Integrált vezérlőrendszer – koncepcionális alapok
A rendszer kialakításának célja

A projekt elektronikai–szoftveres szakaszában tudatos tervezési döntés született: a rendszer nem egyetlen mikrokontrollerre épülő, monolitikus eszköz, hanem elosztott vezérlőarchitektúra formájában valósul meg.
Fontos hangsúlyozni, hogy ez nem az ESP32 „gyengesége” miatt történt. Az ESP32 kiváló terepi vezérlő és adatgyűjtő egység: stabil digitális I/O kezelést biztosít, több kommunikációs interfésszel rendelkezik (I^2C, SPI, UART, WiFi), és önálló vezérlési logika futtatására is alkalmas. Amennyiben a GPIO-k száma nem elegendő, ipari gyakorlatnak megfelelően I/O bővítők (pl. I^2C port expanderek), vagy további csomópontok (node-ok) felfűzése alkalmazható — ez már önmagában az elosztott rendszerek irányába mutat.
A valódi korlát a fejlesztés során nem az I/O vagy a kommunikáció területén jelentkezett, hanem a felügyeleti és megjelenítési (HMI) funkciók megvalósításakor. Az ESP32:

- korlátozott RAM és flash tárterülettel rendelkezik,
- nem webes felhasználói felületek, adatbáziskezelés és tartós naplózás futtatására optimalizált,
- komplex grafikus megjelenítés és több kliens kiszolgálása esetén gyorsan erőforrás-határhoz ér.

Ez a gyakorlatban azt eredményezte, hogy bár az ESP32 képes volt a terepi feladatokat stabilan ellátni (szenzorok, aktorok, lokális vezérlés), a rendszer felügyeleti rétege már indokolttá tette egy nagyobb számítási kapacitású eszköz bevonását.
A kialakított architektúra így világosan elkülönülő rétegekre bontja a rendszert:

Szint	Funkció	Eszköz
Terepi szint	Szenzoradat-gyűjtés, aktorvezérlés, lokális logika	ESP32
Vezérlési / adatkezelési szint	Ciklikus feldolgozás, logikai döntések, Modbus master szerep	Raspberry Pi / PC (Modbus)
Felügyeleti szint (HMI)	Webes megjelenítés, felhasználói vezérlés, adatnaplózás	Raspberry Pi / PC (Flask alapú szerver)

Ebben a modellben az ESP 32 terepi csomópont (field node) szerepkörben működik, és ipari mintát követve szabványos protokollon (Modbus TCP) keresztül szolgáltat adatokat a magasabb szintű rendszernek. Fontos, hogy egy tisztán monolitikus, egyeszközös rendszerben erre a kommunikációs szabványra valóban nem lenne szükség — az elosztott architektúra azonban indokolttá teszi.
A projekt célja tehát nem pusztán egy működő okosfarm-modell létrehozása volt, hanem annak demonstrálása, hogy:

- hogyan válik egy mikrokontroller terepi csomóponttá,
- hogyan szerveződik fölé egy felügyeleti rendszeroldali logikai és HMI réteg,
- és hogyan modellezhető oktatási környezetben az ipari automatizálás rétegzett, hálózati felépítése.
-

Ez a szemlélet közelíti a tanulók számára a valós ipari rendszerek működését, miközben a rendszer továbbra is könnyen bővíthető új szenzorokkal, csomópontokkal és szolgáltatásokkal.

Az elosztott rendszer koncepciója

A projektben kialakított vezérlési modell tudatosan egy két szintű, rétegzett architektúrát követ. Ez a felépítés nemcsak technikai döntés, hanem szemléletbeli lépés is volt a monolitikus, „mindent egy mikrokontrolleren” megközelítéstől az ipari rendszerekre jellemző struktúra felé.

Az alsó szintet a terepi/mezőszint (field level) alkotja, ahol az érzékelők és beavatkozók közvetlen kezelése történik. Ezt a szerepet a rendszerben az ESP32 tölti be. Feladatai közé tartozik a digitális és analóg jelek kezelése, a szenzoradatok begyűjtése, az alapvető előfeldolgozás, valamint a fizikai kimenetek (pl. LED, relé, szervó) működtetése. Ezen a szinten zajlik a valós fizikai folyamattal való közvetlen kapcsolat.

