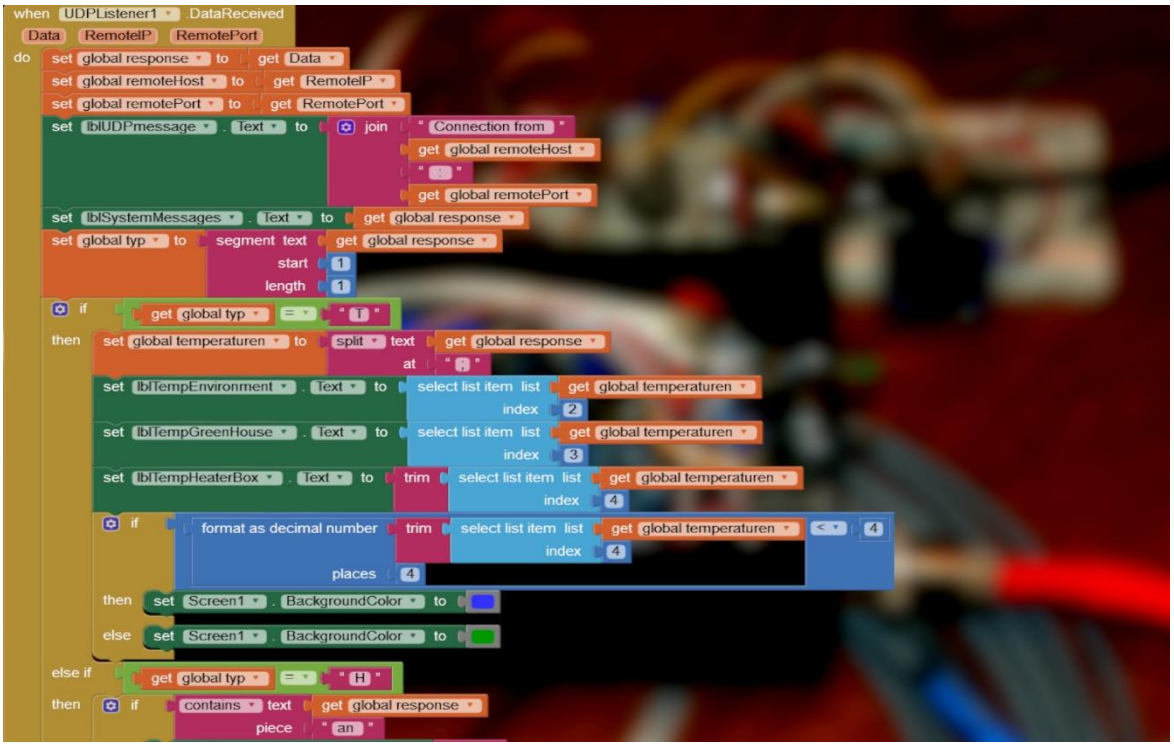


Abbildung 1: Android-App für die Steuerung

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

Today we are going to build an Android app to control the Freeze Guardian in close proximity to the WiFi router. There is a web server for remote access that can be requested from (almost) any browser. "almost" because Firefox makes various complaints when the request is made to a server that does not work with the HTTPS layer, such as Ubuntu16.04 LTS on my 32-bit machine. As of Ubuntu 18, only 64-bit systems are supported. But, our data is not so important that it has to be transmitted in encrypted form. So welcome to the 3rd part of the series

Freeze Guardian – Wächter-App und WWW-Zugriff

In order to get a basic overview of the desired network structure, I will start with a graphic. In the previous post, the Linux computer had already made contact with the ESP8266 via UDP. The Windows PC was also able to talk to the ESP8266 via the Packetsender.exe program, also via UDP. Messages from the ESP8266 were sent to both PCs. The ESP8266 runs in station mode and can therefore not be addressed directly as an access point. So all the traffic runs through the WLAN access point - as far as the radio is concerned. In the graphic these are the light green arrows. Another part of the conversation is mostly or entirely wired - the arrows in medium green. The traffic between the smartphone and the web server on Linux is dark green and mixed. The small, pale green arrow stands for the interprocess communication between the UDP client and the TCP web server on the Linux box. And that's exactly what we're starting with today.

