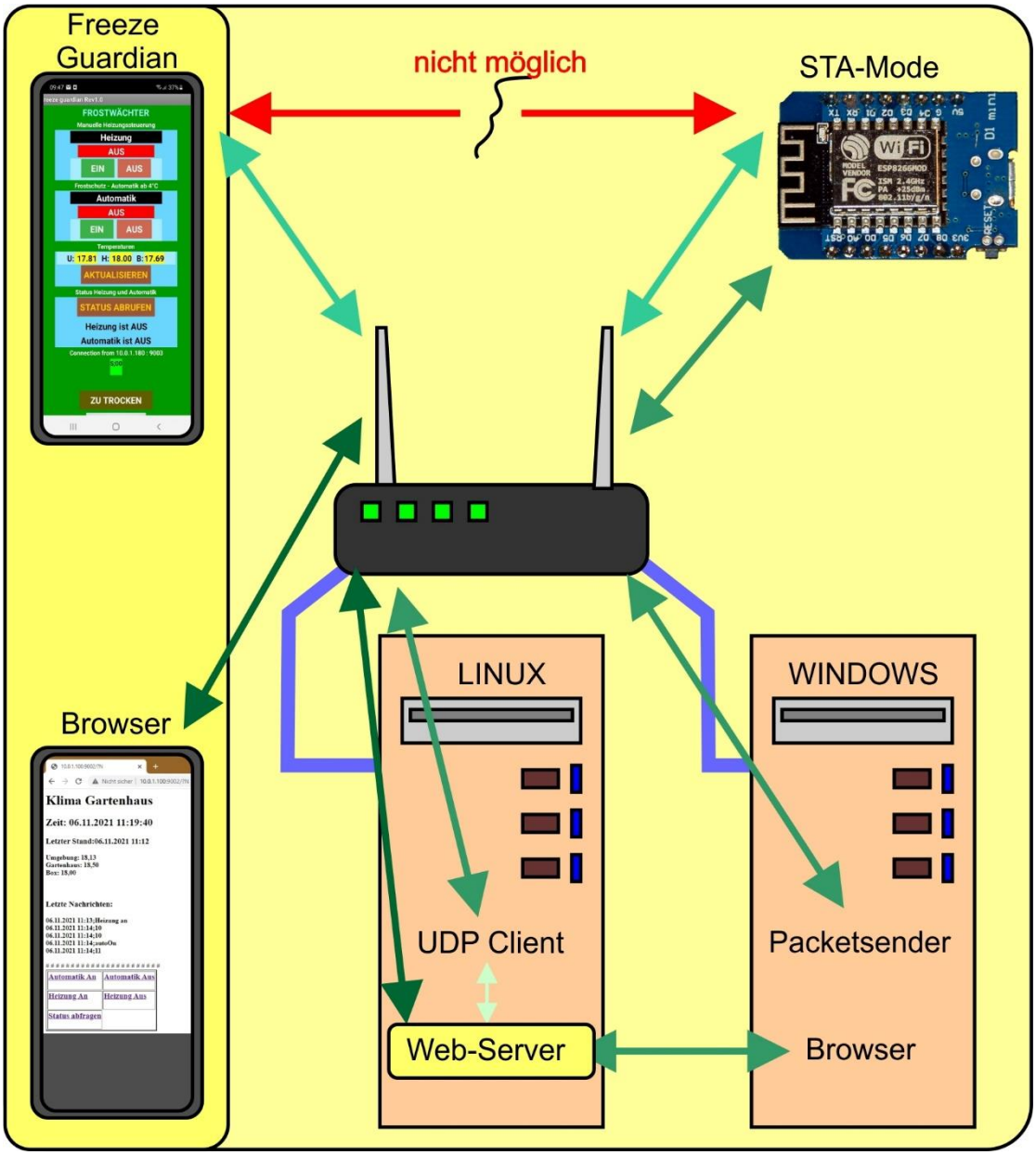


Abbildung 2: Netzstruktur

Der Webserver auf der Linuxkiste

Während der UDP-Client auf dem Linux Arbeiten zeitgesteuert und asynchron im Hintergrund erledigt, sorgt der Webserver unter TCP dafür, dass Daten vom ESP8266 auch außerhalb des lokalen Netzwerks zur Verfügung stehen. UDP ist ein schnelles, schlankes Protokoll, das aber verbindungsfrei arbeitet. Das hat zur Folge, dass der Datenverkehr nicht abgesichert ist und das wiederum will heißen, dass Datenpakete verlorengehen oder verfälscht werden können.

Wenn wir von außenhalb des LAN– WLAN–Bereichs auf Daten von unserem ESP8266 zugreifen wollen, muss das über TCP und eine ausfallsichere Verbindung geschehen.

Der Webserver auf der Linuxkiste stellt den Background dafür zur Verfügung. Von einem (fast) beliebigen Browser aus ist der Server erreichbar, im WLAN-Bereich direkt über die hausinterne, private IP-Adresse und weltweit unter Zuhilfenahme eines dynamischen DNS-Dienstes wie:

- YDNS. ...
- FreeDNS. ...
- Securepoint DynDNS. ...
- Dynu. ...
- DynDNS-Dienst. ...
- DuckDNS. ...
- Keine IP.

Über diese Dienste kann eine Klartext-URL bezogen werden (kostenfrei). Die vom Provider jeden Tag zugeteilte neue IP-Adresse wir durch den Dienst automatisch mit der URL verbunden. Unser Webserver ist damit stets unter derselben URL erreichbar.

Das "fast" habe ich eingangs bereits erläutert. Wie beim UDP-Client auf der Linux-Maschine fällt auch beim Webserver der Verbindungsaufbau zum WLAN-Accesspoint weg, da eine Kabelverbindung besteht. Der Socket ("10.0.1.111",9002) wird, mit einer Ausnahme, genauso eingerichtet, wie bei UDP. Einziger Unterschied ist die fett dargestellte Zeile. TCP arbeitet mit Streams, UDP mit Datagrams.

```
IPS="10.0.1.111"
portNumS=9002
print("Fordere Server-Socket an")
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
server.settimeout(1)
server.bind(('', portNumS))    # an lokale IP und Portnummer
9192 binden
server.listen(5)              # Akzeptiere bis zu 5 eingehende
Anfragen
print("Empfange Anfragen auf {}:{}".format(IPS,portNumS))

# ***************** Socket Aufbau Client ********************
IPC="10.0.1.111"
portNumC=9004  #
print("Fordere Client-Socket an")
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
client.settimeout(2)
client.bind(('', portNumC))    # an lokale IP und Portnummer
9192 binden
print("Sende Anfragen auf {}:{}".format(IPC,portNumC))
# ****************** Ziel: Klimaagent client9191
****************
targetPort=9001
target=("10.0.1.111",targetPort) # UDP-Client auf Linux111
```

Für die rechnerinterne Kommunikation richten wir den UDP-Socket ("10.0.1.111",9004) ein. Beide Sockets werden nichtblockierend aufgesetzt, dafür sorgt das Timeout. Sprechende Namen sind für die drei Adressen sehr sinnvoll, daher trägt auch die dritte Adresse einen solchen. Sie spricht den UDP-Client im Linux-PC als Ziel an. Dorthin sendet der Webserver seine internen Anfragen/Befehle und von dort bekommt der seine Informationen, die an den Browser weitergibt.

Wenn die Sockets bereit sind, geht's in die Endlosschleife des Servers. **server.accept**() schaut kurz nach, ob eine Verbindungsanfrage eingegangen ist. Liegt nichts vor, wird eine Exception geworfen, die von **try** abgefangen wird. Das entsprechende **except** ganz am Ende wird ohne ausführbaren Befehl mit **pass** durchlaufen. Es geht hier nur darum, dass das Programm durch die Exception nicht abgebrochen wird.

