# Hőmérséklet Szabályozó

Iszlai Tamás, Számítástechnika IV.év

## 1. Felhasználói platform (ESP32) bemutatása

Az ESP32 egy rendkívül népszerű, sokoldalú mikrovezérlő platform, amelyet a Espressif Systems fejlesztett ki. Az alábbiakban röviden bemutatom az ESP32 főbb jellemzőit:

### Mikrovezérlő:

Az ESP32 egy kétmagos (dual-core) mikrovezérlő, amelyet a Tensilica LX6 architektúra alkalmazásával hajtanak végre.

A magok frekvenciája változtatható, így lehetőség van az energiahatékony üzemre és a teljesítmény növelésére is.

### Wi-Fi és Bluetooth:

Az ESP32 beépített Wi-Fi és Bluetooth támogatással rendelkezik.

Kiválóan alkalmas IoT (Internet of Things) alkalmazásokra, ahol a vezeték nélküli kommunikáció elengedhetetlen.

### Perifériák és interfészek:

Számos perifériát támogat, beleértve az I2C, SPI, UART és GPIO-kat, lehetővé téve a különböző szenzorok és eszközök csatlakoztatását.

Támogatja az analóg digitális átalakítót (ADC) és a digitális analóg átalakítót (DAC).

### Memória:

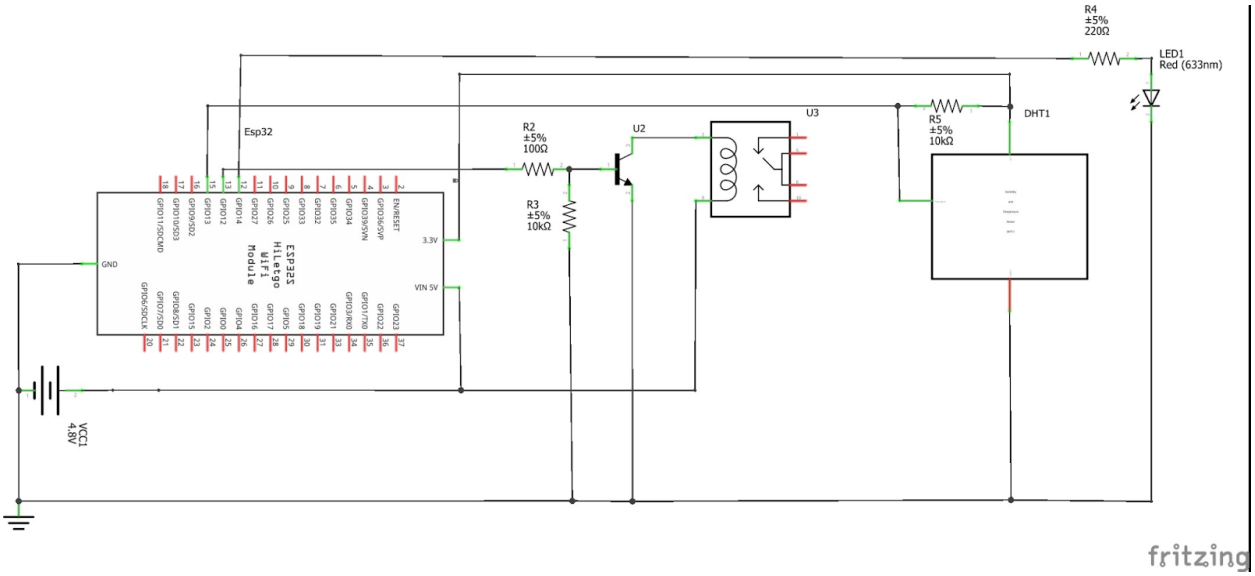
Az ESP32 rendelkezik beépített Flash memóriával a programok tárolásához, és RAM-mal a futási időben használt adatokhoz.

### Fejlesztői környezetek:

Az ESP32 fejlesztéséhez számos fejlesztői környezetet használhatunk, például az Arduino IDE-t vagy a PlatformIO-t.