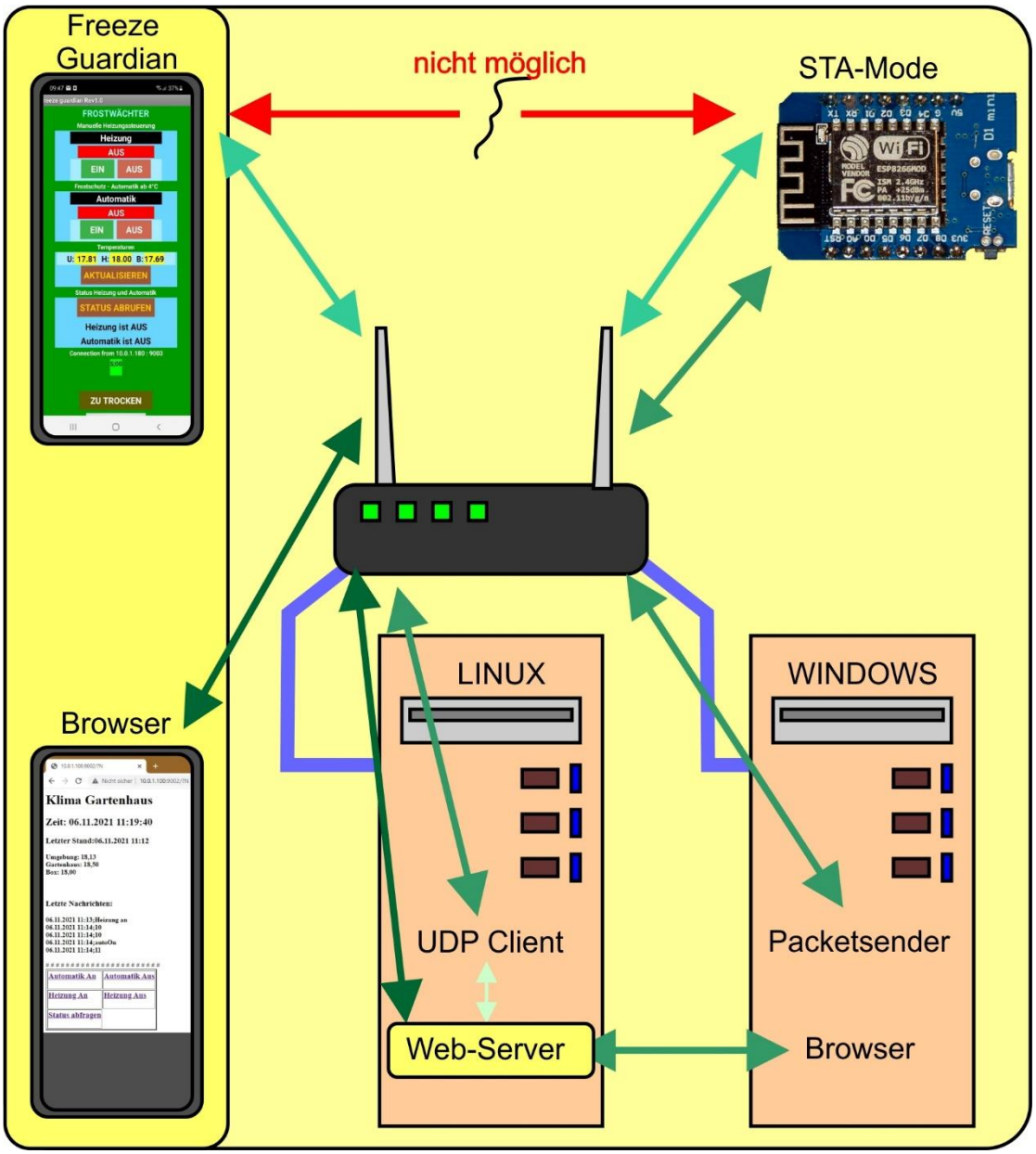


Abbildung 2: Netzstruktur

Webserver on Linux

While the UDP client on Linux does work time-controlled and asynchronously in the background, the web server under TCP ensures that data from the ESP8266 is also available outside of the local network. UDP is a fast, slim protocol that works without a connection. As a result, the data traffic is not secured and that in turn means that data packets can be lost or falsified.

If we want to access data from our ESP8266 from outside the LAN / WLAN area, this must be done via TCP and a fail-safe connection.

The web server on the Linux box provides the background for this. The server can be reached from (almost) any browser, in the WLAN area directly via the in-house, private IP address and worldwide with the help of a dynamic DNS service such as:

• YDNS. ...
• FreeDNS. ...
• Securepoint DynDNS. ...
• Dynu. ...
• DynDNS service. ...
• DuckDNS. ...
• No IP.

A plain text URL can be obtained from these services (free of charge). The new IP address assigned by the provider every day is automatically linked to the URL by the service. Our web server can therefore always be reached under the same URL.

I already explained the "almost" at the beginning. As with the UDP client on the Linux machine, there is no need to establish a connection to the WLAN access point with the web server, as there is a cable connection. The socket ("10.0.1.111", 9002) is set up in exactly the same way as with UDP, with one exception. The only difference is the line shown in bold. TCP works with streams, UDP with 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
```

We set up the UDP socket ("10.0.1.111", 9004) for internal computer communication. Both sockets are set up in a non-blocking manner, this is ensured by the timeout. Meaningful names are very useful for the three addresses, so the third address also has one. It addresses the UDP client in the Linux PC as the target. This is where the web server sends its internal requests / commands and from there it receives its information, which it forwards to the browser.

When the sockets are ready, the server goes into the infinite loop. server.accept () checks briefly whether a connection request has been received. If nothing is found, an exception is thrown, which is caught by try. The corresponding except at the very end is run through with pass without an executable command. The point here is that the program is not aborted by the exception.

If there is a request from a browser, a communication socket c is returned by accept and the address addr of the requesting client. The rest of the communication is handled via socket c, so the server is free again for further inquiries. Five of them can be operated at the same time, as we have specified above.

```
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
```

to (A)
We get the request from c, a byte object that contains more overhead than payload, and convert it to a string (B) that is easier to handle.

to (C)
The annoying habit of browsers to request a favicon.ico file leads to problems and is thrown out by the if construction.

to (D)
These print commands and all the following only help to monitor and control the function of the server in the terminal window. They can easily be removed if everything goes perfectly.

to (E)
If there is a "GET" at the very beginning of the request and an "HTTP" a little further back, then it may be one that the server should answer. This type of request takes one of the following forms.

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
……

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

The parts shown in bold contain the commands to the server that we filter out (F).

to(G)
As always, the decoding is carried out by the parser function, which is also responsible for creating the webpage and is therefore also called web_page (). We pass it the parsed query command. response receives the returned website text.

to (H)
This is followed by the output of the HTML header and the page code.

to (J)
If the request did not meet our expectations, 400 "bad request" is sent back to the browser.

Let's move on to the parser.

to (K)
The time module offers slightly different methods in CPython compared to MicroPython. The strftime () method provides the information on the date and time in connection with a freely definable string structure. Here for example "Time: 06.11.2021 09:14:01 \ n".

to (L)
We request the last saved temperatures from the UDP client on Linux and determine the length of the command string.

to the)
If the command length is 0 or 1, then the else branch only returns the temperatures.

10.0.1.100 or (length = 0)
10.0.1.100/ or (length = 0)
10.0.1.100/? (length = 1)

If it contains 2 or more characters, the question mark should either be followed by an "N" or a "B; befehl_an_den_UDP_Client". In addition to the temperatures, an "N" also requests the last 5 lines from the messages file.

to (P)
A B must be followed by a separator (;) and then one of the following commands:

- getTemp
- sendTemp
- heat on
- heat off
- autoOn
- autoOff
- exit
- reboot

The Linux UDP client forwards these commands to the ESP8266 UDP server and then waits for the response, which is integrated into the HTML page.

```
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
```

to (Q)
From here the HTML page is built up, then merged and returned.

A number of other functions handle more complex tasks.

getTemperaturen () requests the last data record from the daten-gh file from the Linux UDP client and uses it to build an HTML snippet that is returned.

```
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
```

GetMessages () proceeds similarly with the requested data from the messages file. The message is cleared of flanking newline characters (\ n) and the temperature string is separated from the messages by "_". The temperature string is attached to the ";" separated and date, time and temperature values formatted in HTML.

We split the messages in mesg at the end-of-line characters "\ n" and use them to create a list mesgs, which is the basis for the HTML string that we build from it in the for loop.

```
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
```

The doCommand () function also requests the temperatures and also forwards the command in the cmd parameter to the Linux UDP client. We wait for its answer and return it as an HTML sequence.

