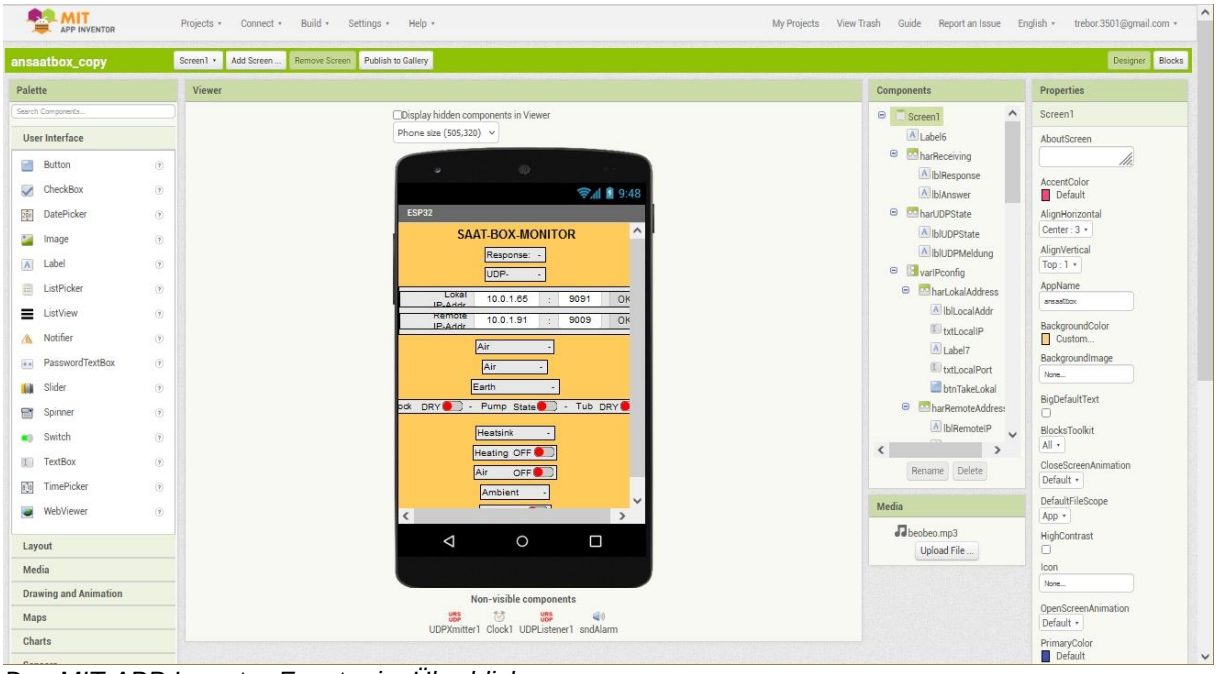


Das MIT APP Inventor-Fenster im Überblick

Diesen Beitrag gibt es auch als PDF-Datei.

Heute werden wir die Teile aus den vier vorangegangenen Folgen

- AHT10 – Sensor für Temperatur und rel. Luftfeuchte
- Es werde Licht
- Wasser marsch
- Peltierheizung

zusammenfügen und mit einer WLAN-Verbindung über UDP ausstatten. Damit wird die Saatbox in die Lage versetzt, uns den Zustand der diversen Hardwaregruppen aufs Handy oder Tablet zu senden. Mit dem MIT-App-Inventor 2 habe ich eine App entwickelt, damit die Daten am Handy überhaupt empfangen und auch in vernünftiger Form dargestellt werden können. Es gibt eine Menge zu tun, packen wir's an in der neuen Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Teil 5 – WLAN für die Ansaatbox

Die Hardware und die Programme sind in den oben genannten, früheren Beiträgen aufgeführt und ausführlich dargestellt. Heute konzentriere ich mich auf die Themen WLAN-Zugang und Handy-App. Deshalb gibt es in diesem Beitrag auch keine Hardwareliste. Sie können die einzelnen Baugruppen nach Ihren Vorstellungen auswählen, zusammenbauen und dann die nötigen Programmteile zu einem kompletten Programm kombinieren. Die Hardwarelisten dazu finden Sie in den oben genannten Beiträgen.

Die Software

Fürs Flashen und die Programmierung des ESP32:

Thonny oder
µPyCraft

Verwendete Firmware für den ESP32:

v1.19.1 (2022-06-18) .bin

Die MicroPython-Programme zum Projekt:

ssd1306.py Hardwaretreiber für das OLED-Display
oled.py API für das OLED-Display
temperatur.py Das Programm zur Lichtsteuerung
aht10.py API für den Klimasensor AHT10
licht.py Beleuchtungssteuerung
wasser.py Bewässerungssteuerung
greenhouse.py das Komplettpaket

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung (english version). Darin gibt es auch eine Beschreibung, wie die Micropython-Firmware (Stand 18.06.2022) auf den ESP-Chip gebrannt wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, damit der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm kompilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Makro fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen.

Autostart

Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP-Chip hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton, oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein kompiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA oder … enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer so, wie hier beschrieben.

Die Ansaatbox wird WLAN-fähig

Für einen WLAN-Zugang brauchen wir einen WLAN-Router, etwa eine Fritzbox oder einen anderen Funkrouter. Damit der ESP32 dort landen kann, muss seine MAC-Adresse dem Router bekannt sein. Wir müssen also herausfinden, wie die MAC-Adresse des ESP32 lautet. Damit der ESP32 es uns verraten kann, müssen wir ein Funk-Interface erzeugen, und dazu brauchen wir das Modul **network** und später wird noch das Modul **socket** benötigt. Wir importieren beide in einer Zeile. Danach folgen alle anderen Klassen und Objekte. Das einzige externe Modul ist **aht10**. Die Datei aht10.py muss daher auf den ESP32 hochgeladen werden.

```
import network, socket
from machine import SoftI2C, Pin, ADC, PWM
from time import sleep,ticks_ms, sleep_ms
from oled import OLED
import sys
from neopixel import NeoPixel
from onewire import OneWire
from ds18x20 import DS18X20
from aht10 import AHT10
import gc
```

Für den Verbindungsaufbau zum Router benötigen Sie Ihre eigenen Credentials. Tragen diese bei mySid uns myPass ein.

```
# ************** Objekte declarieren ******************
#
# debug=True
debug=False

# ***********************************************************
# Geben Sie hier Ihre eigenen Zugangsdaten an
mySid = 'here goes your SSID'
myPass = "here goes your password"
monitor=("10.0.1.10",9091)
client=("10.0.1.65",9091)
tablet=("10.0.1.40",9001)
server=("10.0.1.100",9119)
adressen=[client] # ,client,tablet,server
myPort=9009
# ***********************************************************
```

Sollen die Daten vom ESP32 an mehrere Empfänger geschickt werden, dann tragen Sie bitte die Namen der Verbindungen, getrennt mit einem Komma, in die Liste **adressen** ein.