# 2. Tervdokumentáció:

## Kapcsolási rajz:




## Alkatrészlista:

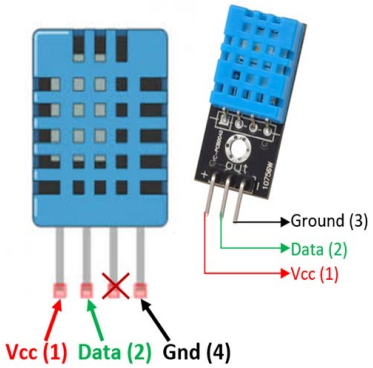
- 1 x ESP-WROOM-32
- 1 x DHT11
- 1 x SRD-05VDC-SL-C
- 1 x BC547
- 1 x LED
- 1 x VCC 5V
- 2 x 10kΩ
- 1 x 100Ω
- 1 x 220Ω

## Alkatrész leírások:

- **DHT11**

A DHT11 egy digitális hőmérséklet- és páratartalom-érzékelő modul, melyet gyakran használnak az időjárás- és környezeti monitoring alkalmazásokban. Ez a szenzor lehetővé teszi, hogy mérje a környezeti hőmérsékletet és páratartalmat, és adatait digitális jelek formájában továbbítsa egy mikrovezérlő vagy más eszköz számára. Mért hőmérséklet tartomány 0°C - 50°C, páratartalom 20% - 90%.

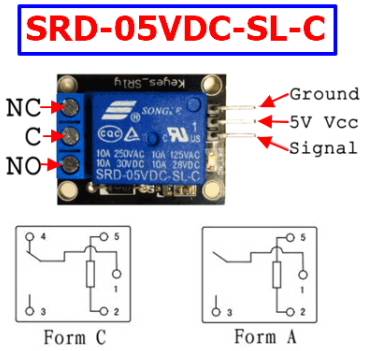
Lábkiosztás:




- **SRD-05VDC-SL-C**

Az SRD-05VDC-SL-C relét gyakran alkalmaznak elektronikus vezérlőkben és kapcsolórendszerekben. A relé olyan elektromágneses kapcsoló, amely egy kis elektromos jel segítségével képes kapcsolni vagy kikapcsolni egy nagyobb áramkört. 5V segítségével képesek vagyunk kapcsolni bármilyen eszközt aminek az áramfelvétele 10A.

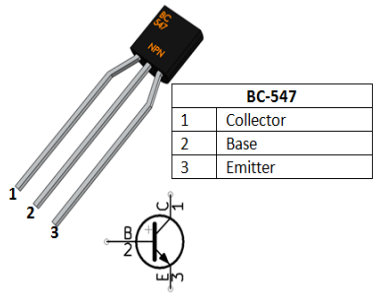
Lábkiosztás:




- **BC547**

A BC547 egy NPN típusú bipoláris tranzisztor, amely gyakran használt alapvető áramköri feladatokban. Tranzisztorokat általában erősítőként vagy kapcsolóként használják az elektronikában.

Lábkiosztás:



# 3. Megvalósítás

- **Működés leírása**

A projekt megvalósítása egy úgynevezett elektronikai próbapanelen (Breadboard) történt meg.

A DHT11 adat lába csatlakozik a GPIO 13-as lábára, emelett 3.3V-al megtáplálva és a közös földdel kommunikál DHT szenzor könyvtáron keresztül az ESP-vel.

A relé megvezérlése egy BC547-es tranzisztor segítségével valósul meg, ennek több oka is van az egyik az, hogy ESP-t megvédjük esetleges magas áram felvételtől ami tönkre tehetné, másfelől a relé megtáplálásához szükséges 5V-t egy külső feszültség forrás biztosítja( akkumulátor / 5V tápegység) amit csak így tudunk vezérelni.

Kijelzés szempontjából készült a projekt mellé egy telefonos alkalmazás, ahol láthatjuk a kijelzésre szánt adatokat és a funkcionalitásokat amire képes a projekt.
Emelett található még egy LED a lapon, amely használható sikeres kliens szerver közötti kommunikáció ellenőrzésére és jelez magas páratartalom esetén.