```
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
```

For sending and receiving with the sendto () and recvfrom () commands, we need to note that in CPython bytes objects are sent and received, not strings. therefore, the bytes object must be decoded to a string on receipt and the string must be encoded before sending. In MicroPython this happens implicitly when sending.

The HTML code sent to the browser contains the temperature data as well as a table with the links that send command sequences to the web server. This saves you having to enter it manually in the address line of the browser and thus prevents potential typing errors.

```
<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

You can download the text of the Linux programs here:

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

You can find the previous articles on the topic of Frost Guard here:

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

The advantage of the web server is clearly the worldwide accessibility, DDNS required. Incidentally, the server can also be trained to send the entire daily file or archived files as web pages. To do this, we would only have to read the web server directly from the respective file and have each line sent as a text line in an HTML framework. It's not difficult, try it! No more know-how is required than is already contained in the two files client9001.py and webserver.py.

The Android -App

But now we turn to the Android app, it can do something the website cannot. She speaks directly to the ESP8266 and reacts to alarm signals from it. As a result, an announcement text and a siren let us know that the plants should be watered. Let's take a look at what is needed for the app, in addition to the Android phone of course.

For the cell phone

AI2-Companion aus dem Google Play Store.

For the 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)

We create the app with the help of the AppInvertor2 tool, which can be used under the MIT license as free software without installation via a browser (e.g. Firefox). This means that the application does not have to be installed on the PC if a WLAN connection is available. How to deal with it I have described in great detail here, so I will not go into it in more detail now. The use of the UDP extension for this tool from UDP-Erweiterung für dieses Tool von Ullis Roboterseite is also explained in detail there.

Programming with the Appinventor is event-driven and object-oriented. That means we react with our program to individual events such as clicks on buttons, the expiry of a timer, the receipt of a UDP message and so on. All of this happens in a modular system.

Let's start the Appinventor in Firefox or Chrome. If this is your first time using the software, I strongly recommend working through my Einführung zu diesem Thema.

http://ai2.appinventor.mit.edu

Figure 4 shows the finished layout that we are about to create.

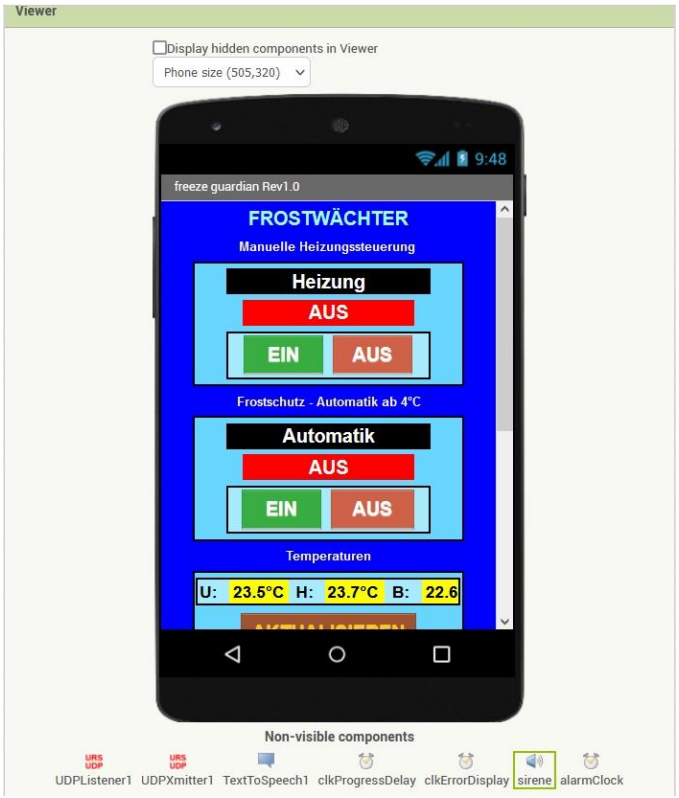


Abbildung 4: Das Viewer-Fenster

From the Palette column, to the left of the viewer, we fetch the objects that we need by dragging and dropping. "Forstwächter" is a label, as is "Manual heating control". To the right of the viewer we have the Components and Properties windows. Components shows the hierarchy of the arranged elements. The properties are set in Properties. "Frost Guard" has a font size of 20 and the color light blue, is visible and centered. The "horizontal alignment" property of "screen1" is also centered.

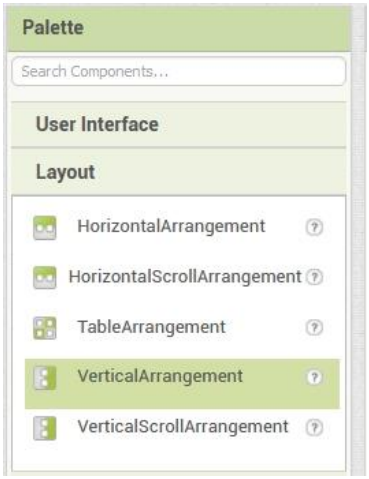
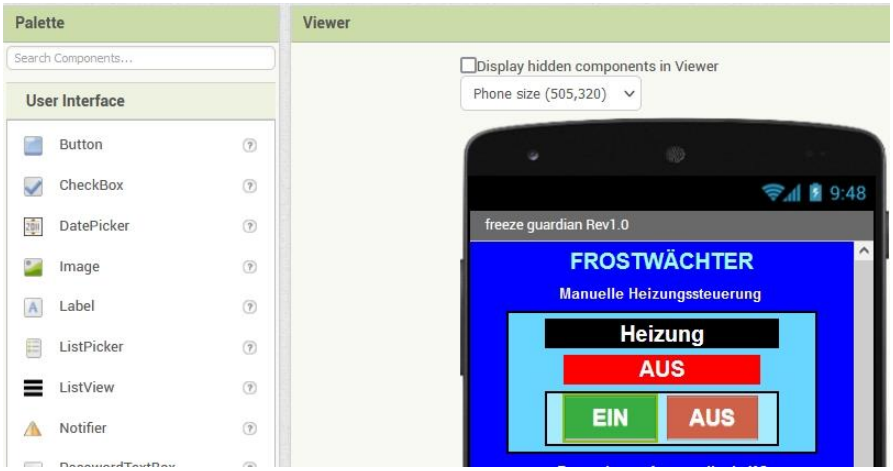


Abbildung 5: layout-Ordner

The structure of the heating and automatic blocks is identical. We start with a "VerticalArrangement". There we put a "HorizontalArrangement".