Der folgende Block deklariert die benutzten Objekte. Die Beschreibungen dazu finden Sie in den jeweiligen Beiträgen.

```
i2c=SoftI2C(scl=Pin(22),sda=Pin(21))
d=OLED(i2c,heightw=32)

# Wasserversorgung
wanne=Pin(23, Pin.IN)
glas =Pin(19, Pin.IN)
pump=Pin(12,Pin.OUT, value=0)
aht=AHT10(i2c)
pwmPin=Pin(27,Pin.OUT,value=0)
pwm=PWM(pwmPin,5)
pwm.freq(5)
pwm.duty(0)

# Licht
ldr=ADC(Pin(34))
ldr.atten(ADC.ATTN_0DB)
ldr.width(ADC.WIDTH_10BIT)
maxCnt=1023

neo=16
neoCnt=128
neoPin=Pin(neo,Pin.OUT)
np = NeoPixel(neoPin, neoCnt)
aus=(0,0,0)
an =(127,0,100)
state=("aus","an")
neoRel=Pin(15,Pin.OUT,value=0)
```

```
# Temperatur
heater=Pin(13,Pin.OUT, value=0)
vent=Pin(17,Pin.OUT, value=0)
ds18=14
dsPin=Pin(ds18)
ds = DS18X20(OneWire(dsPin))
heatsink,earthtemp = ds.scan()
print("Chip-ROM-Codes:",earthtemp,heatsink)

taste=Pin(0,Pin.IN,Pin.PULL_UP)
```

Beim Verbindungsaufbau liefert der ESP32 den Status in Form von numerischen Werten. Das Dictionary **connectStatus** übersetzt sie in Klartext.

```
connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "UNKNOWN"
    }
```

Der ESP32 liefert uns die MAC-Adresse als bytes-Objekt. Das ist in vielen Fällen absolut kryptisch.

>>> import network
>>> nic = network.WLAN(network.STA_IF)
>>> nic.active(True
>>> nic.config('mac')
b'\s\x08\xd1\xd2A'

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

Die Funktion **hexMac**() macht daraus eine lesbare Fassung. Wir werden später diese Funktion auch aus unserem Programm heraus aufrufen. Sie können aber bereits jetzt die MAC-Adresse in den Sicherheitseinstellungen Ihres Routers eintragen. Wie das geht, verrät Ihnen das zugehörige Handbuch.

>>> hexMac(nic.config('mac'))
'5c-73-8-d1-d2-41'

Eine ganze Reihe von Funktionen können Sie eins zu eins aus den bisherigen Folgen übernehmen. Dort finden Sie auch die Besprechung der Arbeitsweise.

```
def licht(val=an):
    r,g,b=val
    if r != 0:
        neoRel.value(1)
        sleep(0.3)
        for i in range(neoCnt):
            np[i]=(r,g,b)
        np.write()
    else:
        neoRel.value(0)

def heizung(val=None):
    assert val in (0,1,None)
    if val is not None:
        heater.value(val)
    else:
        return state[heater.value()]

def pumpe(val=None):
    assert val in (0,1,None)
    if val is not None:
        pump.value(val)
    else:
        return state[pump.value()]

def luefter(val=None):
    assert val in (0,1,None)
    if val is not None:
        vent.value(val)
    else:
        return state[vent.value()]

def temperatur(c):
    ds.convert_temp()
    sleep_ms(750)
    return ds.read_temp(c)

def getADC(adc,n=50):
    sum=0
    for _ in range(n):
        sum = sum + adc.read()
    avg = int(sum/n)
    return avg

def TimeOut(t):
    start=ticks_ms()
    def compare():
        return int(ticks_ms()-start) >= t
    return compare
```

```
def sound(dauer,freq=8):
    period=TimeOut(dauer*1000)
    pwm.freq(freq)
    pwm.duty(512)
    while not period():
        pass
    pwm.duty(0)
```

Änderungen habe ich an den Funktionen **wasserstand**(), **klima**() und **beleuchtung**() vorgenommen. Hier wurden an diversen Stellen Aufrufe der Funktion **senden**() eingefügt. Übermittelt wird ein String, der den Zustand der Baugruppen an das Handy oder weitere Empfänger sendet. Jeder String setzt sich aus einem Funktionscode und einem Wert zusammen. Sie sind durch einen ":" getrennt. Folgende Codes werden verwendet.

Code	**Typ**	**Bemerkung**
S	int	Umgebungslicht vom ADC
L	0|1	Beleuchtungs-LEDs aus | an
F	0|1	Lüfterrmotor aus | an
C	0|1	Relais am Peltierelement aus | an
H	float	Wert der rel. Luftfeuchtigkeit
T	float	Lufttemperatur
G	0|1	Vorratsgefäß leer | voll
W	0|1	Wanne leer | voll
P	0|1	Pumpe aus | an
K	float	Kühlkörpertemperatur
D	0|1	Warnung, aus | an
B	float	Boden- / Wassertemperatur

```
def wasserstand():
    pegelWanne=wanne.value() # 1 = leer; 0 = voll
    pegelGlas=glas.value()
    state=0 if pegelGlas == 1 else 1
    senden("G:{}".format(state))
    if debug: print("Wanne",pegelWanne)
    if pegelWanne == 0:
        if debug: print("Wanne Pegel OK")
        pumpe(0)
        senden("W:1")
        senden("P:0")
        senden("D:0")
        return 1 # Wanne OK
    else:
        # Wanne zu wenig
        senden("W:0")
        if debug: print("Glas",pegelGlas)
        if pegelGlas==0:
            if debug: print("Glas Pegel OK, pumpen")
            pumpe(1) # Wasser vom Glas zur Wanne
```

```
            senden("P:1")
            senden("D:0")
            senden("G:1")
            return 2
        else:
            if debug: print("Glas nachfüllen")
            pumpe(0) # Pumpe aus
            senden("P:0")
            senden("D:1")
            senden("G:0")
            sound(2,5) # kein Wasser im Pool
            return 0
```

Die **if debug**-Zeilen geben im Terminal einen Hinweis auf den Zustand des Systems.

Sendebefehle wurden auch in die Funktion **klima**() eingebaut. Die Grenzwerte für die Temperaturen können Sie gerne nach Ihren Wünschen ändern.

```
def klima():
    tempKK=temperatur(heatsink)
    tempSubstrat=temperatur(earthtemp)
    aht.triggerDevice()
    temp=aht.getTemp()
    feuchte=aht.getHum()
    senden("H:{:0.2f}".format(feuchte))
    senden("T:{:0.2f}".format(temp))
    senden("B:{:0.2f}".format(tempSubstrat))
    d.writeAt("Temp: {:0.2f}*C".format(temp),0,1)
    d.writeAt("Hum : {:0.2f} %".format(feuchte),0,2)
    d.writeAt("TEMP_SUB: {0:>4.1f} %".\
              format(tempSubstrat),0,2)
    senden("K:{:0.2f}".format(tempKK))
    if debug: print("Temperatur: {:0.2f}".format(temp))
    if temp >= 22:
        if debug: print("zu heiß, Luefter an")
        senden("F:1")
        senden("C:0")
        luefter(1)
        heizung(0)
    elif temp < 19: #
        if debug: print("zu kühl")
        if debug: print("KK-Temp:",tempKK)
        luefter(1)
        senden("F:1")
        if tempKK <= 25:
            if debug: print("KK Temp OK, heizen")
            senden("C:1")
            heizung(1)
        else:
            if debug: print("KK zu heiss, nur umwaelzen")
            heizung(0)
            senden("C:0")
```

```
    else:
        if debug: print("Temperatur OK")
        heizung(0)
        senden("C:0")
        if tempKK >= 25:
            if debug: print("heizen Nachlauf")
            luefter(1)
            senden("F:1")
        else:
            if debug: print("No Action needed")
            luefter(0)
            senden("F:0")
```

```
def beleuchtung():
    h=getADC(ldr)
    if debug: print("LDR: ",h)
    if h > 200:
        licht(an)
        resp="L:1"
    else:
        licht(aus)
        resp="L:0"
    senden(resp)
    return h
```