- **Következtetések**

A projekt jól szemlélteti, hogy minimális erőforrások segítségével létre tudunk hozni egy olyan rendszert amivel elősegíthetjük otthonunk automatizálását az által, hogy különböző eszközöket vezéreljünk meg távolról a web szerver segítségével, jelen esetben szemléltetve annak fűtésének a szabályozását egy adott pont szerint vagy annak a szobának a hőmérséklete alapján ahol az eszköz található, emelett biztosítva a bővíthetőség lehetőségét.

# 4. Beágyazott vezérlő programok rövid leírása

Az alábbi képen látható az említett LED ki-, bekapcsolásáért felelős 2 függvény, melyek működése azon alapszik, hogy egy status bitet változtatnak meg, amely kiértékelődik a fő program és annak hatására kapcsolja ki vagy be a jelző LED-t.

```
void handle_LedOn() {
  LED1Status = HIGH;
  Serial.println("GPIO14 Status: ON");
  server.send(200, "text/html", SendHTML(true,transistorStatus, data.t));
}

void handle_LedOff() {
  LED1Status = LOW;
  Serial.println("GPIO14 Status: OFF");
  server.send(200, "text/html", SendHTML(false,transistorStatus, data.t));
}
```

A fűtés kapcsolására is hasonló logikát használtam, annyi különbséggel több állapot jelző beállítása lehetséges, amelyekre mind külön függvényeket használtam, így inkább azt a részt mutattom be ahol felhasználtam az adott állapotokat.

A sourceStatus felelős azért, hogy a szenzor olvasta hőmérsékletet használjuk referenciaként vagy a telefon által biztosított értéket.

A modeStatus szerepe az automata vagy manuális beavatkozás lehetőségének a biztosítása.

A transistorStatus jelöli a beavatkozás állapotát, hogy jelenleg be vagy ki kell kapcsolni a fűtést.

```
if(sourceStatus){
  if (millis() - lastDHT11ReadTime >= DHT11_READ_INTERVAL) {
  readDHT11Values();
  lastDHT11ReadTime = millis();

    if (!isnan(data.h)){
      if (data.h > 50.0){
      handle_HumidityPercentageWarning();
      }
    }
  }
}
else{
  data.t = tp;
}

if (!isnan(data.t) && modeStatus) {
  if (data.t < data.setTemperature && transistorStatus == LOW) {
    Serial.println(F("Heating is getting switched on!"));
    handle_HeatingOn();
  } else if(data.t >= data.setTemperature && transistorStatus == HIGH){
    Serial.println(F("Heating is getting switched off!"));
    handle_HeatingOff();
  }
}
```

A handle_SetTemperature() függvény végzi el a küszöb értéknek a beállítását ami fölött kapcsol a fűtés, elmenti az értéket az EEPROM-ba, ez az az érték amit a felhasználói felületről adunk meg.

```
void handle_SetTemperature() {
  if (server.hasArg("temperature")) {
    String temperatureStr = server.arg("temperature");
    data.setTemperature = temperatureStr.toFloat();

    EEPROM.put(address, data.setTemperature);
    EEPROM.commit();

    Serial.print("Set Temperature: ");
    Serial.println(data.setTemperature);
  }
  server.send(200, "text/html", SendHTML(LED1Status, transistorStatus, data.t));
}
```

A handle_GetAmbientTemperature() függvény segítségével olvassuk a az eszközről érkező környezeti hőmérsékletet amelyet egy globális változóba tárolunk.

```
void handle_GetAmbientTemperature(){
  if (server.hasArg("ambienttemperature")) {
    String temperatureStr = server.arg("ambienttemperature");
    tp = temperatureStr.toFloat();

    Serial.print("Ambient Temperature: ");
    Serial.println(tp);
  }
  server.send(200, "text/html", SendHTML(LED1Status, transistorStatus, tp));
}
```