Liegt eine Anfrage eines Browsers vor, dann wird ein Kommunikationssocket **c** von **accept** zurückgegeben und die Adresse **addr** des anfragenden Clients. Über den Socket **c** wird die restliche Kommunikation abgewickelt, **server** ist damit wieder frei für weitere Anfragen. Fünf davon können gleichzeitig bedient werden, so haben wir es oben festgelegt.

```
while 1:                        # Endlosschleife
    try:
      c, addr = server.accept()  # Web-Anfrage entgegennehmen
      print('Got a connection from {}:{}\n'.\
             format(addr[0],addr[1]))
      rawRequest=c.recv(1024)  # (A)
      # rawrequest ist ein bytes-Objekt
      # und muss als string decodiert
      # werden, damit string-Methoden
      # darauf angewandt werden koennen.
      try:
        request = rawRequest.decode("utf-8")  # (B)
        getPos=request.find("GET /")
        if request.find("favicon")==-1:  # (C)
            print("**********************************")
            print("Position:",getPos)
            print("Request:")
            print(request)
            print("**********************************")  # (D)
            pos=request.find(" HTTP")
            if getPos == 0 and pos != -1:  # (E)
                query=request[5:pos]  # (F)
                print("*****QUERY:{}******\n\n".format(query))
                response = web_page(query)  # (G)
                print("---------\n",response,"\n-----------")
                c.send('HTTP/1.1 200 OK\n'.encode())  #(H)
                c.send('Content-Type: text/html\n'.encode())
                c.send('Connection: close\n\n'.encode())
                c.sendall(response.encode())
            else:   # (J)
                print("############\nNOT HTTP\n############")
                c.send('HTTP/1.1 400 bad request\n'.encode())
```

```
        else:
            print("favicon request found")
            c.send('HTTP/1.1 200 OK\n'.encode())
      except:
        request = rawRequest
        c.send('HTTP/1.1 200 OK\n'.encode())
      c.close()
    except:
        pass
```

zu (A)
Wir holen von c die Anfrage, ein Bytesobjekt, das mehr Overhead als Nutzdaten enthält und wandeln es in einen String um (B), der leichter zu handeln ist.

zu (C)
Die nervige Angewohnheit von Browsern, eine Datei favicon.ico anzufordern, führt zu Problemen und wird durch die if-Konstruktion abgeschmettert.

zu (D)
Diese print-Befehle und alle folgenden tragen nur dazu bei, die Funktion des Servers im Terminalfenster zu verfolgen und zu kontrollieren. Sie können problemlos entfernt werden, wenn alles perfekt läuft.

zu (E)
Steht ganz am Beginn der Anfrage ein "GET " und etwas weiter hinten ein " HTTP", dann handelt es sich möglicherweise um eine solche, die der Server beantworten sollte. Anfragen dieser Art haben eine der folgenden Formen.

GET / HTTP/1.1
Host: 10.0.1.100:9002
Connection: keep-alive

GET /?**N** HTTP/1.1
Host: 10.0.1.100:9002
Connection: keep-alive
……

oder
GET /?**B;heizenAn** HTTP/1.1
Host: 10.0.1.100:9002
Connection: keep-alive
……

Die fett dargestellten Teile enthalten die Befehle an den Server, die wir herausfiltern (F).

zu (G)
Die Dekodierung übernimmt wie immer die Parserfunktion, die auch gleichzeitig für das Erstellen der Webpage verantwortlich ist und deshalb auch so heißt, **web_page**(). Wir übergeben ihr den geparsten Befehl **query**. response erhält den zurückgegebenen Webseitentext.

zu (H)
Es folgt die Ausgabe des HTML-Headers und des Seitencodes.

zu (J)
Entsprach die Anfrage nicht unseren Erwartungen, wird 400 "bad request" zurück an den Browser gesandt.

Gehen wir zum Parser weiter.

zu (K)
Das Modul time bietet in CPython im Vergleich zu MicroPython etwas andere Methoden an. So liefert die Methode strftime() die Angaben zu Datum und Uhrzeit in Verbindung mit einem frei definierbaren Stringgerüst. Hier zum Beispiel "Zeit: 06.11.2021 09:14:01\n".

zu (L)
Wir fordern die zuletzt abgespeicherten Temperaturen vom UDP-Client am Linux an und bestimmen die Länge des Befehlsstrings.

zu (M)
Ist die Befehlslänge 0 oder 1, dann liefert der else-Zweig nur die Temperaturen zurück.

10.0.1.100 oder (laenge=0)
10.0.1.100/ oder (laenge=0)
10.0.1.100/**?** (laenge=1)

Sind aber 2 und mehr Zeichen enthalten, dann sollte dem Fragezeichen entweder ein "N" folgen oder ein "B;befehl_an_den_UDP_Client". Ein "N" fordert neben den Temperaturen auch die letzten 5 Zeilen aus der Datei messages an.

zu (P)
Einem B muss ein Trennzeichen folgen (;) und dann einer der folgenden Befehle:

- getTemp
- sendTemp
- heizenAn
- heizenAus
- autoOn
- autoOff
- exit
- reboot

Diese Befehle gibt der Linux-UDP-Client an den ESP8266-UDP-Server weiter und wartet dann auf die Antwort, die in die HTML-Seite eingebaut wird.

```
def web_page(befehl):
    datum=strftime("<H2>Zeit: %d.%m.%Y %H:%M:%S</H2>\n") # (K)
    tempString=getTemperaturen()  #(L)
    werte=""
    laenge=len(befehl)
    print("Laenge:",laenge)
    if laenge>1:   # (M)
        befehl=befehl[1:]
        print("Befehl:{},\
               Länge:{}".format(befehl,len(befehl)))
        # Befehl ausführen
        # Antwort holen
        cmdReply="<BR>\n"
        if befehl[0]=="N":   # (N)
            tempString=getMessages()
        werte=tempString+cmdReply
        if befehl[0]=="B":   # (P)
            werte=doCommand(befehl[2:])
            werte=tempString+werte+cmdReply
    else:
        print("Temperaturen{}".format(tempString))
        werte=tempString
        # nur Temperaturen
    html1 = """<html>   # (Q)
    <head>
    <meta name="viewport" content="width=device-width,
          initial-scale=1">
    </head>
    <body>
    <h1>Klima Gartenhaus</h1>"""
    html1=html1+ datum + werte

    html2 = "# # # # # # # # # # # # # # # # # # # # # #"
    html3 ="""
    <table border=2 cellspacing=2>
    <tr>
    <td><a href='http://10.0.1.100:9002/?B;autoOn'>
        <H3>Automatik An</H3> </a></td>
    <td><a href='http://10.0.1.100:9002/?B;autoOff'>
        <H3>Automatik Aus</H3> </a></td>
    </tr>
    <tr>
    <td><a href='http://10.0.1.100:9002/?B;heizenAn'>
        <H3>Heizung An</H3></a></td>
    <td><a href='http://10.0.1.100:9002/?B;heizenAus'>
        <H3>Heizung Aus</H3> </a></td>
    </tr>
    <tr>
    <td><a href='http://10.0.1.100:9002/?B;status'>
```

```
        <H3>Status abfragen</H3> </a></td>
    </tr>
    </table>"""
    html9 = "</body> </html>"
    html = html1 + html2+html3+html9
    #print("Antwort: \n",html)
    return html
```

zu (Q)
Ab hier wird die HTML-Seite aufgebaut, zum Schluss zusammengefügt und zurückgegeben.