Neu hinzugekommen sind die Funktionen **erde**() und **senden**().

```
def erde():
    tempErde=temperatur(earthtemp)
    senden("B:{:0.2f}".format(tempErde))
```

```
def senden(sub):
    code=sub.encode()
    if debug: print("Senden",sub)
    for adr in adressen:
        if debug: print("Senden",sub,adr)
        sent=s.sendto(sub,adr)
        sleep(0.1)
    sleep(0.1)
```

erde() übermittelt die Boden- oder Wassertemperatur, die mit einem der DS18B20 erfasst wird.

senden() nimmt einen der oben beschriebenen Strings, wandelt ihn in ein bytes-Objekt um und sendet ihn an alle Empfänger, die in der Liste **adressen** enthalten sind.

Der folgende Block stellt die Verbindung mit dem Router her. Nach der Instanziierung des Netzinterface-Objekts und dessen Aktivierung besorgen wir uns die MAC-Adresse. Falls der ESP32 vom Handy aus angefunkt werden soll, ist es gut, wenn er

eine feste IP-Adresse bekommt, hier die 10.0.1.180. Die IP-Adressen und Ports gleichen Sie bitte auf Ihr Heimnetz an. Nun wird versucht, eine Verbindung aufzubauen. Während das läuft, wird im Sekundentakt ein Punkt ausgegeben. Der Vorgang dauert um die 2 bis 5 Sekunden. Schließlich lassen wir uns die Verbindungsdaten auf dem Display ausgeben – drei Sekunden zum Lesen.

```
nic=network.WLAN(network.AP_IF)
nic.active(False)

nic = network.WLAN(network.STA_IF)  # erzeugt WiFi-Objekt nic
nic.active(True)                    # nic einschalten
MAC = nic.config('mac')   # binaere MAC-Adresse abrufen und
myMac=hexMac(MAC)         # in eine Hexziffernfolge umwandeln
print("STATION MAC: \t"+myMac+"\n") # ausgeben
sleep(1)
# sys.exit()
STAconf = nic.ifconfig(("10.0.1.180","255.255.255.0",\
                        "10.0.1.20", "10.0.1.100"))
if not nic.isconnected():
  nic.connect(mySid, myPass)
  print("Status: ", nic.isconnected())
  d.writeAt("WLAN connecting",0,1)
  points="..........."
  n=1
  while nic.status() != network.STAT_GOT_IP:
    print(".",end='')
    d.writeAt(points[0:n],0,2)
    n+=1
    sleep(1)
print("\nStatus: ",connectStatus[nic.status()])
d.clearAll()
STAconf = nic.ifconfig()
print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",STAconf[1],\
                    "\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
d.writeAt(STAconf[0],0,0)
d.writeAt(STAconf[1],0,1)
d.writeAt(STAconf[2],0,2)
sleep(3)
```

Es fehlt uns noch die Tür für die Daten. Die öffnet der UDP-Socket **s**, den wir als erstes erzeugen müssen. Dann stellen wir dessen Eigenschaften ein und binden ihn an den eingangs angegebenen Port. Der Timeout von 0,1 Sekunden ist wichtig, damit die Empfangsschleife nicht die Hauptschleife blockiert.

```
# ******************Socket for UDP-Server*****************
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(('', myPort))
print("x-mitting on port",myPort)
s.settimeout(0.1) # timeout, damit 'while True:' durchlaeuft
d.writeAt("ON PORT {}            ".format(myPort),0,2)
sleep(3)
```

```
d.clearAll()

d.writeAt("GREEN HOUSE",2,0)
```

Gleich zu Beginn räumen wir den Speicher auf und schauen dann nach, ob eine Nachricht eingetroffen ist. Wenn der Empfangspuffer, den wir mit **recvfrom**() abfragen, leer ist, wird nach 0,1 Sekunden eine Exception geworfen, die wir mit **except** abfangen. Im **try**-Block müsste nun das Parsen der Nachricht folgen.

```
while 1:
    gc.collect()
    try:
        rec,adr=s.recvfrom(10)
        rec=rec.decode()
        print("Empfangen:",rec,adr)
    except :
        pass
    beleuchtung()
    wasserstand()
    klima()
    erde()
    if debug: sleep(3)
    if debug: print(" ")
    if taste.value() == 0:
        d.clearAll()
        d.writeAt("PROGRAM CANCELED",0,0)
        heizung(0)
        luefter(0)
        pump(0)
        licht(aus)
        sys.exit()
```

Es folgen die Aufrufe der vier Funktionen, welche die Messwerte und Zustände abrufen und versenden. Den Schluss macht die Abfrage der Flash-Taste, die einen geordneten Ausstieg aus dem Programm ermöglicht, wobei alle Aktoren ausgeschaltet werden. Ein Schleifendurchlauf dauert zwischen 5 und 7 Sekunden.

Das ganze Programm greenhouse.py können sie herunterladen.

Die Andriod-App

Das MIT (Massachusetts Institute of Technology) stellt mit dem App Inventor 2 (AI2) ein Werkzeug zur Verfügung, mit dem man Apps für Android Smartphones im Baukastensystem erstellen kann. Das Tool arbeitet über einen Browser. Auf dem PC muss man nichts installieren. Damit man das Ergebnis sofort live auf dem Handy sehen kann, muss dort die App AI2 Companion installiert sein, die man über den Google Playstore beziehen kann. Download und Benutzung beider Tools sind kostenlos. Sie benötigen aber für den Login ein Google-Konto, auch das ist kostenlos. Wie die Software zu bedienen ist, habe ich in dem PDF-Dokument MIT

App_Inventor_2_ger.pdf detailliert beschrieben. **Lesen Sie bitte das Tutorial unbedingt durch, wenn Sie mit AI2 noch nicht gearbeitet haben oder wenn Sie UDP damit noch nie benutzt haben. Genau dieses Protokoll brauchen wir nämlich sofort. Im Tutorial ist genau beschrieben, wo Sie die Erweiterung herbekommen und wie sie installiert werden muss.**