We get two labels from the "User Interface" folder and place them over the horizontal arrangement. There are two buttons in this.



Via the properties we set the width of the elements, the font size, font color and the background according to our wishes. For example, if the width is set to "Automatic", the width of the field adapts to the text width. "Heating" is set to 60%, "OFF" to 50 and the buttons to 24. The level is "Automatic". All alignments are "center". In addition to the fixed colors, you can use "Custom" to choose from a color area RGB colors and transparency.

Abbildung 6: Colors

The "sceren1" is switched to "Scrollable" so that you can scroll to the lower area.

For the temperatures, a "VerticalArrangement" contains a horizontal one with 6 labels and an "UPDATE" button underneath. "REQUEST STATUS" has a button and two labels below it.

Abbildung 7: Unterer Screenbereich

UDP messages are displayed in the next label, feedback from the ESP8266 in the green label field and error messages in the red. The gray button below shows

messages from the ESP8266 that reach the app without a prior request to the controller. These are messages from the humidity sensor. Whenever the situation changes and as a reminder at the end of each day, we are informed of the moisture level of the potting soil. In the case of "too dry" messages, a voice announcement informs about it and a siren wakes us from our dreams of a satisfied plant paradise. The alarm can be switched off using the button. "RESTART LISTENER" informs us about the status of the UDP receiver.

Don't be alarmed if an error message with the number 2 or 6 appears. This happens every now and then during development when changing properties and / or adding or removing elements. The mobile phone then tries to restart the app, which sometimes happens without the UDP socket having been closed beforehand. The start with the same IP and port number then fails, and the connection between the mobile phone and the Appinventor has to be cut and re-established.

In addition to the ones listed so far, a whole series of "invisible" components can be seen at the bottom of the screen in the "Viewer".

Abbildung 8: Unsichbare Elemente

From left to right these are the UDP receiver and transmitter, TextToSpeech for the voice output, two timers, a sound playback called siren and another timer for the alarm repetition.

Once all the components have been arranged, we switch from the "Designer" to the "Blocks", left-click on the far right in the action bar.

Abbildung 9: Menüs

Each event is now provided with an event handler through an arrangement of appropriate bricks. There are several short handlers, a few mediocre ones, and one absolute monster handler. The latter corresponds to the parser routines in our Python programs. In general, we encounter the same structures here several times as in the MicroPython programs, the indentations by the brackets, the reaction to events, etc.

We start with the definition and declaration of the required global variables. We also create the procedure for obtaining the temperature values. The definition blocks of the variables are drawn from the Variables folder, the assignments from text, including join, come from there.

We take the procedure bracket from the folder Procedures. UDPXmitter1 appears in blocks at the very bottom, from there we drag a call block into the viewer. It should send the text constant getTemp to the ESP8266. Because of course the answer will not be available immediately, we set the alarm clock clkProgressDelay to 1000ms and arm it. We'll come back to that later.

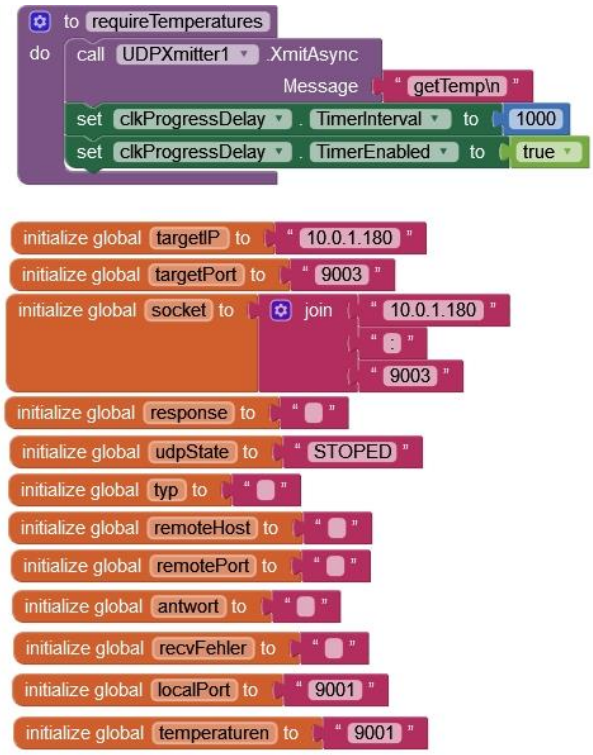


Abbildung 10: Blocks_Variablen und Funktionen

Every program needs an initialization phase, here is ours. With the screen layout, things also happen in the when screen1 initialize bracket.

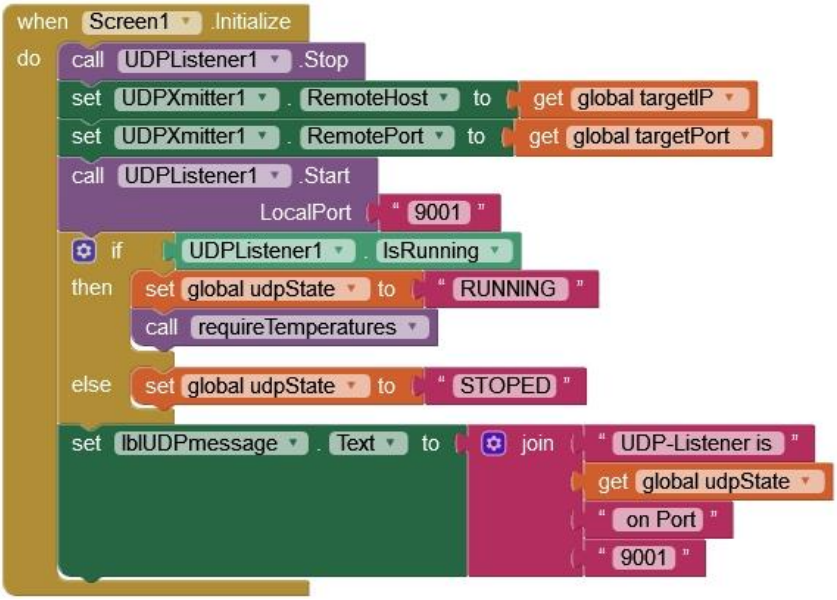


Abbildung 11:Blocks_Initialisierung

The UDP receiver is stopped, the sender is aligned with the ESP8266 as the target and the receiver is set to port number 9001. Then we query the status of the recipient and use join to create a message for the label lblUDPmessage. Admittedly, that looks monstrous, and in MicroPython it is shorter. But this creates a product that runs under Android without us having to have any idea of programming under this operating system.