Eine Reihe weiterer Funktionen erledigt komplexere Aufgaben.

getTemperaturen() fordert vom Linux-UDP-Client den letzten Datensatz aus der Datei daten-gh an und baut daraus ein HTML-Schnipsel, das zurückgegeben wird.

```
def getTemperaturen():
    # Temperaturen holen
    print("Temperaturen anfordern")
    client.sendto("R".encode(),target)
    try:
        sleep(0.2)
        antwort,adr=client.recvfrom(256)
        antwort=antwort.decode().strip("\n")
        datum,envir,haus,box=antwort.split(";")
        temps="<H3>Letzter Stand:{}</H3><B>Umgebung:
{}<BR>\nGartenhaus: {}<BR>\nBox: {}<BR>
</B>\n"\
            .format(datum,envir,haus,box)
    except:
        temps="KEINE DATEN<BR>"
    return  temps # evironment,haus,box
```

Ähnlich verfährt **getMessages**() mit den angeforderten Daten aus der Datei **messages**.
Die Nachricht wird von flankierenden Newline-Zeichen (\n) gesäubert und der Temperaturstring an "_" von den Nachrichten getrennt. Der Temperaturstring wird an den ";" getrennt und Datum, Zeit und Temperaturwerte in HTML formatiert.

Die Nachrichten in **mesg** splitten wir an den Zeilenendezeichen "\n" auf und erzeugen daraus eine Liste **mesgs**, sie ist Grundlage für den HTML-String, den wir daraus in der for-Schleife bauen.

```
def getMessages():
    # Temperaturen und Nachrichten holen
    print("Temperaturen und Nachrichten anfordern")
    client.sendto("N".encode(),target)
    try:
        sleep(0.2)
        antwort,adr=client.recvfrom(512)
        antwort=antwort.decode().strip("\n")
```

```
        temp,mesg=antwort.split("_")
        datum,envir,haus,box=temp.split(";")
        temps="<H3>Letzter Stand:{}</H3><B>Umgebung:
{}<BR>\nGartenhaus: {}<BR>\nBox: {}<BR></B>\n"\
           .format(datum,envir,haus,box)
        mesgs=mesg.split("\n")
        mesg="<BR><BR><H3>Letzte Nachrichten:</H3><B>"
        for m in mesgs:
            mesg=mesg+m+"<BR>"
            print(mesg)
        mesg=mesg+"</B>"
        alles = temps+mesg
    except:
        alles="DATEN UNVOLLSTAENDIG<BR>"
    return  alles # evironment,haus,box
```

Die Funktion **doCommand**() fordert ebenfalls die Temperaturen an und gibt aber zusätzlich den Befehl im Parameter **cmd** an den Linux-UDP-Client weiter. Dessen Antwort warten wir ab und geben sie als HTML-Sequenz zurück.

```
def doCommand(cmd):
    # Temperaturen holen
    onOff=["aus","an"]
    print("Temperaturen anfordern")
    client.sendto(("B;"+cmd).encode(),target)
    try:
        sleep(1.0)
        antwort,adr=client.recvfrom(256)
        antwort=antwort.decode().strip("\n")
        result=antwort[2:]
        print("***************",antwort,"***************")
        if antwort[0]=="S":
            result="Heizung "+onOff[int(result[0])]+\
                   " - Automatik "+onOff[int(result[1])]
        temp="<H3>Befehl: {}</H3><B>{}<BR></B>\n".\
             format(cmd,result)
    except:
        temp="KEINE DATEN - TIMEOUT<BR>"
    return  temp # evironment,haus,box
```

Zum Senden und Empfangen mit den Befehlen sendto() und recvfrom() müssen wir beachten, dass in CPython bytes-Objekte gesendet und empfangen werden, keine Strings. daher muss beim Empfang das bytes-Objekt zum String decodiert und der String vor dem Senden encodiert werden. In MicroPython geschieht das beim Senden implizit.

Der an den Browser gesendete HTML-Code enthält neben den Temperaturdaten eine Tabelle, mit den Links, die Befehlssequenzen an den Webserver senden. Das erspart die Eingabe in die Adresszeile des Browsers von Hand und verhindert somit potenzielle Tippfehler.

```
<html>
    <head>
    <meta name="viewport" content="width=device-width,
       initial-scale=1">
    </head>
    <body>
    <h1>Klima Gartenhaus</h1><H2>
        Zeit: 06.11.2021 20:22:53</H2>
    <H3>Letzter Stand:06.11.2021 20:20</H3>
    <B>Umgebung:  18,50<BR>
       Gartenhaus:  18,75<BR>
       Box:  18,44<BR></B>
    # # # # # # # # # # # # # # # # # # # # # # #
    <table border=2 cellspacing=2>
    <tr>
    <td><a href='http://10.0.1.100:9002/?B;autoOn'>
        <H3>Automatik An</H3> </a></td>
    <td><a href='http://10.0.1.100:9002/?B;autoOff'>
        <H3>Automatik Aus</H3> </a></td>
    </tr>
    <tr>
    <td><a href='http://10.0.1.100:9002/?B;heizenAn'>
        <H3>Heizung An</H3></a></td>
    <td><a href='http://10.0.1.100:9002/?B;heizenAus'>
        <H3>Heizung Aus</H3> </a></td>
    </tr>
    <tr>
    <td><a href='http://10.0.1.100:9002/?B;status'>
        <H3>Status abfragen</H3> </a></td>
    </tr>
    </table></body> </html>
```

Abbildung 3: Webseite

Den Text der Linuxprogramme können Sie hier herunterladen:

client9001.py
converttemp.py
archivate.py
webserver.py

Die vorangegangenen Beiträge zum Thema Frostwächter finden Sie hier:

Teil 1 - Hardware und Programmierung des ESP8266 in MicroPython
Teil 2 – UDP-Client auf dem Linuxrechner (Ubuntu 16.04 LTS)
Teil 3 – Wächter-App und WWW-Zugriff

Der Vorteil des Webservers ist eindeutig die weltweite Erreichbarkeit, DDNS vorausgesetzt. Der Server kann übrigens auch darauf abgerichtet werden, die ganze Tagesdatei oder archivierte Dateien als Webseiten zu versenden. Dazu müssten wir nur den Webserver aus der jeweiligen Datei direkt lesen und jede Zeile als Textzeile in einem HTML-Gerüst versenden lassen. Es ist nicht schwer, versuchen Sie es! Es ist nicht mehr Knowhow nötig, als bereits in den beiden Dateien client9001.py und webserver.py steckt.

Die Android -App

Jetzt wenden wir uns aber der Android-App zu, Sie kann etwas, was die Webseite nicht kann. Sie spricht direkt mit dem ESP8266 und reagiert auf Alarmzeichen von diesem. Als Folge davon lassen uns ein Ansagetext und eine Sirene wissen, dass die Pflanzen gegossen werden sollen. Sehen wir uns an was für die App gebraucht wird, neben dem Android-Phone natürlich.

Für das Handy

AI2-Companion aus dem Google Play Store.

Für die Handy-App

http://ai2.appinventor.mit.edu
https://ullisroboterseite.de/android-AI2-UDP/UrsAI2UDP.zip
App-Inventor installieren und benutzen – Detaillierte Anleitung
Die fertige Entwicklung (freezeguardian.aia)
Die fertige App (freezeguardian.apk)

Wir erstellen die App mit Hilfe des Tools AppInvertor2, das unter der MIT-Licence als freie Software ohne Installation über einen Browser (z.B. Firefox) genutzt werden kann. Das bedeutet, dass die Anwendung eben nicht auf dem PC installiert werden muss, wenn eine WLAN-Verbindung zur Verfügung steht. Wie man damit umgeht, habe ich hier sehr detailliert beschrieben, deshalb gehe ich jetzt nicht näher darauf ein. Auch die Verwendung der UDP-Erweiterung für dieses Tool von Ullis Roboterseite wird dort genau erläutert.

Die Programmierung mit dem Appinventor ist ereignisgesteuert und objektorientiert. Das heißt wir reagieren mit unserem Programm auf einzelne Ereignisse wie Klicks

auf Buttons, den Ablauf eines Timers, den Empfang einer UDP-Nachricht und so weiter. Das alles passiert in einem Baukastensystem.

Starten wir den Appinventor in Firefox oder Chrome. Falls Sie die Software das erste Mal benutzen, empfehle ich dringend das Durcharbeiten meiner Einführung zu diesem Thema.

http://ai2.appinventor.mit.edu

Abbildung 4 zeigt das fertige Layout, das wir gleich erstellen wollen.