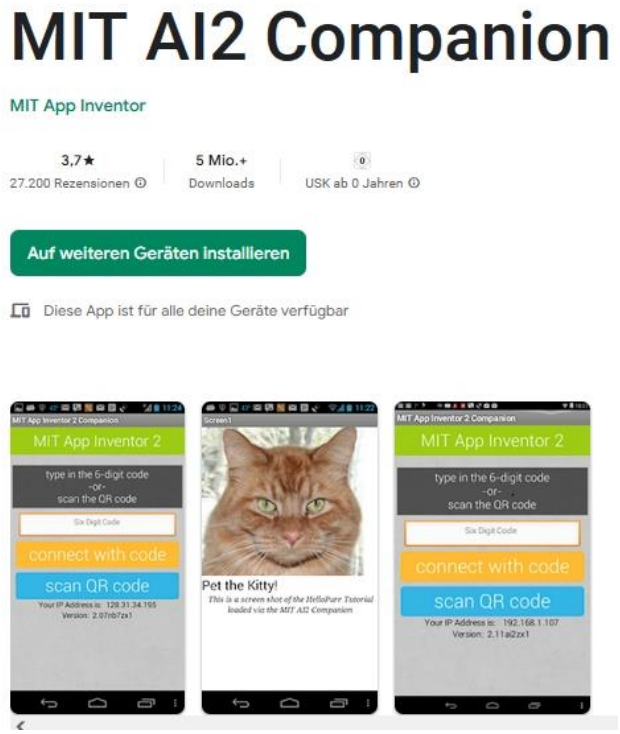


Abbildung 1: MIT AI2 Companion

Gehen Sie jetzt auf die Seite http://appinventor.mit.edu/ und klicken Sie auf **Create Apps!** Aus dem Fenster **Palette** ganz links werden die Elemente für den Screen auf die Abbildung des Handys im **Viewer** gezogen. Im **Components**-Fenster werden die Elemente über den Button **Rename** mit sprechenden Namen versehen. Die Eigenschaften des gerade ausgewählten Elements können wir im Fenster **Properties** einstellen. Abbildung 2 zeigt das Ziel unserer Bestrebungen. Für Zeilen mit zwei oder mehr Elementen brauchen wir ein **Horizontal Arrangement** aus dem Ordner **Layout**. Einzelne Zutaten werden direkt auf dem Screen platziert, wie das Label **lblTitle** mit dem Inhalt **SAAT-BOX-MONITOR**. Es kommt aus dem Ordner **User Interface**. Zur Bezeichnung gebe ich durch die ersten drei Buchstaben den Typ des Elements an, dann folgt der Name mit einem Großbuchstaben. Detailbezeichnungen folgen auch wieder mit einem Großbuchstaben. Dieses System ist nicht zwingend notwendig, aber es erleichtert später bei der Arbeit mit den Blöcken das Auffinden und Zuordnen ungemein. Scrollen Sie auch ruhig bei jedem neuen Elementtyp durch die Liste mit den Eigenschaften. Das Angebot ist für jeden Typ anders.

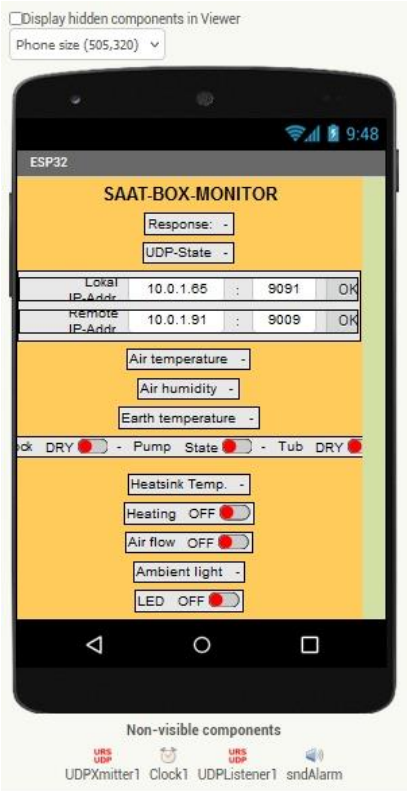


Abbildung 2: So soll die App aussehen

Die Baumstruktur in Components zeigt sehr schön, wie die einzelnen Elemente eingebracht werden. **har** steht für Horizontal Arrangement. Ich bin überzeugt, dass Sie jetzt die beiden Zeilen nach der Überschrift alleine hinkriegen. Die Labels werden einfach auf die Fläche des Horizontal Arrangements gezogen.

Abbildung 3: Erste Schritte

Der Bereich mit den IP-Adressen ist schon etwas komplexer. Hier sind zwei Horizontal Arrangements (har)in einem Vertical Arrangement (var) untergebracht. Zwei weitere Elementtypen kommen hinzu, Textfelder (txt) und Buttons (btn).

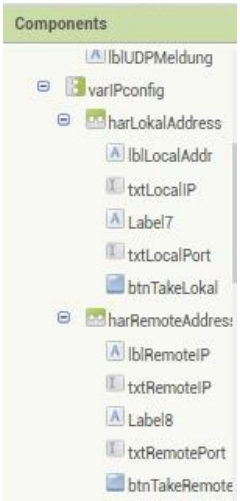


Abbildung 4: Kombi-Arrangement

Elemente, die bei der Programmierung mit den Blöcken nicht gebraucht werden, wie der Platzhalter Label7, benenne ich nicht um. Abbildung 5 zeigt die Properties des Textfelds **txtLocalIP**. Für die Eigenschaft Text habe ich die IP des Handys vorgelegt.

Abbildung 5: Eigenschaften eines Textfeldes

Sehr einfach schauen die nächsten drei Felder aus, in denen nur ein Zahlenwert über ein Label ausgegeben wird.

Abbildung 6: Felder für Wertausgabe

Auch der Bereich für die Wasserversorgung ist nicht kompliziert, nur etwas umfangreicher. Weil nur Zustände angezeigt werden und keine Werte, habe ich den Typ Switch (swi) dafür gewählt. Abbildung 8 zeigt die Eigenschaften eines Schalters. Beachten Sie, dass das Feld Enabled nicht aktiviert ist. Dadurch kann das Element als Anzeige dienen, aber nicht bedient werden.

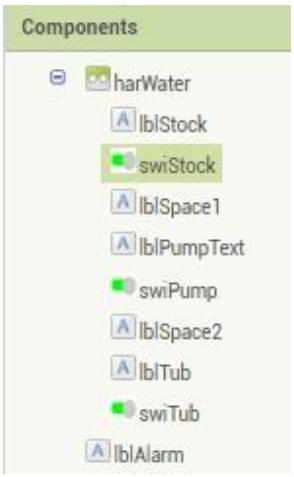


Abbildung 7: Zustände bei der Wasserversorgung

Abbildung 8: Eigenschaften eines Schalters

Die letzten fünf sichtbaren Bereiche bringen nichts neues mehr.

Abbildung 9: Die letzten sichtbaren Arrangements

Interessant wird es wieder zum Schluss. **UDPXmitter** und **UDPListener** wohnen im Ordner **Extensions**. Diesen Ordner sehen Sie nur, wenn Sie, wie im Tutorial beschrieben, Die UDP-Erweiterung von Ullis Roboterseite installiert haben. Dort finden Sie auch eine detaillierte Beschreibung der verfügbaren Blöcke.