A handle_GetTemperature() a kliens kérésére küldi el a DHT11 által mért adatokat csatolva mellé a jelenlegi küszöb értéket megjelenítés céljából.

```
void handle_GetTemperature(){

  const size_t capacity = JSON_OBJECT_SIZE(3);
  DynamicJsonDocument jsonDoc(capacity);
  jsonDoc["temperature"] = data.t;
  jsonDoc["humidity"] = data.h;
  jsonDoc["reftemp"] = data.setTemperature;

  String message;
  serializeJson(jsonDoc, message);

  server.send(200,"application/json", message);
}
```

Magas páratartalom esetén az alábbi függvény hajtódik végre, amely jelzésként elkezdi villogtatni a lapon található LED-t.

```
void handle_HumidityPercentageWarning() {
  digitalWrite(LED1Pin, HIGH);
  delay(100);
  digitalWrite(LED1Pin, LOW);
  Serial.println("High Humidity Percentage");
  server.send(200, "text/html", SendHTML(LED1Status,true, data.t));
}
```

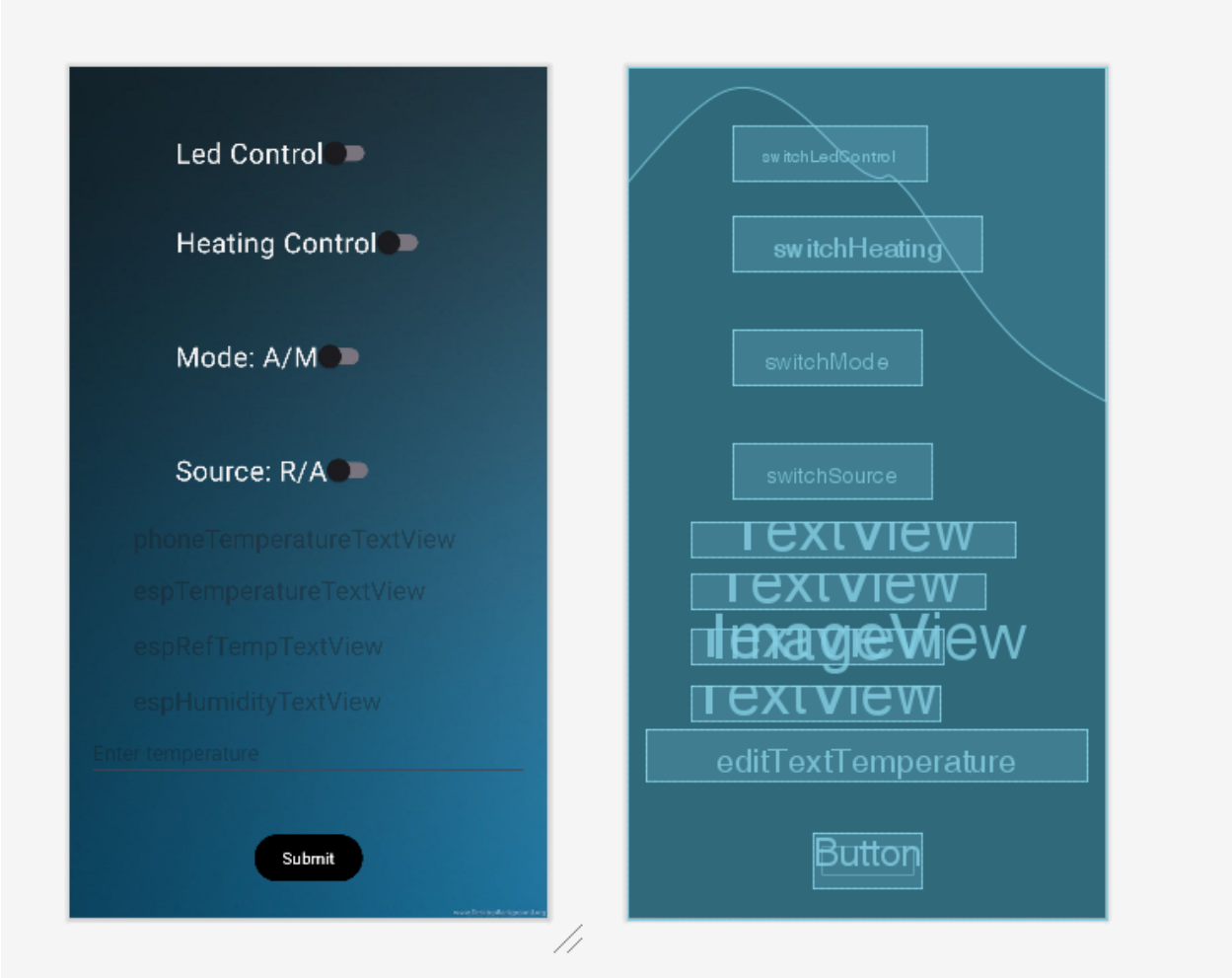
Végezetül pedig itt a readDHT11Values() függvény, amely felhasználja a DHT sensor nevű könyvtárat, amely segítségével beolvassuk az értékeket, eltároljuk egy struktúrába könnyebb feldolgozás véget.

```
SensorData readDHT11Values() {
  data.h = dht.readHumidity();
  data.t = dht.readTemperature();
  if (!isnan(data.h) && !isnan(data.t)) {
    Serial.print("Humidity: ");
    Serial.print(data.h);
    Serial.print("%  Temperature: ");
    Serial.print(data.t);
    Serial.println("°C");
  } else {
    Serial.println("Failed to read from DHT sensor!");
  }

  return data;
}
```