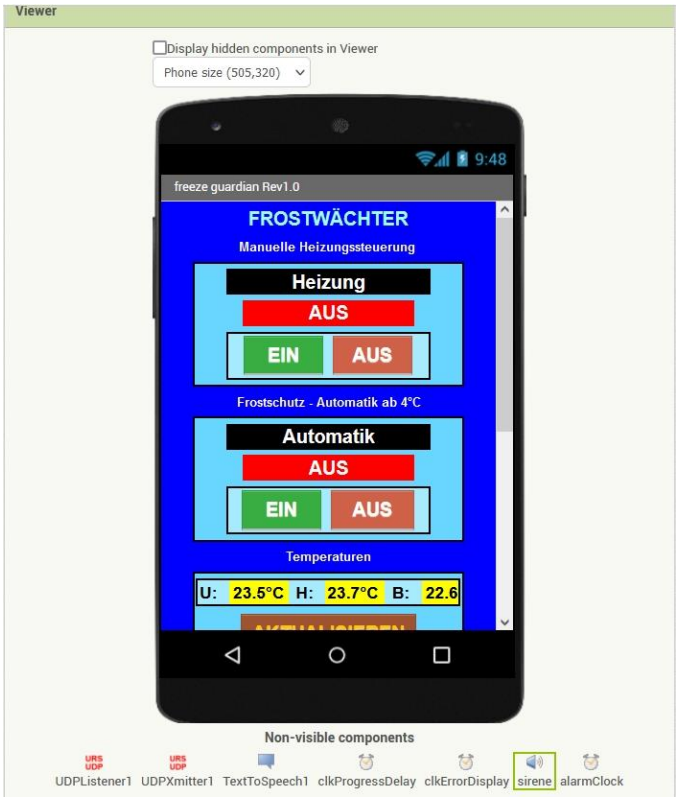


Abbildung 4: Das Viewer-Fenster

Aus der Spalte **Palette**, links neben dem **Viewer** holen wir uns die Objekte, die wir benötigen, durch Drag & Drop. "Forstwächter" ist ein **Label**, "Manuelle Heizungssteuerung" auch. Rechts neben dem Viewer haben wir die Fenster **Components** und **Properties**. **Components** zeigt die Hierarchie der angeordneten Elemente. In **Properties** werden die Eigenschaften eingestellt. "Frostwächter" hat die Schriftgröße 20 und die Farbe Hellblau, ist sichtbar und zentriert. Auch die Eigenschaft " horizontale Ausrichtung" des "**screen1**" ist zentriert.

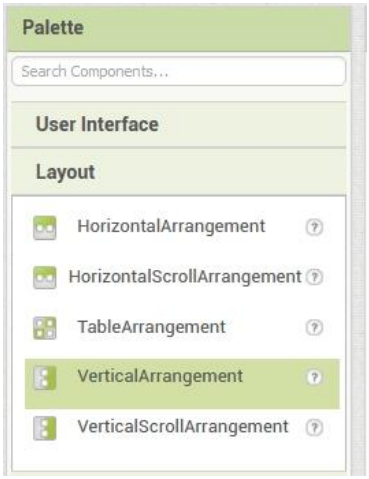
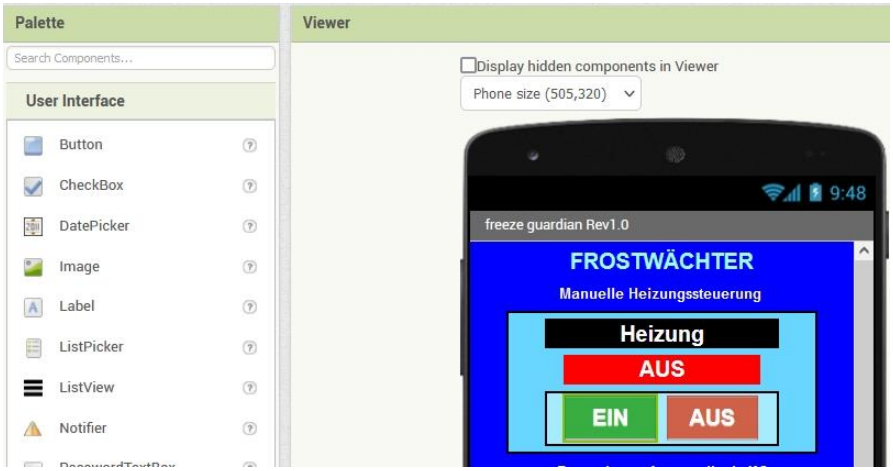


Abbildung 5: layout-Ordner

Die Struktur der Blöcke **Heizung** und **Automatik** ist identisch. Wir beginnen mit einem "**VerticalArrangement**". Dort hinein setzen wir ein "**HorizontalArrangement**".

Aus dem Ordner "**User-Interface**" holen wir zwei **Labels** und setzen sie über das horizontale Arrangement. In dieses hinein kommen zwei **Buttons**.



Über die Eigenschaften stellen wir die Breite der Elemente, die Schriftgröße, Schriftfarbe und den Hintergrund nach unseren Wünschen ein. Steht die Breite zum Beispiel auf "**Automatic**", dann passt sich die Breite des Feldes der Textbreite an. "Heizung" ist hier mit 60% eingestellt, "AUS" auf 50 und die Buttons auf 24. Die Höhe ist "Automatic". Alle Alignments sind "**center**". Außer den festen Farben kann man durch "**Custom**" aus einer Farbfläche RGB-Farben und Transparenz wählen.

Abbildung 6: Colors

Der "**sceren1**" ist "**Scrollable**" geschaltet, damit man in den unteren Bereich rollen kann.

Für die Temperaturen enthält ein "**VerticalArrangement**" ein horizontales mit 6 **Lables** und darunter einen **Button** "AKTUALISIEREN". "STATUS ABRUFEN" besitzt einen Button und darunter zwei Labels.

Abbildung 7: Unterer Screenbereich

UDP-Meldungen werden in dem darauffolgenden Label angezeigt, Rückmeldungen vom ESP8266 im grünen Labelfeld und Fehlermeldungen im roten. Der graue Button darunter zeigt Meldungen vom ESP8266, die die App ohne vorherige Anfrage an den Controller erreichen. Das sind Nachrichten vom Feuchtesensor. Immer dann, wenn der Zustand kippt und als Erinnerung nach Ablauf jedes Tages, werden wir vom Feuchtezustand der Pflanzerde informiert. Bei den "zu trocken"-Meldungen informiert eine Sprachansage darüber und eine Sirene weckt uns aus unseren Träumen von einem zufriedenen Pflanzenparadies. Der Alarm kann über den Button ausgeschaltet werden. "RESTART LISTENER" informiert uns über den Zustand des UDP-Empfängers.

Nicht erschrecken, wenn eine Fehlermaldung mit der Nummer 2 oder 6 erscheint. Das passiert hin und wieder während der Entwicklung beim Ändern von Eigenschaften und/oder dem Hinzufügen oder Entfernen von Elementen. Das Handy versucht dann die App neu zu starten, was mitunter geschieht, ohne dass vorher der UDP-Socket geschlossen worden ist. Der Start mit derselben IP und Portnummer geht dann schief, und die Verbindung zwischen Handy und Appinventor muss gekappt und neu hergestellt werden.

Neben den bisher aufgeführten sind noch eine ganze Reihe "unsichtbarer" Komponenten am unteren Bildrand im "Viewer" zu erkennen.

Abbildung 8: Unsichbare Elemente

Von links nach rechts sind das UDP-Empfänger und Sender, TextToSpeech für die Sprachausgabe, zwei Timer, eine Soundwiedergabe namens sirene und ein weiterer Timer für die Alarmwiederholung.