Abbildung 10: UDP wohnt in Extensions

Die Elemente aus Extensions werden auf dem Screen abgelegt, aber nicht dort platziert, sondern sie wandern nach Süden aus. Es sind nichtsichtbare Objekte, wie die folgenden auch-

Abbildung 11: Nicht sichtbare Elemente

Das Element **clock** ist ein Timer und als Standardelement aus dem Ordner **Sensors** zu haben, ebenso wie **sound** aus dem Ordner **Media**.

Abbildung 12:Aus Sensors kommt Clock

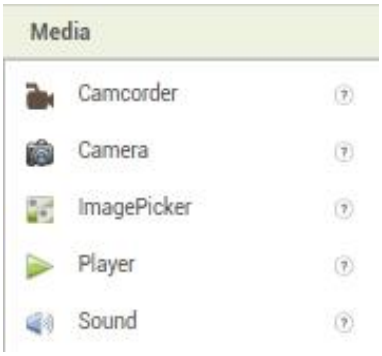


Abbildung 13: Sound aus dem Ordner Media

Zum Abspielen brauchen Sie natürlich eine mp3-Datei. Ich habe mir einen Sirenensound mit dem Tool Audacity gebastelt. Dateien werden im Fenster Media hochgeladen, das sie unterhalb Components finden.

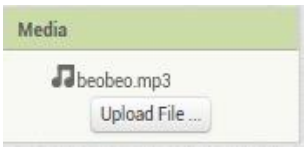


Abbildung 14: Hochladen von Dateien

Im Designer sind wir jetzt fertig. Wechseln Sie nun mit dem Button Blocks ganz rechts oben im AI2-Fenster in diesen Bereich. Unterhalb des Ordners **Built-in**, mit den Subordnern **Control**, **Logic**, **Math** und so weiter, finden Sie die von uns deklarierten Objekte in derselben Reihenfolge wie im Designer. Zu jeder Eigenschaft aus dem Fenster Properties blendet AI2 beim Markieren des Elements eine Liste von Gettern, Settern, Controls und gegebenenfalls weiteren Blocks ein. Wie in der OOP (Objekt Oriented Programing) üblich, wird nicht auf die Attribute einer Instanz direkt zugegriffen, sondern über Methoden, die auch eine Überprüfung der übergebenen Werte vornehmen können. In unserem MicroPython-Programm findet die Überprüfung zum Beispiel bei den Funktionen **pumpe**() und **luefter**() durch **assert** statt.

Die Getter rufen einen Attributwert ab, das sind die Blocks mit dem Knubbel links. Die Setter-Blöcke haben links oben eine Delle und darunter eine Beule. In die Aussparung rechts werden Getter eingehängt.

Abbildung 15: Blocks - Getter und Setter

Wir beginnen, wie bei MicroPython mit dem Deklarieren von Variablen.

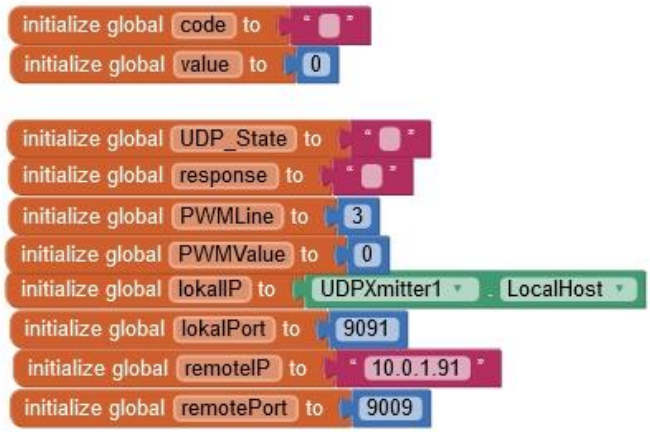


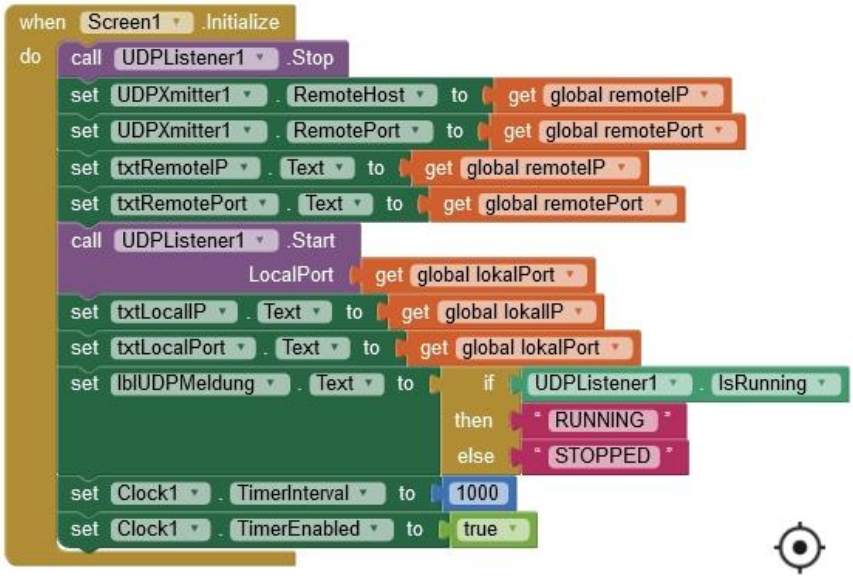
Abbildung 16: Variablen deklarieren

Ziehen Sie aus dem Subordner **Variables** in **Built-in** einen **initialize global**-Block

Abbildung 17: Blocks für Variablen

Klicken Sie auf **name** und geben Sie "**code"** ein. Holen Sie dann aus dem Subordner **Text** den leeren String und klippsen Sie ihn an den ersten Block. Wiederholen Sie den Vorgang für alle Variablen. Die numerischen Getter finden Sie in **Math**, **UDPXmitter1** ganz unten im Fenster **Blocks**.

Die Programme, die Sie mit AI2 bauen, ich sage ganz bewusst nicht schreiben, sind ereignisgesteuert. Wenn ein Ereignis eintritt, dann muss darauf reagiert werden. Wenn der Screen1 initialisiert wird, dann muss der UDP-Listener erst einmal angehalten werden, falls er läuft, dann werden die Attribute des Senders auf die IP unseres ESP32 gesetzt und am Screen dargestellt. Wir starten den Listener und übergeben der Methode Start die lokale Portnummer. Das entspricht dem Befehl **bind** aus dem MicroPython-Programm. Die Control **when Screen1.initialize** wohnt im Ordner Blocks im Objekt Screen1. Durch die Punktnotierung können Sie jedes Objekt und die Eigenschaft gezielt orten. Die Farben helfen bei den Built-in-Objekten, die ja keinen Objektnamen enthalten.



Nachdem weitere Labels upgedatet sind, setzen wir den Timer auf 1000ms und starten ihn.