A felső réteget a felügyeleti szint (supervisory level) jelenti, amelyet a projektben egy Raspberry Pi (vagy PC) alapú rendszer valósít meg. Fontos különbség, hogy ezen a szinten nem a központi vezérlési logika fut, hanem a rendszer digitális állapotképe jelenik meg. A Pi a mezőszintű eszközök (ESP32) adatait gyűjti, naplózza és HMI webes felületen megjeleníti, valamint lehetőséget ad távoli beavatkozásra.

A tényleges vezérlés a mezőszinten, az ESP32-n történik. Az I/O kezelés, a szenzoradatok feldolgozása és az alap működési logika helyben fut, ezért a rendszer offline módban is működőképes marad. A felügyeleti réteg kiesése nem állítja le a fizikai folyamatot, csak a megjelenítést és a távoli elérést érinti.

Ez a struktúra így pontosabban a következő ipari modellhez hasonlítható:

A projektben ennek megfelelői:

- ESP32 mint intelligens terepi egység, amely
  - I/O kezelést végez
  - szenzoradatokat gyűjt
  - lokális vezérlési logikát futtat
  - Modbus felügyeleti rendszerként kommunikál
- Raspberry Pi + Python + Flask mint felügyeleti réteg, amely
  - a rendszer állapotát tükrözi
  - adatokat gyűjt és megjelenít
  - operátori beavatkozást tesz lehetővé
  - de nem kritikus a folyamat működéséhez

A jövőben a Raspberry Pi szinten megvalósíthatók olyan kiegészítő funkciók, mint például:

- adatnaplózás és hosszú távú trendanalízis
- grafikonos megjelenítés
- felhasználói jogosultságkezelés
- értesítések, riasztási naplók
- távoli konfigurációs felület

Ezek a funkciók csak felügyeleti és információs jellegűek lehetnek. Tervezési alapelv, hogy a Raspberry Pi-n futó kiegészítések nem befolyásolhatják a rendszer alapműködését vagy biztonságát. A kritikus vezérlési logika és az alapállapotok kezelése továbbra is a terepi szinten marad.

Az elosztott architektúra műszaki előnyei

A rendszer egyik kulcsa a funkcionális szétválasztás, amely közvetlenül növeli a megbízhatóságot és a bővíthetőséget. Az ESP32 a terepi szinten marad: szenzorok, bemenetek, kimenetek és az alap vezérlési logika helyben fut, így a fizikai folyamat működése nem függ folyamatos hálózati kapcsolattól. Ez determinisztikusabb I/O viselkedést és stabilabb működést eredményez.

A felső szint feladata a felügyelet, adatkezelés és megjelenítés, nem pedig a kritikus vezérlés. Ez a rétegzett felépítés ipari mintát követ, ahol a terepi eszköz, a vezérlési szint és a felügyeleti rendszer külön szerepkörben működik.

Az architektúra jelentős előnye a skálázhatóság: új ESP-alapú node-ok illeszthetők a hálózathoz, további szenzorok integrálhatók, és a rendszer funkcionálisan bővíthető anélkül, hogy egyetlen eszköz erőforrásaira terhelnénk mindent. Ez a horizontális bővítés az elosztott rendszerek alapvető sajátossága.

A megbízhatóság szempontjából alapelv, hogy a mezőszint autonóm marad: a felügyeleti rendszer vagy a hálózat kiesése nem állítja le a helyi működést, csak a monitorozást és a távoli beavatkozást érinti. Ez közvetlenül megfelel az ipari automatizálás gyakorlatának.