It continues with the reactions to buttons and timers. Six of the brackets have the same or at least a similar structure, the treatment of the buttons. As with the others, when btnAutomatikOn.Click clears the content of the lblError error display and the corresponding command string is sent to the ESP8266. The task to update the temperatures is more complex and is initiated by the procedure requireTemperatures.

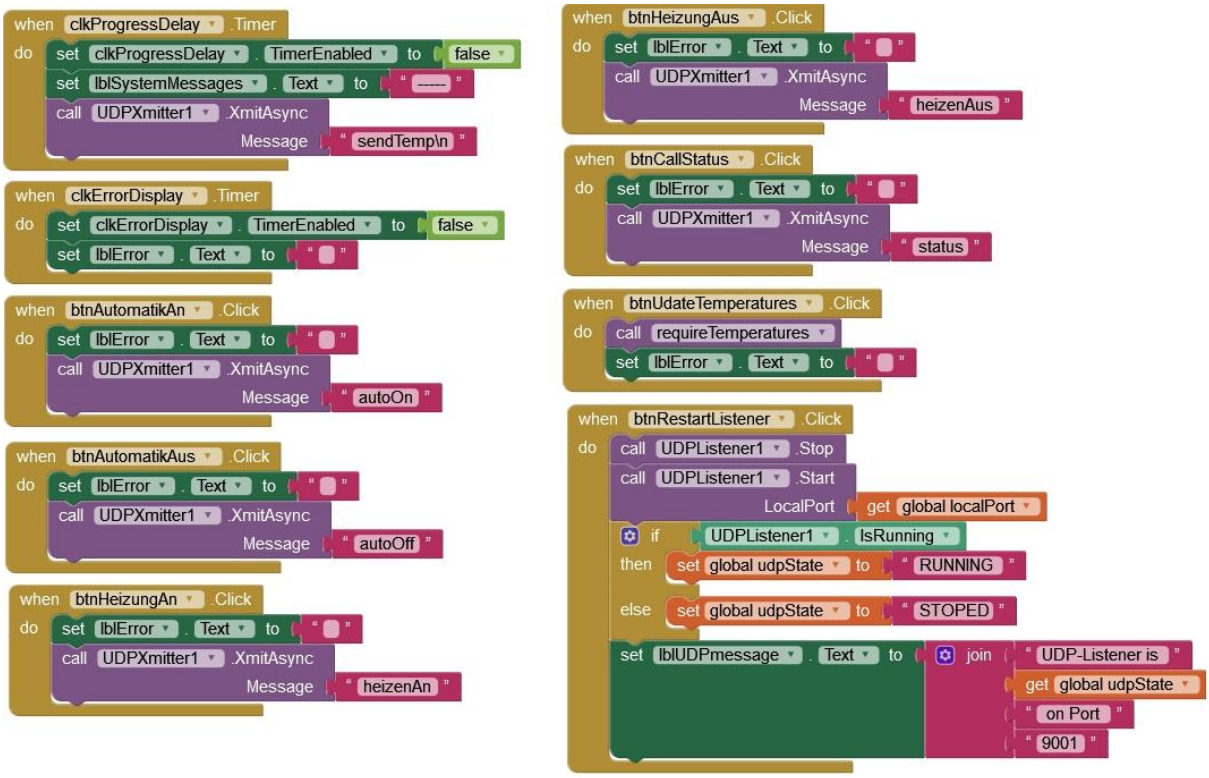


Abbildung 12: Blocks_Ereignisse

The situation is similar with the timers. In any case, the timer must be deactivated after it has been addressed. If clkProgressDelay has expired, the handler deletes the system message in lblSystemMessages and issues the command to send the temperature values to the ESP8266. Error messages are deleted when the clkErrorDelay timer expires.

The RESTART LISTENER button triggers almost the same actions as during initialization.

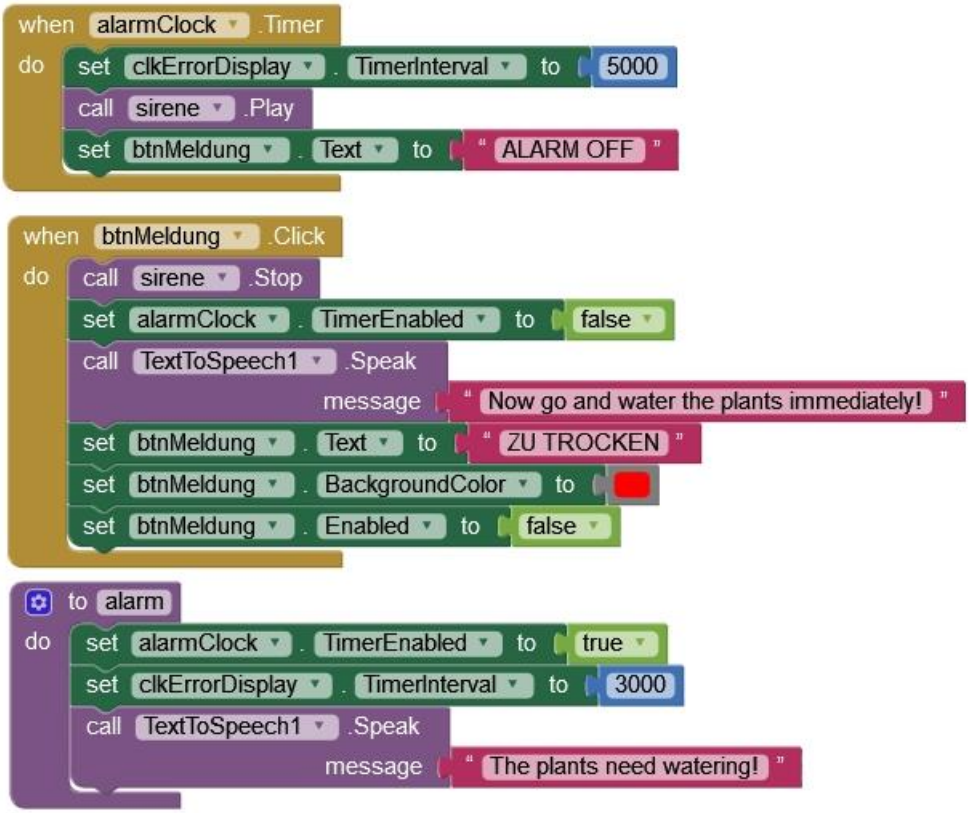


Abbildung 13: Blocks_Alarmbehandlung

These three blocks deal with the alarm case. The alarm procedure is responsible for triggering the alarm. The alarm timer is initially set to 3 seconds and armed. The voice message "The plants need watering" is output via the cell phone loudspeaker. So that language really comes out, we set the country and language properties to the correct values in the designer. If they don't fit, there is only silence from the loudspeaker.