Der Timer schaut immer wieder nach, ob der UDP-Listener noch läuft und meldet seinen Zustand an den Screen. Danach wird der Timer neu gestartet.

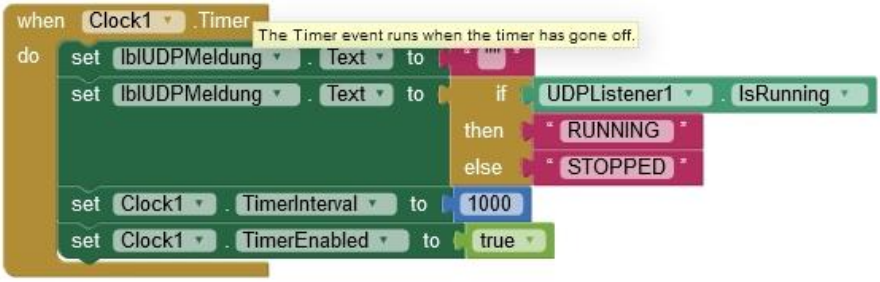


Abbildung 18: UDP-Watchdog

Wenn die Daten für die Netzverbindung geändert werden, müssen Sie auch an das Programm weitergegeben werden. Die Übernahme passiert mit Tipp auf den Button mit der Beschriftung **OK**, er hat den internen Namen **btnTakeRemote** von uns erhalten.

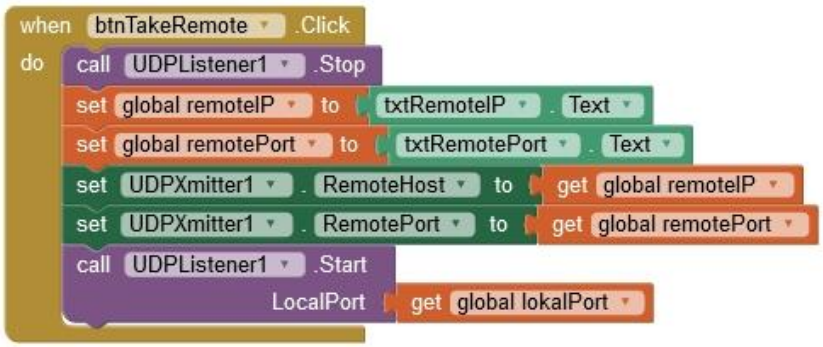


Abbildung 19: Daten für Remote übernehmen

Ähnlich sieht die Übernahme der lokalen Verbindungsdaten aus.

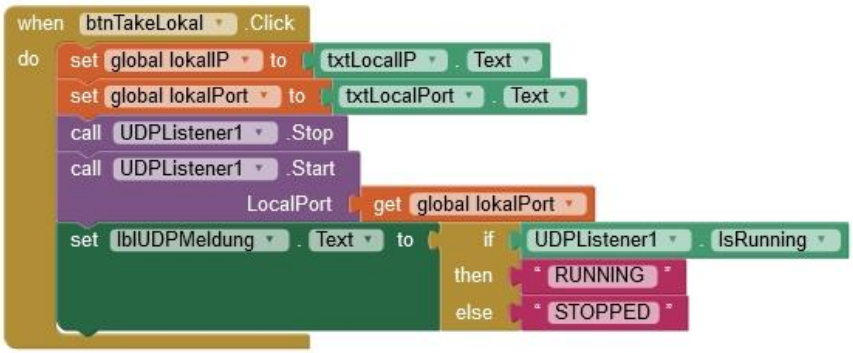


Abbildung 20: Lokale Verbindungdaten übernehmen

Falls der Listener einen Fehler meldet, muss dieser abgefangen werden. Wir kennen das aus MicroPython, dort codieren wir so etwas durch **try** und **except**. Hier bekommen wir den Fehlercode im Label **lblAnswer** angezeigt, außerdem den Zustand des Listeners.

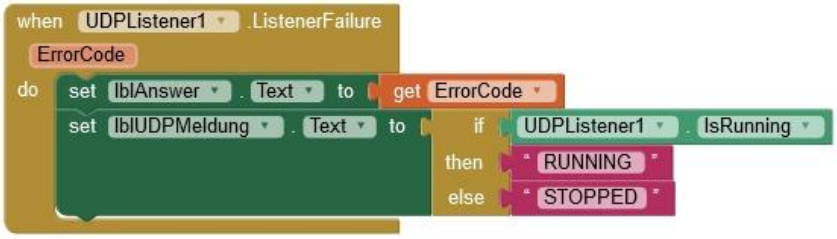


Abbildung 21: UDP-Fehlerbehandlung

Bis jetzt haben wir nur eine Verbindung aufgebaut und gepflegt und den Bildschirm initialisiert. Wo bleiben denn nun die vom ESP32 übermittelten Daten? Das Parsen der Daten kommt sofort, aber erschrecken Sie nicht! Damit man die Schrift besser entziffern kann, habe ich das gesamte Listing in drei Abbildungen verpackt.

Wenn Daten angekommen sind, muss die Meldung geparst werden. Aus dem Controll **when UDPListener.DataReceived** - **Data** ziehen wir den Getter **get Data,** weisen den Inhalt der globalen Variablen **response** zu, geben ihn am Screen aus und überprüfen noch einmal den Status der Verbindung.

Dann teilen wir die Nachricht in Code- und Wertanteil auf. Denken Sie daran, dass die Farben der Blocks ohne Objektname ihre Herkunft verraten. **segment text** stammt also genauso wie **length** aus dem Subordner **Text**.