Sind alle Komponenten arrangiert, wechseln wir vom "Designer" zu den "Blocks", Linksklick ganz rechts in der Aktionsleiste

.

Abbildung 9: Menüs

Jeder Event wird nun durch ein Arrangement von entsprechenden Ziegeln mit einem Eventhandler versehen. Es gibt verschiedene kurze Handler, ein paar mittelprächtige und einen absoluten Monsterhandler. Letzterer entspricht dem Parser-Routinen in unseren Pythonprogrammen. Überhaupt treffen wir hier mehrfach auf die gleichen Strukturen wie in den MicroPython-Programmen, die Einrückungen durch die Klammern, die Reaktion auf Ereignisse etc.

Wir starten mit der Definition und Deklaration der benötigten globalen Variablen. Außerdem legen wir die Prozedur zum Einholen der Temperaturwerte an. Die Definitionsblocks der Variablen werden aus dem Ordner **Variables** gezogen, die Belegungen aus **Text**, auch **join** stammt von dort.

Die Prozedurklammer entnehmen wir dem Ordner **Procedures**. In Blocks ganz unten taucht **UDPXmitter1** auf, von da ziehen wir einen **call**-Block in den Viewer. Der soll die Textkonstante **getTemp** an den ESP8266 senden. Weil natürlich die Antwort nicht sofort zur Verfügung stehen wird, stellen wir den Wecker **clkProgressDelay** auf 1000ms und machen ihn scharf. Wir kommen später darauf zurück.

Abbildung 10: Blocks_Variablen und Funktionen

Jedes Programm braucht eine Initialisierungsphase, hier ist unsere. Mit dem Bildschirmaufbau passieren auch die Dinge in der **when screen1 initialize**-Klammer.

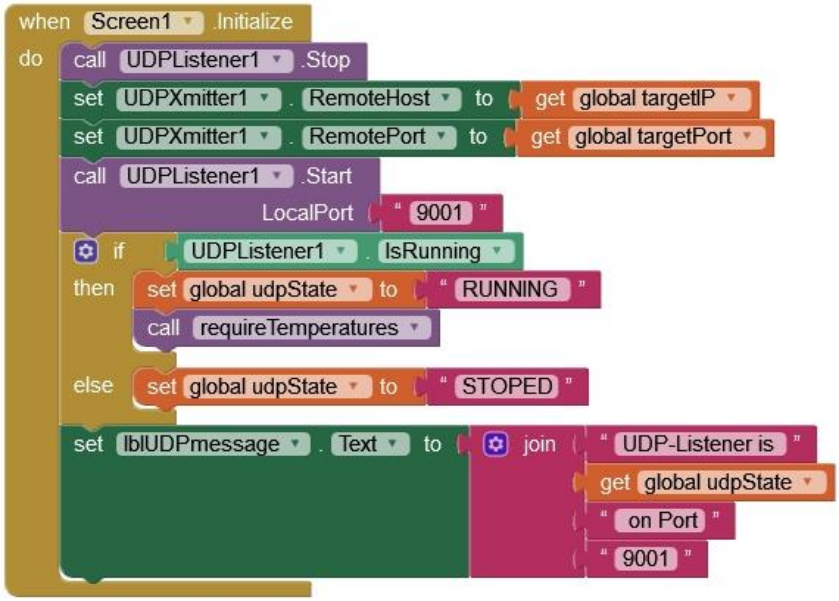


Abbildung 11:Blocks_Initialisierung

Der UDP-Empfänger wird angehalten, der Sender auf den ESP8266 als Ziel ausgerichtet und der Empfänger auf die Portnummer 9001 festgelegt. Dann fragen wir den Status des Empfängers ab und basteln daraus mit join eine Nachricht für das Label **lblUDPmessage**. Zugegeben, das sieht monströs aus, und in MicroPython geht das kürzer. Aber dafür entsteht daraus ein Produkt, das unter Android läuft, ohne dass wir eine Ahnung von der Programmierung unter diesem Betriebssystem haben müssen.

Weiter geht es mit den Reaktionen auf Buttons und Timer. Sechs der Klammern weisen dieselbe oder wenigstens eine ähnliche Struktur auf, die Behandlung der Buttons. Wie bei den anderen wird bei **when btnAutomatikOn.Click** der Inhalt der Fehleranzeige **lblError** gelöscht und der entsprechende Befehlsstring an den ESP8266 gesandt. Der Auftrag zum Aktualisieren der Temperaturen ist aufwendiger und wird durch die Prozedur **requireTemperatures** eingeleitet.

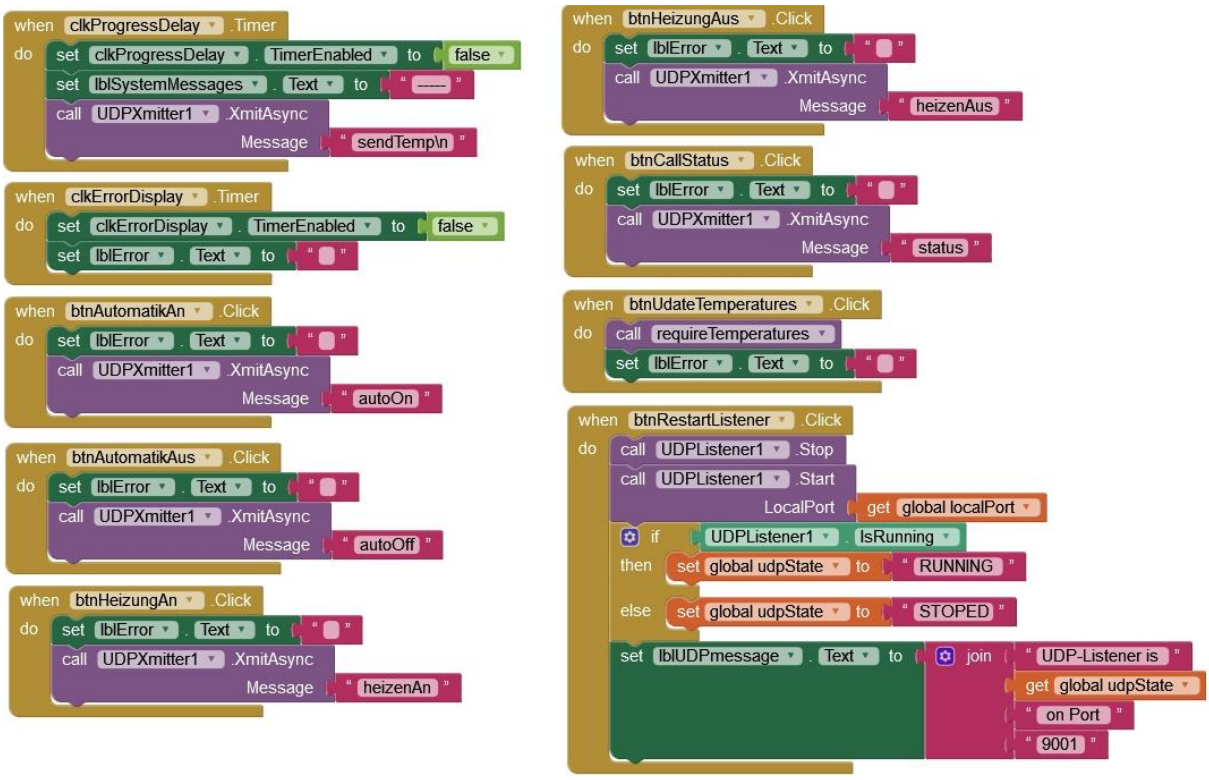


Abbildung 12: Blocks_Ereignisse

Ähnlich verhält es sich bei den Timern. In jedem Fall ist der Timer nach dem Ansprechen zu entschärfen. Ist **clkProgressDelay** abgelaufen, löscht der Handler die Systemmeldung in **lblSystemMessages** und erteilt den Befehl zum Senden der Temperaturwerte an den ESP8266. Fehlermeldungen werden mit Ablauf des Timers clkErrorDelay gelöscht.