Kommunikációs modell és szemlélet

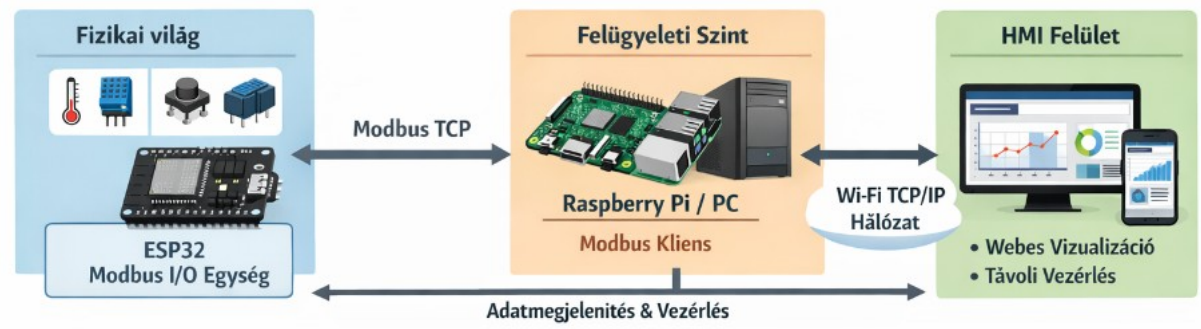
A rétegek közötti kapcsolat Modbus TCP alapú. Ez állapotalapú adatcserét valósít meg, szabványos regiszterstruktúrával. A szenzoradatok és vezérlőjelek nem konkrét „kérések–válaszok” formájában mozognak, hanem jól definiált címeken elérhető állapotokként. Ez eltér a tipikus HTTP-alapú, tranzakciós szemlélettől, és közelebb áll az ipari kommunikációs modellekhez, ahol a vezérlő ciklikusan olvassa a mezőszint állapotait, majd ennek megfelelően írja a kimeneti regisztereket.

A rendszer váza – logikai felépítés

Magas szinten a rendszer adatútja a következőképpen írható le:

A szenzorok (gomb, DHT stb.) és az aktorok a fizikai világban az ESP32-höz csatlakoznak. Az ESP32 Modbus felügyeleti rendszerként működik, és regisztereken keresztül teszi elérhetővé a bemenetek és kimenetek állapotát. A WiFi-alapú TCP hálózaton keresztül a felügyeleti szinten futó rendszer – Raspberry Pi vagy PC – Modbus kliensként ciklikusan olvassa és írja ezeket az adatokat. A feldolgozott állapotok és mért értékek ezután a HMI webes felületen jelennek meg, ahol vizualizáció és távoli vezérlés is megvalósul.

Ebben a modellben az ESP nem központi vezérlő, hanem intelligens terepi I/O egység, míg a Raspberry Pi tölti be a logikai és felügyeleti központ szerepét.



A rendszer váza (Forrás: AI generált)

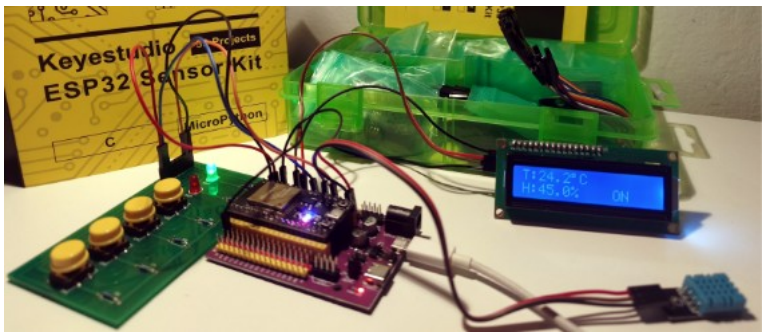
Oktatási jelentőség

Az alkalmazott architektúra nemcsak műszaki megoldás, hanem szemléletformáló eszköz. A tanulók megértik a terepi és a felügyeleti szint szétválasztásának okát, az erőforrás-elosztás szerepét, valamint azt, hogyan épül fel egy ipari automatizálási rendszer rétegzett struktúrája. Ez közvetlen alapot ad a PLC-programozási, HMI-tervezési és SCADA-rendszerek megértéséhez, mert a projektben alkalmazott modell ezek leegyszerűsített, de valósághű leképezése.
A következő részben az architektúra konkrét szoftveres megvalósítását mutatjuk be: az ESP terepi programját, a felügyeleti oldali ciklikus feldolgozást, valamint a webes kezelőfelület működését.

Terepi vezérlőcsomópont – önálló működésű mintaegység

A projekt elektronikai rendszerének alsó szintjét egy önállóan működő terepi csomópont (field node) képviseli, amely egy ESP32 mikrokontrollerre épül. Ez az egység közvetlenül kapcsolódik a fizikai világhoz: érzékelőket olvas, beavatkozókat vezérel, és helyi visszajelzést biztosít. A bemutatott konfiguráció tudatosan egyszerű, oktatási célú minta, amely a rendszer alaplogikáját szemlélteti, nem pedig végleges kiépítést jelent.
A csomópont a következő elemeket kezeli:

- egy digitális bemenetként működő nyomógomb,
- egy digitális kimeneti LED,
- egy DHT11 hőmérséklet- és páratartalom-érzékelő,
- egy I²C buszon csatlakozó, 16×2 karakteres LCD kijelző.