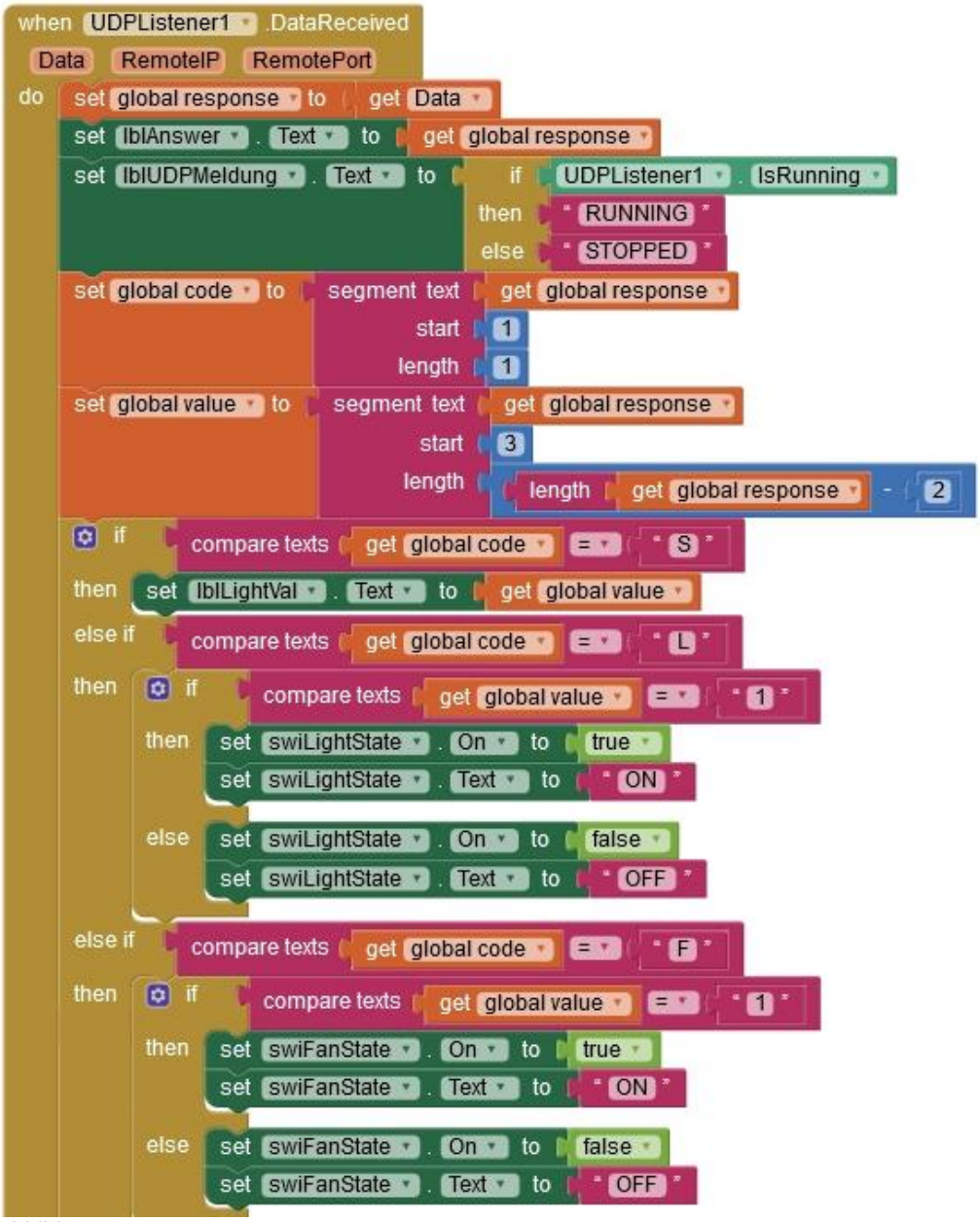


Abbildung 22: Parser_1

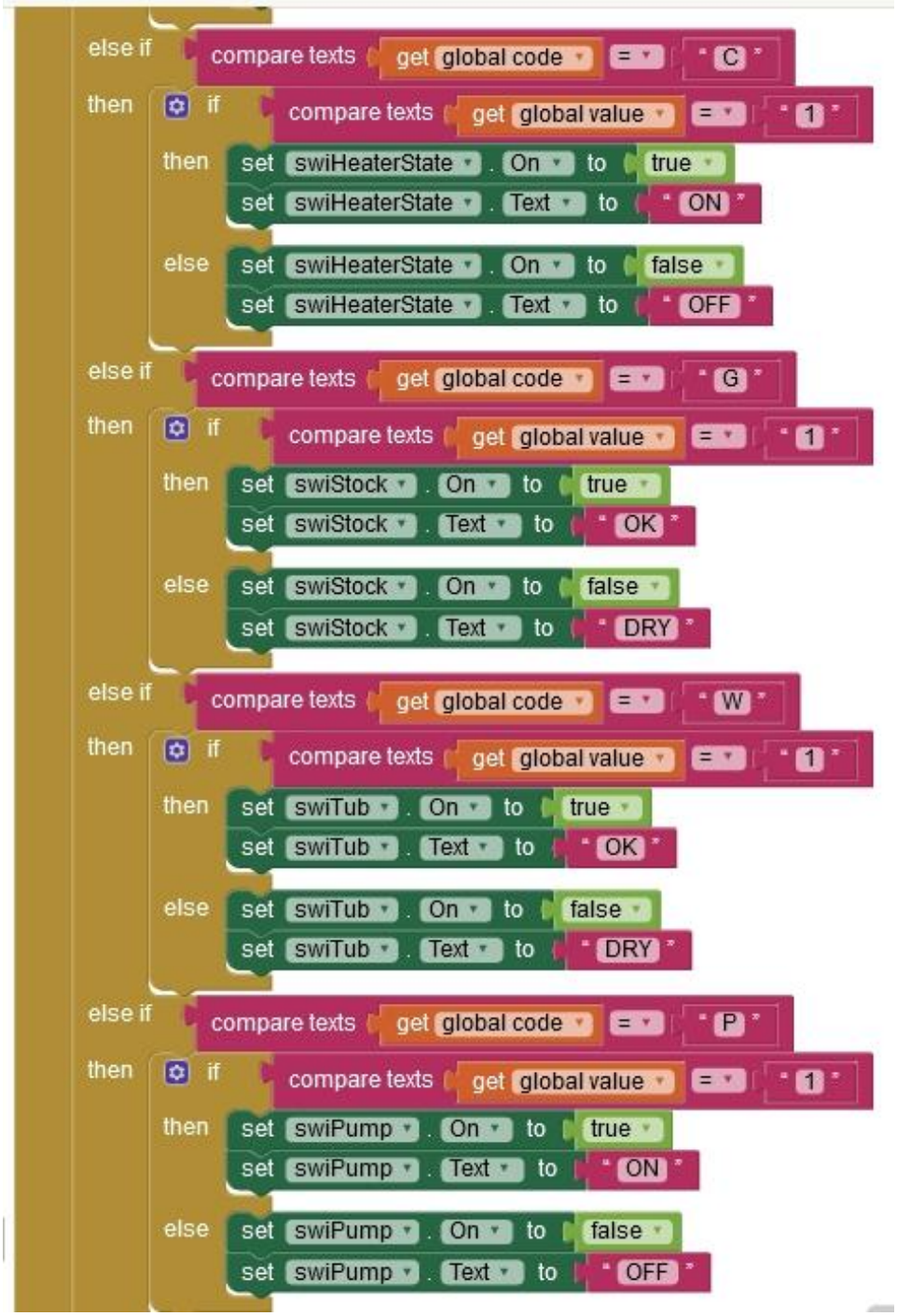


Abbildung 23: Parser_2

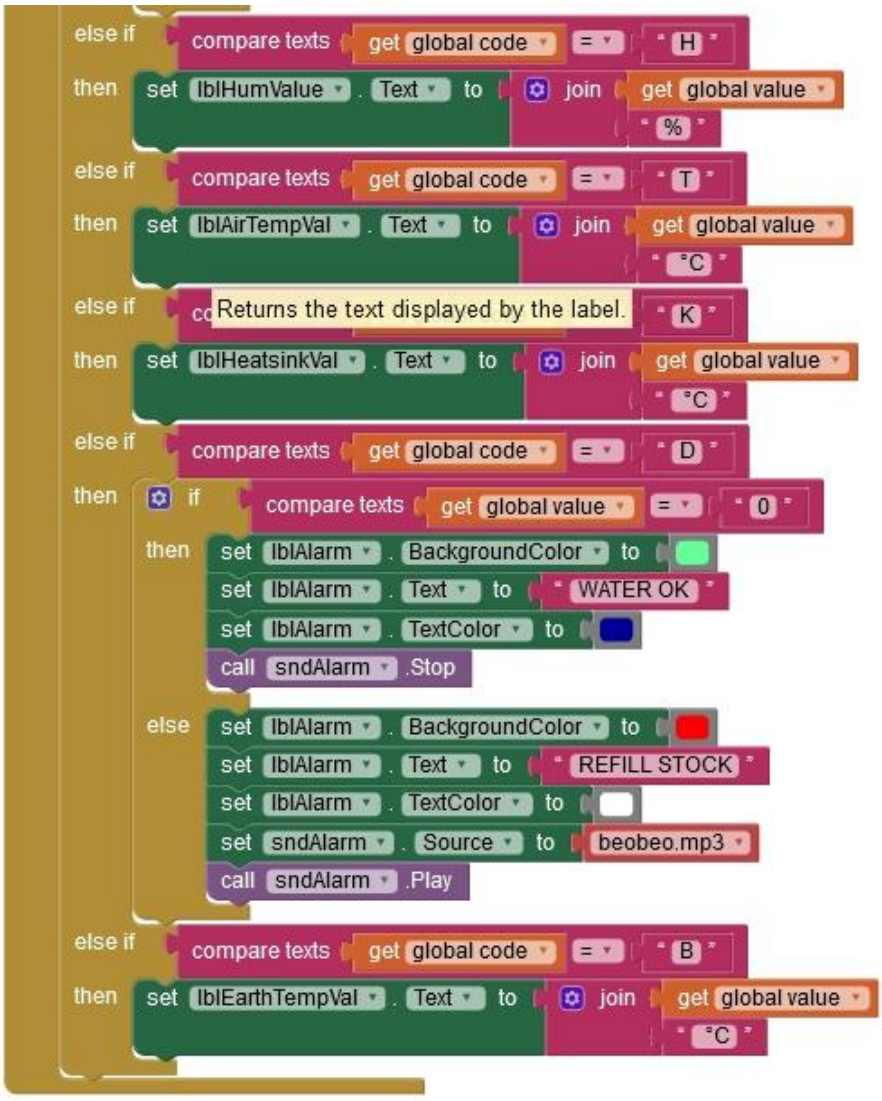


Abbildung 24: Parser_3

Weitere **else if**-Felder in der Control **if-then-else if-else** bekommen Sie, wenn Sie auf das blaue Feld mit dem weißen Rad klicken. Ziehen Sie dann ein Element von der linken Seite nach rechts und haken Sie es dort ein. **else** brauchen wir nicht, ziehen Sie es einfach ins linke Feld.

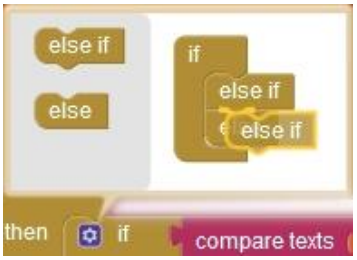


Abbildung 25: if -then-else if erweitern

Um Ihnen die Arbeit des Programmbauens zu erleichtern habe ich noch einen Tipp für Sie. Mit einem Rechtsklick auf einen Block rufen Sie das Kontextmenü auf. Mit **Duplicate** erstellen Sie eine Kopie des Blocks. Haken Sie diese an der gewünschten

Stelle ein und passen Sie die Namen und Eigenschaften der Objekte an. Das geht schneller als alles aus dem Blocks-Fenster zu holen.