Der Button RESTART LISTENER löst nahezu dieselben Aktionen aus wie bei der Initialisierung.

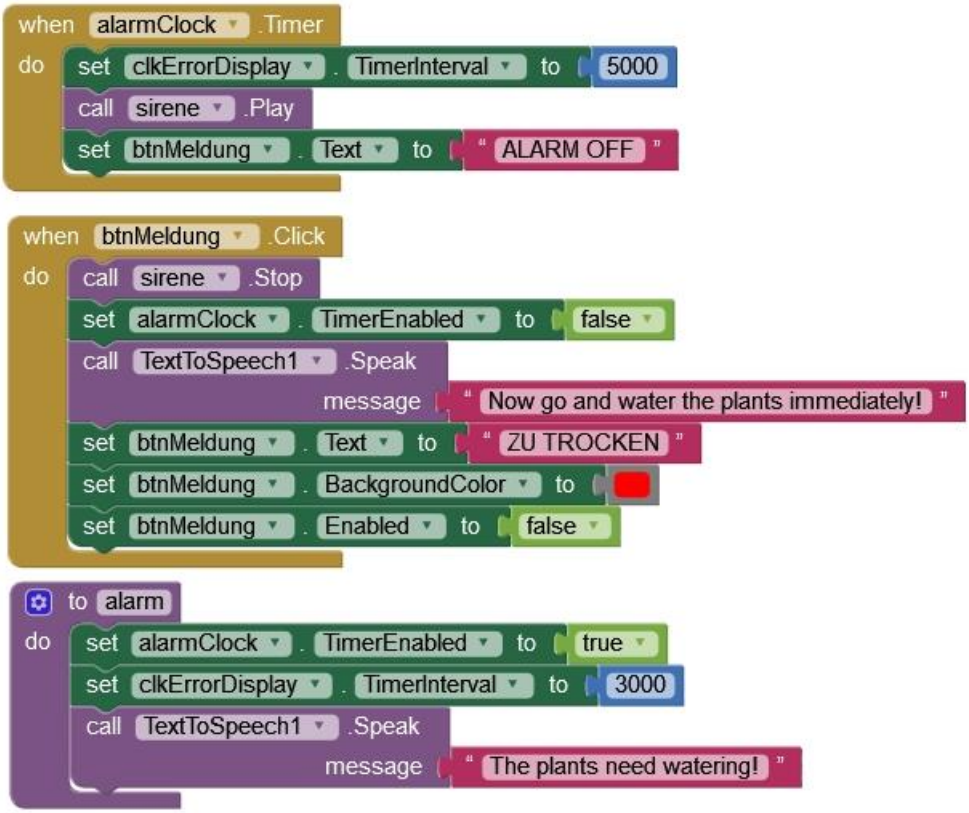


Abbildung 13: Blocks_Alarmbehandlung

Diese drei Blöcke behandeln den Alarmfall. Die Alarmauslösung obliegt der Prozedur **alarm**. Der Alarmtimer wird auf zunächst 3 Sekunden gesetzt und scharf gemacht. Die Sprachnachricht "The plants need watering" wird über den Handylautsprecher ausgegeben. Damit da wirklich Sprache herauskommt, setzen wir die Eigenschaften **Country** und **Language** bereits im Designer auf die richtigen Werte. Passen die nicht, kommt nur Stilleschweigen aus dem Lautsprecher.

Mit Ablauf des Alarmtimers setzt der Sound der Sirene ein. Dazu muss im Designer eine Sounddatei im MP3-Format zum Handy hochgeladen werden. Freie Sounddateien findet man im Internet haufenweise (zum Beispiel bei Salamisound).

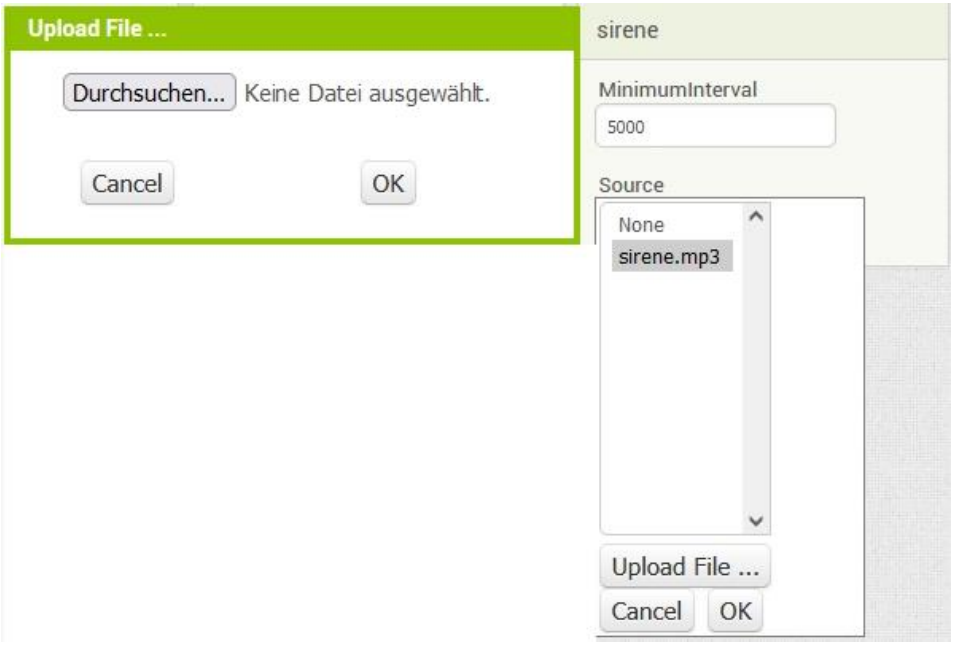


Abbildung 14: Sirene

Nur wenn der Alarm aktiv ist, ist auch der Button scharf geschaltet. Damit schalten wir die nervige Lärmtröte aus. Der Sound wird gestoppt, der Timer disabled, eine Erinnerung zum Gießen der Pflanzen ausgesprochen und der Button mit der Warnmeldung "ZU TROCKEN" versehen.

Der Parser

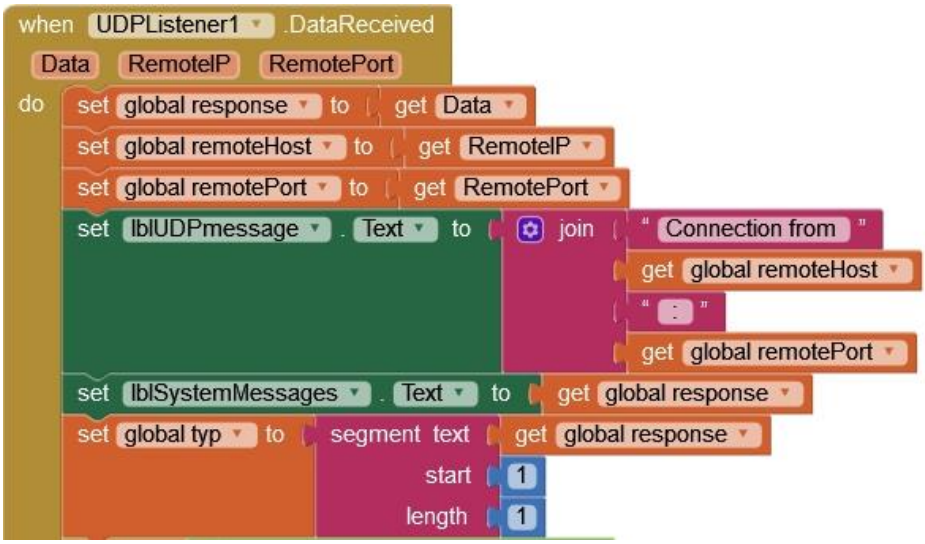


Abbildung 15: Blocks_Start Parsing

Das Parsing wird gestartet, wenn eine Nachricht im Buffer des UDP-Empfängers eintrifft. **when UDPListener.DataReceived** spricht dann an, holt die Nutzlast nach **response** und fragt die Sendeadresse ab. Diese wird im Label **lblUDPmessage** dargestellt. Im Label für Systemmeldungen wird der empfangene Text ausgegeben. Schließlich ermitteln wir den Typ der Nachricht, indem wir den ersten Buchstaben isolieren.