When the alarm timer expires, the sound of the siren starts. To do this, a sound file in MP3 format must be uploaded to the mobile phone in the Designer. You can find tons of free sound files on the Internet (for example at Salamisound).

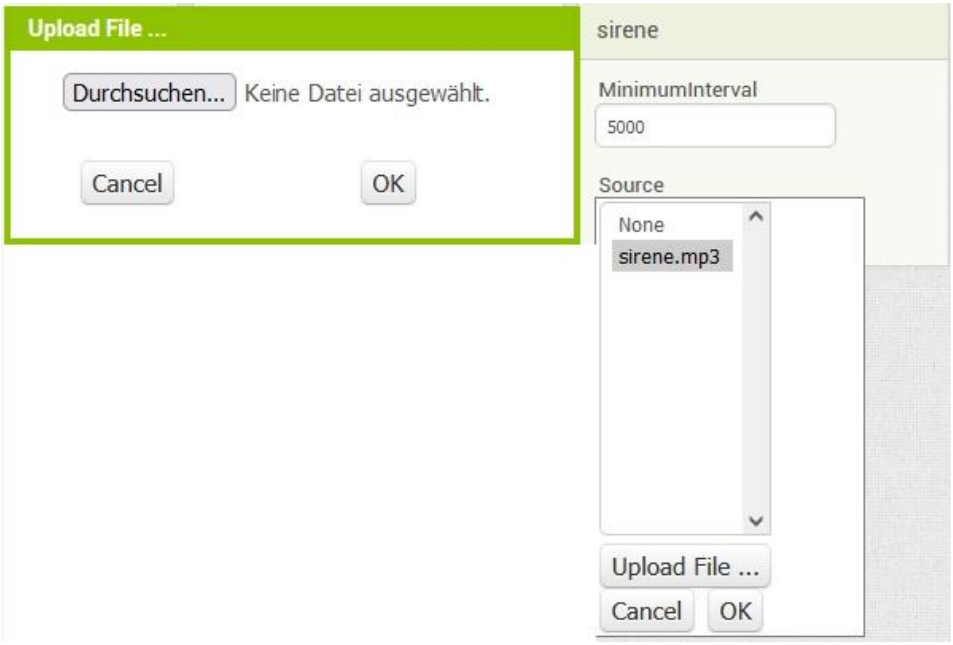


Abbildung 14: Sirene

The button is only armed if the alarm is active. With this we turn off the annoying noise flare. The sound is stopped, the timer is disabled, a reminder to water the plants is given and the button is given the warning "TOO DRY".

The Parser

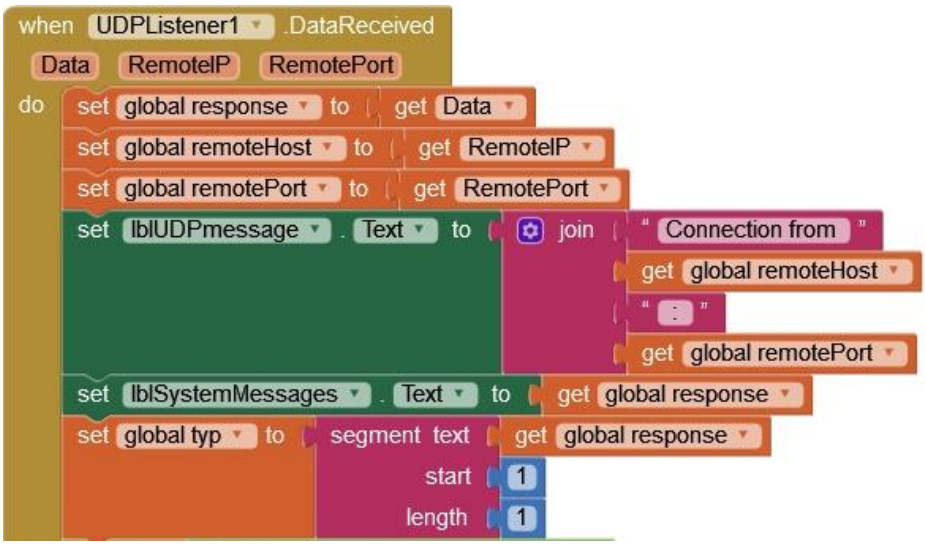


Abbildung 15: Blocks_Start Parsing

The parsing is started when a message arrives in the buffer of the UDP receiver. when UDPListener.DataReceived then responds, fetches the payload after response and asks for the send address. This is shown in the label lblUDPmessage. The received text is output in the label for system messages. Finally, we determine the type of message by isolating the first letter.

Several if-elseif blocks follow and filter out the individual responses of the ESP8266, as we have already done in the MicroPython programs. Type T is the most interesting because it shows the List object in the application.