Für die ganz Eiligen kann ich leider kein übliches Listing zum Download anbieten, aber ich habe etwas noch viel Besseres für Sie. Laden Sie doch einfach als erstes die Datei greenhouse.aia herunter. Im Menüpunkt Projects importieren Sie die Datei und haben das Design und die Bausteine alle im Viewer. Vergessen Sie aber vor dem Build nicht, die IP-Adressen und Ports an ihr WLAN anzupassen. Übrigens, das Ganze funktioniert nur im lokalen Netz, nicht von außerhalb. Der ESP32 kann aus dem LAN keine Verbindung über den Router des Handy-Providers zum Handy aufbauen.

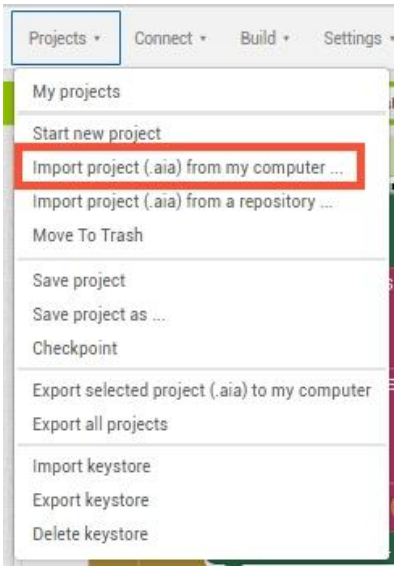


Abbildung 26: Projekt importieren

Jetzt fehlt nur noch der Download der fertigen App aufs Handy. Im Tutorial sind ab Seite 16 beide Möglichkeiten vorgestellt und beschrieben. Sie können die App via QR-Code herunterladen. Dazu brauchen Sie ein QR-Code-Leser, zum Beispiel QR-Droid aus dem Playstore.

Die andere Variante ist der Download auf den PC. Auch dieser Weg ist im Tutorial ausführlich beschrieben. Und so sieht die App auf dem Handy aus.

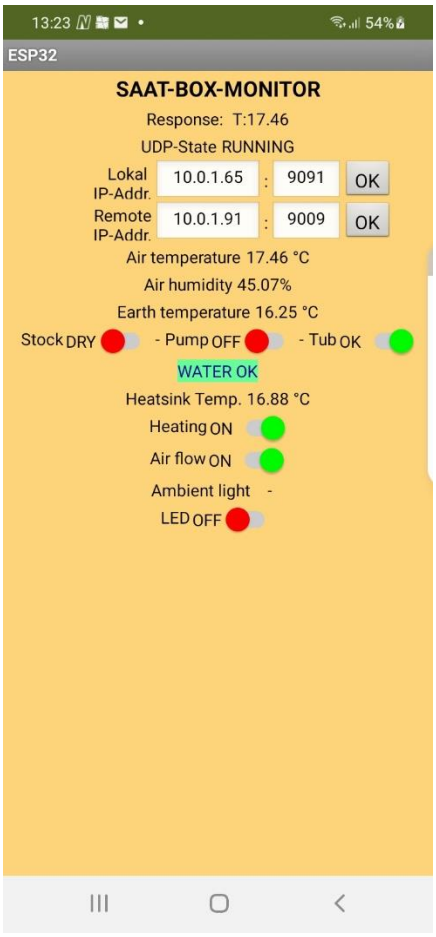


Abbildung 27: Screenshot vom Handy

Viel Spaß beim Basteln, Bauen und der Pflanzenanzucht!