Mehrere if-elseif-Blöcke folgen und filtern die einzelnen Antworten des ESP8266 heraus, so wie wir es bereits in den MicroPython-Programmen getan haben. Der Typ T ist der interessanteste, weil er das Objekt Liste in der Anwendung zeigt.

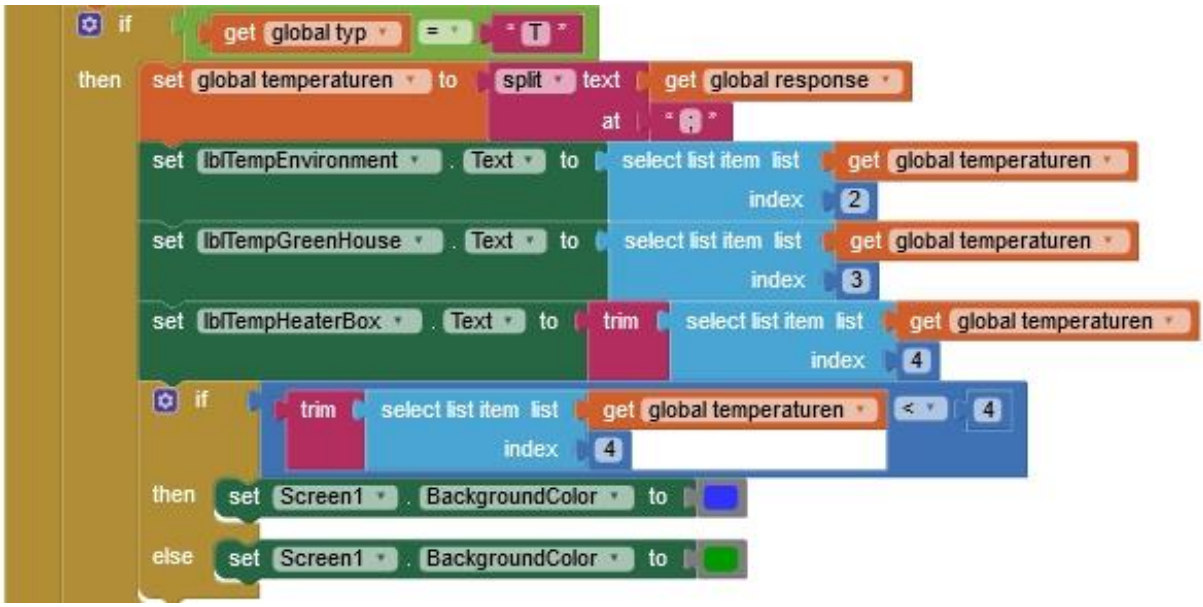


Abbildung 16 :Block_Temperaturen

Der Block **split text** teilt den Inhalt der Antwort vom ESP8266 an den Strichpunkten in einzelne Strings und weist diese der globalen Variablen **temperaturen** als Liste zu. Das entspricht dem MicroPython-Befehl

```
temperaturen = response.split(";")
```

Die Listenelemente weisen wir nun den Ausgabefeldern einzeln zu. Aus dem **Lists**-Ordner holen wir uns für diesen Zweck **select list item**. bei **list** klinken wir unsere Liste **temperaturen** ein und bei **index** den Index, der mit 2 beginnt, weil bei 1 der Typ steht. Bei der Boxtemperatur muss das "\n" am Ende abgeschnitten werden. Wenn die Temperatur in der Box kleiner als 4 °C ist, wird die Hintergrundfarbe des Bildschirms auf blau gesetzt, bei höheren Werten auf grün.

Die Blöcke für Heizung und Automatik sind quasi identisch und nicht sehr anspruchsvoll. Sie werden ausgeführt, wenn als **typ** ein "H" festgestellt wird. Das einzige neue Element ist **contains text** aus dem Ordner **Text**. Es untersucht den String in **text** auf das Vorkommen des Substrings in **piece**.

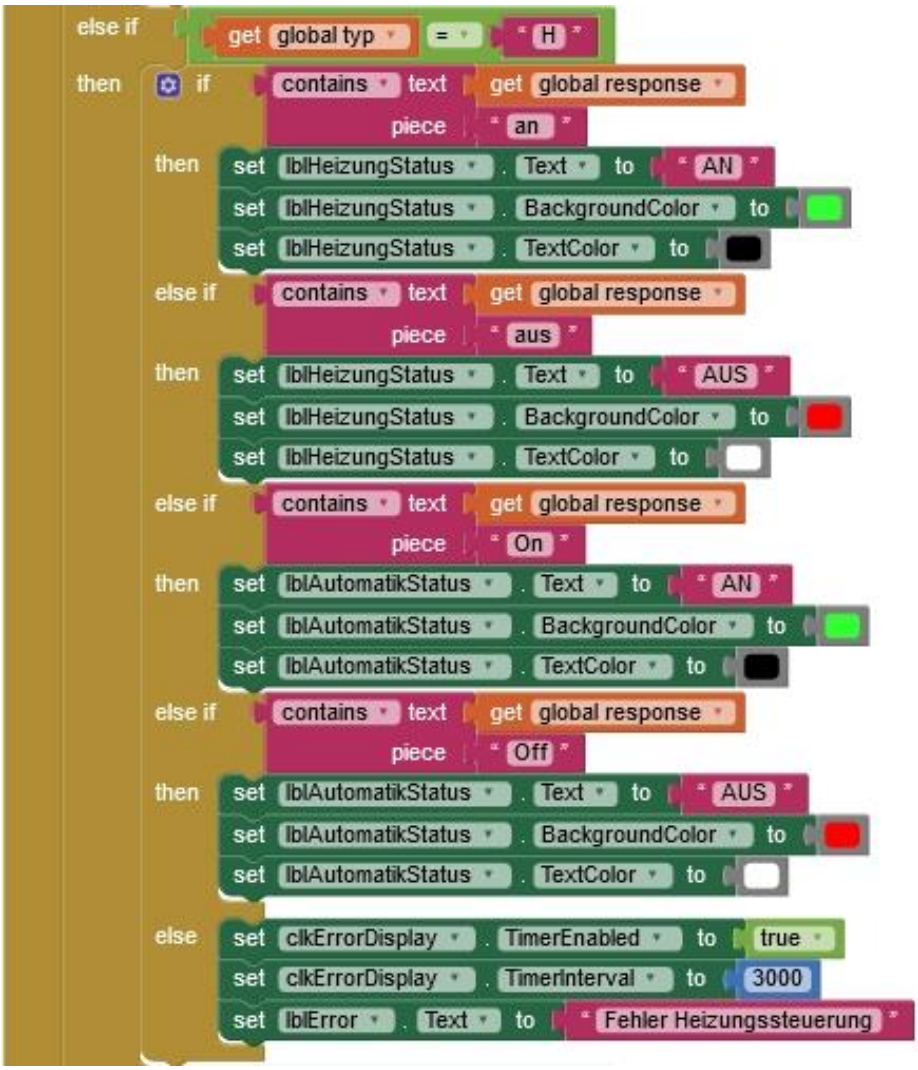


Abbildung 17: Block_Heizung

Der Rest des Parsers bringt auch nicht mehr bewegend Neues. Ich stelle die Blöcke nur der Vollständigkeit wegen dar.