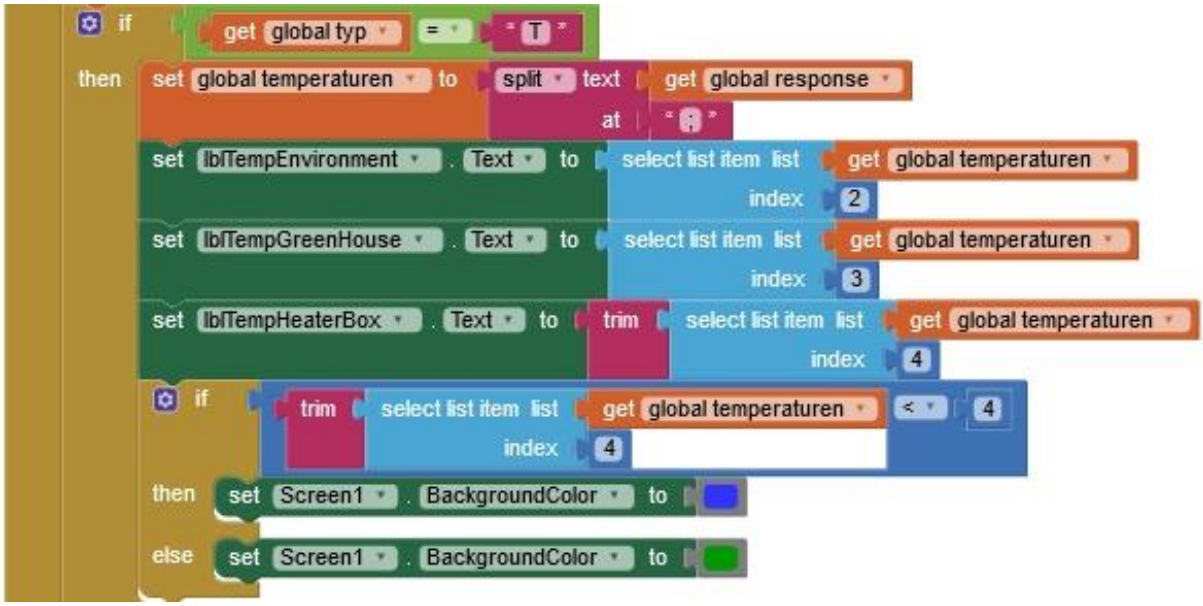


Abbildung 16 :Block_Temperaturen

The split text block divides the content of the response from the ESP8266 into individual strings at the semicolons and assigns them to the global variable temperatures as a list. This corresponds to the MicroPython command

temperatures = response.split (";")

We now assign the list elements to the output fields individually. For this purpose we get select list item from the Lists folder. With list we plug in our list of temperatures and with index the index that starts with 2, because 1 is the type. At the box temperature, the "\ n" must be cut off at the end. If the temperature in the box is less than 4 ° C, the background color of the screen is set to blue, for higher values to green.

The blocks for heating and automation are practically identical and not very demanding. They are executed if an "H" is detected as type. The only new element is contains text from the Text folder. It examines the string in text for the occurrence of the substring in piece.

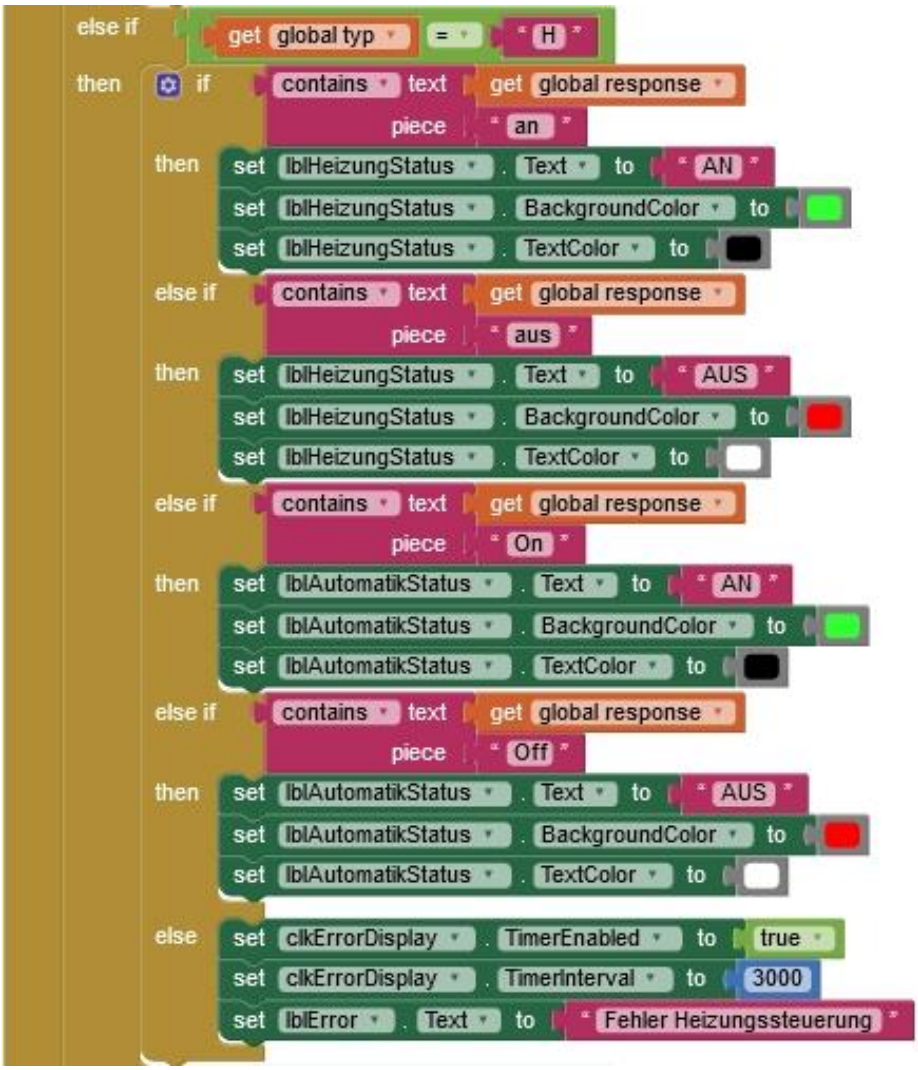


Abbildung 17: Block_Heizung

The rest of the parser also doesn't bring anything new. I am only showing the blocks for the sake of completeness.

Die Statusabfrage:

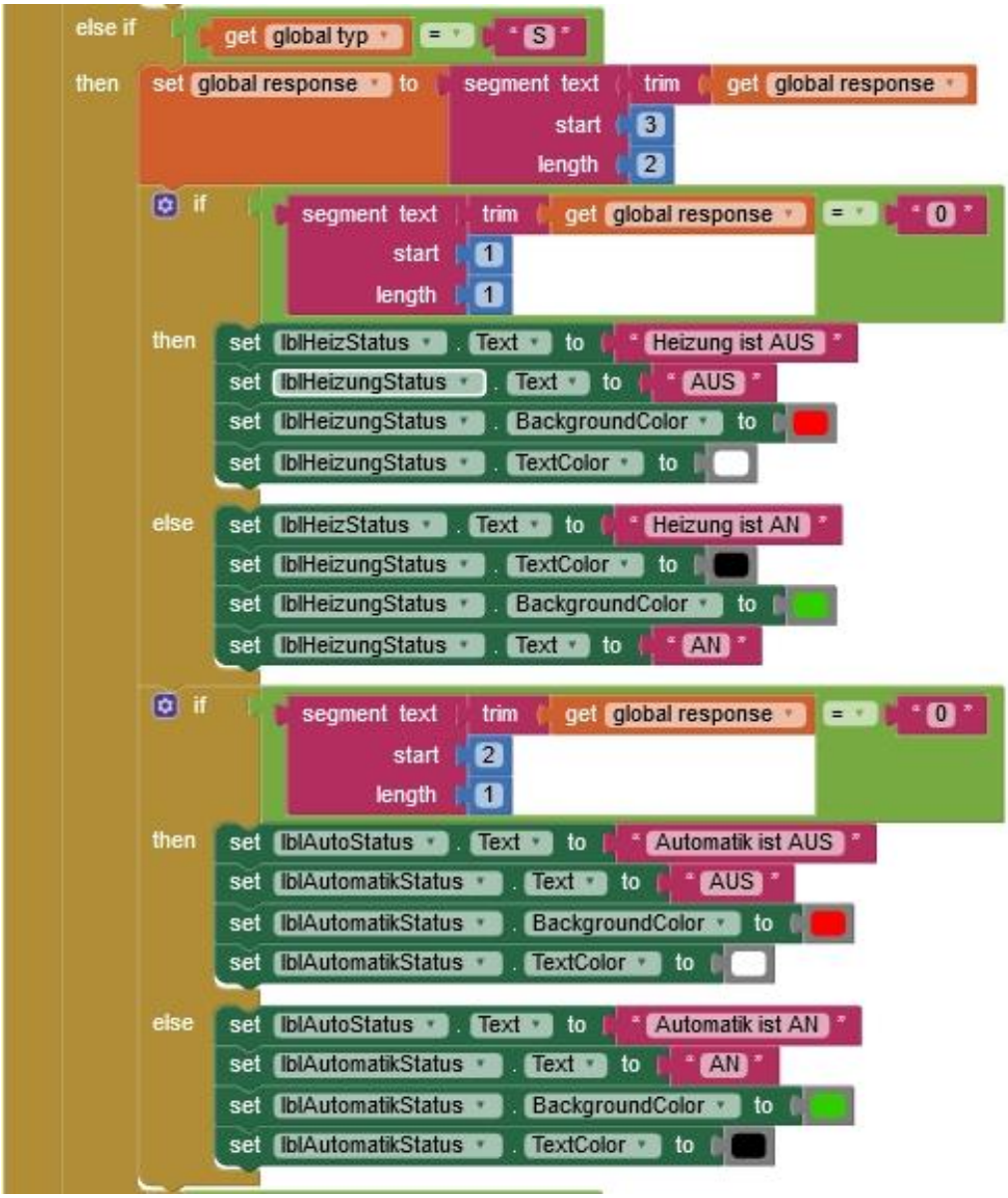
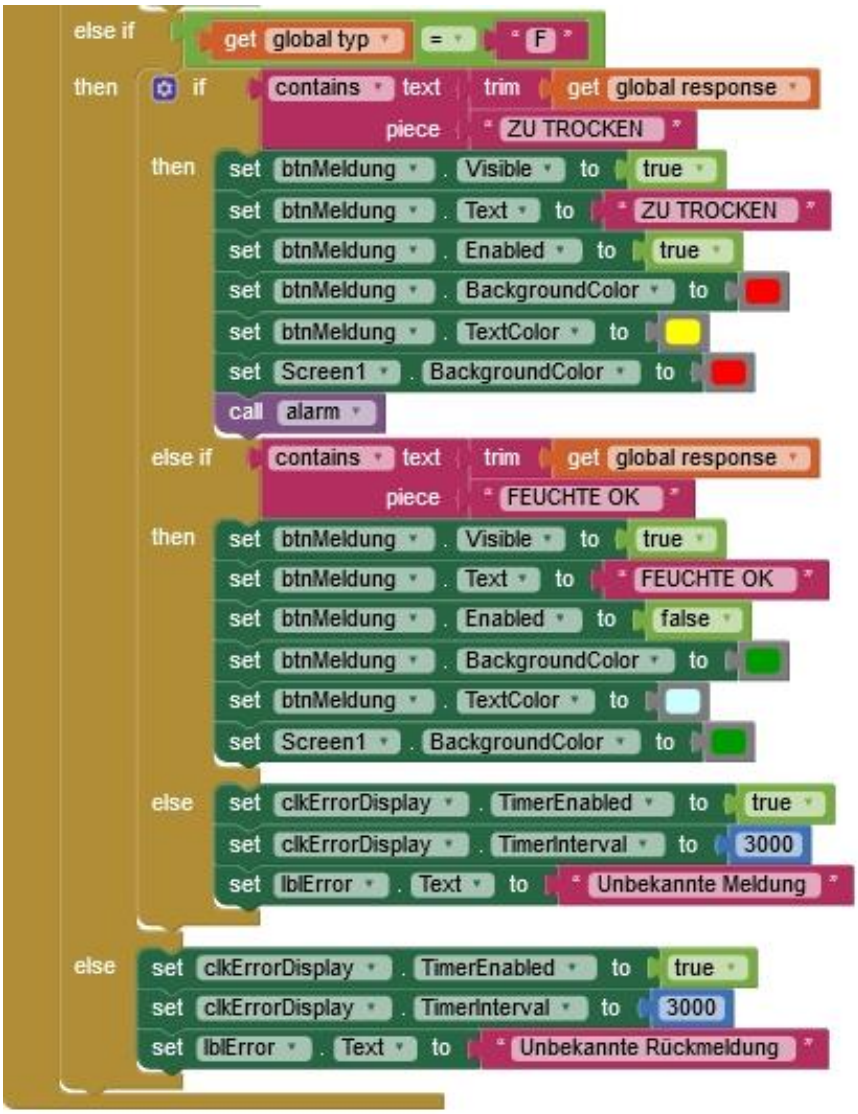


Abbildung 18: Block_Status

In the rest of the process, the alarm procedure that we have already discussed is called when changing from wet to dry.



Now it is up to you whether you want to place the objects yourself in the designer and arrange the blocks in the viewer yourself, or whether you cannot wait to see the finished solution in action. But there is one thing you have to consider, so that it works, you have to adapt the socket data, IP address and port number, to your network.

Otherwise, I hope you enjoy programming additional servers based on UDP or TCP or in connection with Appinventor 2. You have received the tools for working through this project.

You can find the previous articles on the topic of Frost Guard here:

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