Tesztkörnyezet (Forrás: Saját szerkesztés)

A működés lényege, hogy az eszköz teljesen önállóan, hálózati kapcsolat nélkül is ellátja alapfeladatait. A felhasználó a fizikai gombbal képes beavatkozni a rendszer állapotába (LED ki-/bekapcsolás), miközben a környezeti adatok folyamatosan mérésre kerülnek és megjelennek a kijelzőn. Ez a megközelítés azt az ipari alapelvet modellezi, hogy a terepi szintnek képesnek kell lennie autonóm működésre, függetlenül a felügyeleti rendszertől.
A bemutatott program tehát egy intelligens I/O egység viselkedését demonstrálja. A struktúra moduláris: új szenzorok, kimeneti eszközök vagy kijelzők egyszerűen hozzáadhatók további GPIO-k vagy kommunikációs buszok felhasználásával. A későbbiekben ez a csomópont magasabb szintű rendszerekhez (PLC, HMI, SCADA) is illeszthető, de jelen fejezetben kizárólag a helyi erőforrások kezelésére koncentrálunk.

A program felépítése funkcionális egységekre bontva

Könyvtárak és hardverdefiníciók

```
// ================= KÖNYVTÁRAK BETÖLTÉSE =================
// Ebben a szakaszban történik a felhasznált perifériákhoz szükséges könyvtárak betöltése.
// DHT könyvtár a szenzor kommunikációját valósítja meg.
// A Wire és hd44780_I2Cexp az I²C LCD vezérléséhez szükséges.
#include <DHT.h>
#include <Wire.h>
#include <hd44780.h>
#include <hd44780ioClass/hd44780_I2Cexp.h>


// ================= HARDVER KIOSZTÁS =================
//A #define direktívák rögzítik a fizikai bekötés és a program közötti kapcsolatot, így a hardverkonfiguráció egyértelműen dokumentált és könnyen módosítható.

#define BTN 17      // Fizikai gomb bemenet
#define LED 2       // LED kimenet
#define DHTPIN 4    // DHT adatvonal
#define DHTTYPE DHT11 //DHT típus megadása
```

Ebben a szakaszban történik a felhasznált perifériákhoz szükséges könyvtárak betöltése.

- A DHT könyvtár a szenzor kommunikációját valósítja meg.
- A Wire és hd44780_I2Cexp az I²C LCD vezérléséhez szükséges.

A #define direktívák rögzítik a fizikai bekötés és a program közötti kapcsolatot, így a hardverkonfiguráció egyértelműen dokumentált és könnyen módosítható.

Objektumok és globális változók

```
//Itt jönnek létre a szenzor- és kijelzőobjektumok. Ezek a program teljes futása alatt elérhetők, így bármelyik ciklusban használhatók.
// Szenzor objektum
DHT dht(DHTPIN, DHTTYPE);
// I2C LCD objektum
hd44780_I2Cexp lcd;
```

Itt jönnek létre a szenzor- és kijelzőobjektumok. Ezek a program teljes futása alatt elérhetők, így bármelyik ciklusban használhatók.

Gomb prellegés (debounce) kezelése

```
// ================= GOMB PRELLEGÉS SZŰRÉS =================
// A mechanikus gombok érintkezői záráskor rövid ideig „rezegnek” (prellegnek), ami több hamis jelváltást okozhat.
// Ez a logika időalapú szűréssel biztosítja, hogy csak a stabil állapotváltás legyen érvényes.

bool lastReading = false;      // Legutóbbi nyers bemenet
bool stableState = false;      // Szűrt, stabil állapot
unsigned long lastChangeTime = 0;
const unsigned long debounceTime = 50;  // 50 ms stabilitási küszöb
```

A mechanikus gombok érintkezői záráskor rövid ideig „rezegnek” (prellegnek), ami több hamis jelváltást okozhat. Ez a logika időalapú szűréssel biztosítja, hogy csak a stabil állapotváltás legyen érvényes.