## 5. Felhasználói interfész programok rövid leírása

A felhasználói interfész tartalmaz 4 kapcsolót, 4 szövegmezőt, 1 beviteli mezőt és egy gombot.




A felhasználó felület  HTTP kéréseken keresztül kommunikál a szerverrel, amelyre a Retrofit Android könyvtárat használtam.

```
interface ApiService {

    @GET("/settemperature")
    fun setTemperature(@Query("temperature") temperature: String): Call<Void>

    @GET("/sendambienttemperature")
    fun sendAmbientTemperature(@Query("ambienttemperature") ambienttemperature: String): Call<Void>

    @GET("/setmode")
    fun setMode(@Query("mode") mode: String): Call<Void>

    @GET("/setsource")
    fun setSource(@Query("mode") mode: String): Call<Void>

    @GET
    suspend fun sendRequest(@Url url: String): Call<Void>
}
```

Egy kérés a következőképpen néz ki:

```
private suspend fun sendRequest(route: String) {
    try {

        val url = URL(serverUrl)
        val retrofit = Retrofit.Builder() Retrofit.Builder
            .baseUrl(url) Retrofit.Builder
            .addConverterFactory(GsonConverterFactory.create())
            .build()

        val apiService = retrofit.create(ApiService::class.java)
        val call = apiService.sendRequest(route)

        call.enqueue(object : Callback<Void> {...})
    } catch (e: Exception) {
        // Handle the exception
        Log.d( tag: "MIAU",  msg: "Network error: ${e.message}")
    }
}
```

A szenzor adatok lekérése pedig így:

```
    private fun getRoomTemperature(callback: (String) -> Unit) {
        val route = "$serverUrl/gettemperature"

        class SensorDataTask : AsyncTask<Void, Void, String>() {

            override fun doInBackground(vararg params: Void?): String {
                var result = ""
                try {
                    val url = URL(route)
                    val connection = url.openConnection() as HttpURLConnection
                    connection.requestMethod = "GET"

                    val reader = BufferedReader(InputStreamReader(connection.inputStream))
                    result = reader.readLine()
                    reader.close()
                    connection.disconnect()
                } catch (e: Exception) {
                    e.printStackTrace()
                }
                return result
            }
```

A UI 5 másodpercenként frissül ami az alábbi módon van megvalósítva.A 2 függvény egy-egy Get kérést hajt végre s annak eredményét helyezi el a felületen látható szöveg dobozokba.

```
lifecycleScope.launch { this: CoroutineScope
    while (true) {
        updatePhoneTemperature()
        updateRoomTemperature()
        delay( timeMillis: 5000)
    }
}
```