Die Statusabfrage:

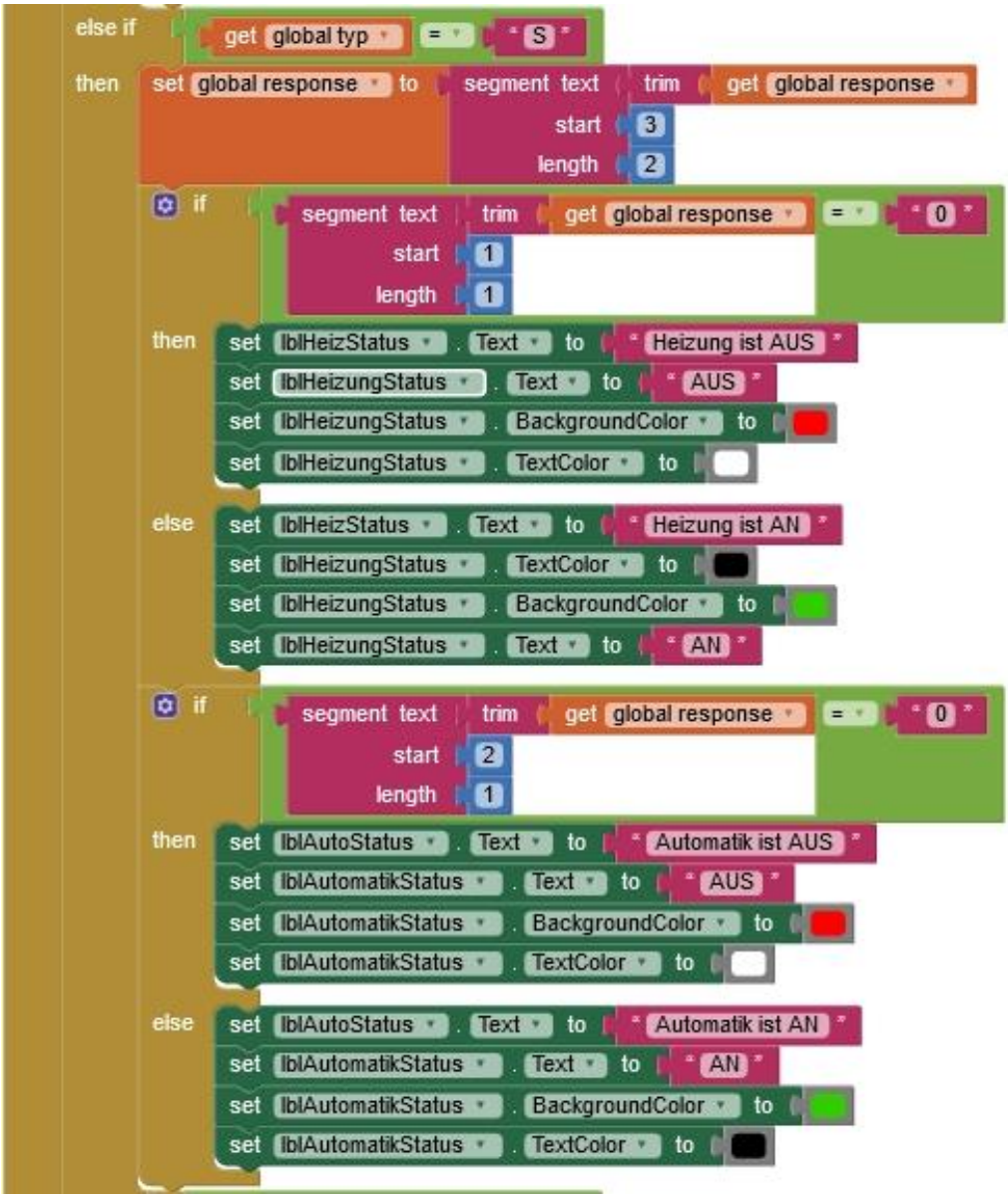
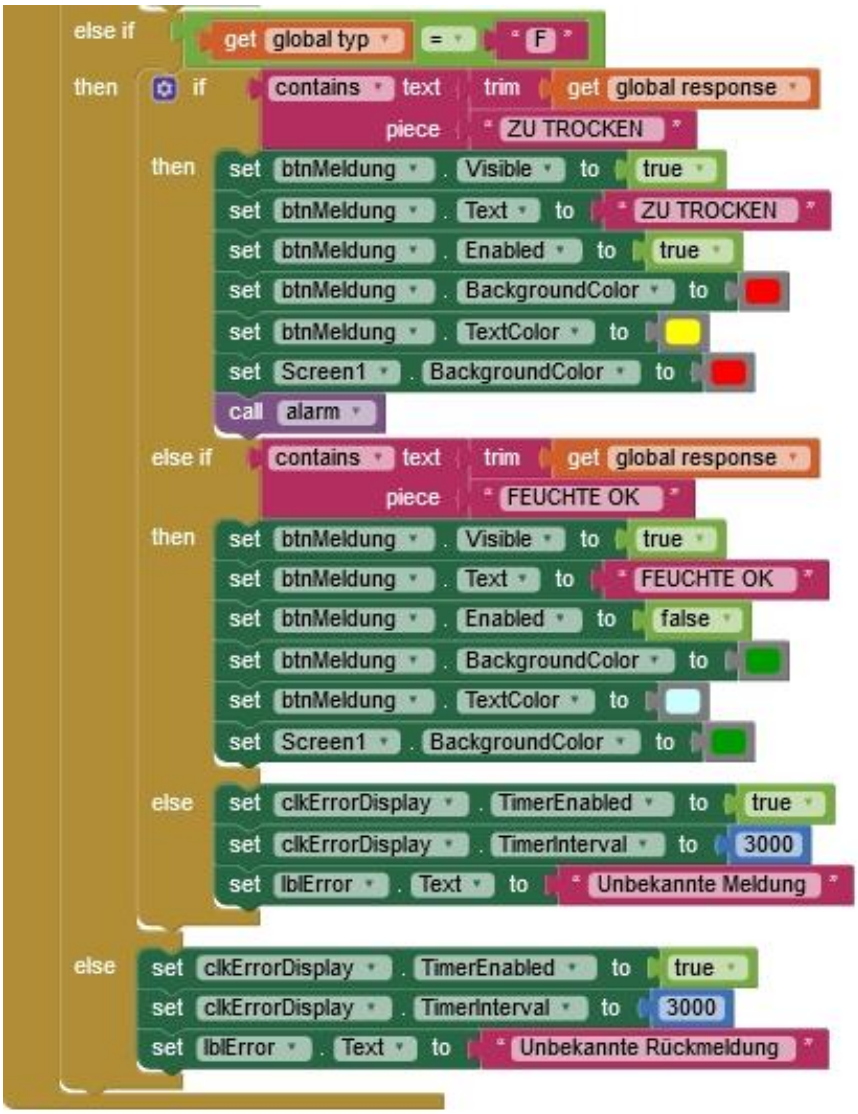


Abbildung 18: Block_Status

Im Rest wird beim Wechsel von feucht zu trocken die Prozedur **alarm** aufgerufen, die wir schon besprochen haben.



Jetzt liegt es an Ihnen, ob Sie im Designer die Objekte selbst platzieren und die Blöcke im Viewer selbst arrangieren möchten, oder es nicht erwarten können, die fertige Lösung in Funktion zu sehen. Eines müssen Sie aber beachten, damit es funktioniert, müssen Sie die Socketdaten, IP-Adresse und Portnummer, an Ihr Netz anpassen.

Ansonsten wünsche ich viel Vergnügen beim Programmieren weiterer Server auf der Basis von UDP oder TCP oder im Zusammenhang mit Appinventor 2. Das Rüstzeug dafür haben Sie mit dem Durcharbeiten dieses Projekts erhalten.

Die vorangegangenen Beiträge zum Thema Frostwächter finden Sie hier:

Teil 1 - Hardware und Programmierung des ESP8266 in MicroPython
Teil 2 – UDP-Client auf dem Linuxrechner (Ubuntu 16.04 LTS)
Teil 3 – Wächter-App und WWW-Zugriff