A kapcsolókra megfigyelők vannak állítva amelyek interakció hatására küldenek kérést a szerver felé.

```
// Set up mode switch
binding.switchMode.setOnCheckedChangeListener { _, isChecked ->
    val mode = if (isChecked) "manual" else "automatic"
    modeStatus = mode == "manual"
    binding.switchHeating.isEnabled = modeStatus

    lifecycleScope.launch { this: CoroutineScope
        sendModeRequest(mode)
    }
}

// Set up source switch
binding.switchSource.setOnCheckedChangeListener { _, isChecked ->
    val mode = if (isChecked) "ambient" else "room"
    sourceStatus = mode == "room"

    lifecycleScope.launch { this: CoroutineScope
        sendSourceRequest(mode)
    }
}

// Set up LED switch
binding.switchLedControl.setOnCheckedChangeListener { _, isChecked ->
    val action = if (isChecked) "led1on" else "led1off"
    lifecycleScope.launch { this: CoroutineScope
        sendRequest( route: "/$action")
    }
}

// Set up heating switch
binding.switchHeating.isEnabled = false
binding.switchHeating.setOnCheckedChangeListener { _, isChecked ->
    val action = if (isChecked) "heatingon" else "heatingoff"
    lifecycleScope.launch { this: CoroutineScope
        sendRequest( route: "/$action")

    }
}
```

A felhasználói felületen magas páratartalom esetén meghívódik az alábbi függvény, amely küld egy értesítést és 10 percenként küld még egyet ha fent áll az adott probléma.

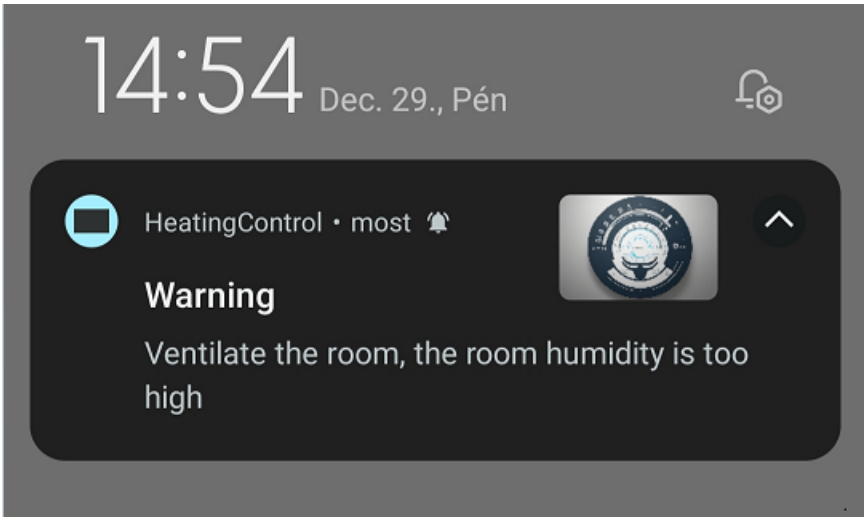
```
private fun createAndShowNotification() {
    val largeIconBitmap: Bitmap = BitmapFactory.decodeResource(resources, R.drawable.app_logo2)
    // Notification format
    val notificationBuilder = NotificationCompat.Builder( context: this, channelId)
        .setSmallIcon(R.drawable.app_logo2)
        .setLargeIcon(largeIconBitmap)
        .setContentTitle("Warning")
        .setContentText("Ventilate the room, the room humidity is too high")
        .setPriority(NotificationCompat.PRIORITY_DEFAULT)

    // Unique Id for notification
    val notificationId = Random.nextInt()

    // Showing the notification

    Handler(Looper.getMainLooper()).post {
        val notificationManager = NotificationManagerCompat.from( context: this)
        notificationManager.notify(notificationId, notificationBuilder.build())
    }

}
```



# 6. Programok forráskódja, dokumentációja

- A teljes projekt forráskódja megtalálható az alábbi linken:
  - https://github.com/Tomi08/HeatingControl
- További linkek:
  - https://square.github.io/retrofit/
  - https://randomnerdtutorials.com/getting-started-with-esp32/
  - https://randomnerdtutorials.com/esp32-web-server-arduino-ide/