LED állapot és mérési változók

```
// A LED állapota külön változóban tárolódik. A DHT mérések időzítéséhez és az aktuális értékek tárolásához is globális változók szükségesek.
bool ledState = false;
unsigned long lastDHTread = 0; // Utolsó olvasás időpontja
float temp = 0;                // Hőmérséklet cache
float hum = 0;                 // Páratartalom cache
```

A LED állapota külön változóban tárolódik. A DHT mérések időzítéséhez és az aktuális értékek tárolásához is globális változók szükségesek.

Inicializálás – setup()

```
50  void setup() {
51
52    // Hardver inicializálás
53    pinMode(BTN, INPUT);
54    pinMode(LED, OUTPUT);
55
56    // DHT inicializálás
57    dht.begin();
58
59    // I2C inicializálás ESP32 alapértelmezett lábakkal
60    Wire.begin(21,22);
61
62    // LCD inicializálás
63    lcd.begin(16,2);
64    lcd.backlight();
65
66    lcd.setCursor(0,0);
67    lcd.print("Smart Farm Node");
68    delay(2000);
69    lcd.clear();
```

A setup() egyszer fut le induláskor. Itt történik:

- a GPIO irányok beállítása,
- a szenzor és az I²C busz inicializálása,
- az LCD indítása.

Gombkezelés és LED váltás

Ez a blokk valósítja meg a lenyomási élre történő LED-váltást. A LED csak akkor változik, amikor a gomb stabilan lenyomott állapotba kerül.

```
74    //Ez a blokk valósítja meg a lenyomási élre történő LED-váltást.
75    //A LED csak akkor változik, amikor a gomb stabilan lenyomott állapotba kerül.
76    // Gomb olvasása
77    bool reading = digitalRead(BTN);
78
79    // Prellegés detektálása
80    if (reading != lastReading) {
81      lastChangeTime = millis();
82    }
83
84    // Stabil állapot vizsgálata
85    if ((millis() - lastChangeTime) > debounceTime) {
86
87      // Állapotváltozás kezelése
88      if (reading != stableState) {
89        stableState = reading;
90
91        // Lenyomás detektálás
92        if (stableState == true) {
93          ledState = !ledState;         // LED kapcsolás
94        }
95      }
96    }
97
98    lastReading = reading;
```

DHT11 periodikus mérés

```
100   // DHT szenzor olvasás időzítetten
101   //A szenzor csak 2 másodpercenként kerül kiolvasásra, ami megfelel a DHT11 időzítési követelményeinek.
102   if(millis() - lastDHTread > 2000){
103     lastDHTread = millis();
104
105     float h = dht.readHumidity();
106     float t = dht.readTemperature();
107
108     // Hibás olvasás kizárása
109     if(!isnan(h) && !isnan(t)){
110       temp = t;
111       hum = h;
112     }
113   }
```

A szenzor csak 2 másodpercenként kerül kiolvasásra, ami megfelel a DHT11 időzítési követelményeinek.

LCD kijelzés

```
115     // LCD kijelzés
116     //A kijelző a mért adatokat és a LED állapotát jeleníti meg, így a rendszer állapota hálózat nélkül is
        ellenőrizhető.
117     lcd.setCursor(0,0);
118     lcd.print("T:");
119     lcd.print(temp,1);
120     lcd.print((char)223);
121     lcd.print("C ");
122
123     lcd.setCursor(0,1);
124     lcd.print("H:");
125     lcd.print(hum,1);
126     lcd.print("% ");
127
128     lcd.setCursor(11,1);
129     lcd.print(ledState ? "ON " : "OFF");
```

A kijelző a mért adatokat és a LED állapotát jeleníti meg, így a rendszer állapota hálózat nélkül is ellenőrizhető.

Összegzés

Ez a program egy autonóm terepi vezérlőegység működését demonstrálja:

- fizikai bemenet feldolgozás
- aktorvezérlés
- szenzoradat-gyűjtés
- helyi vizualizáció

A kód szerkezete jól elkülöníti az egyes funkciókat, ami megkönnyíti a későbbi bővítést és magasabb szintű rendszerekhez való csatlakoztatást.

Terepi vezérlőcsomópont hálózati bővítéssel – WiFi + Modbus TCP

Ebben a verzióban a korábban bemutatott önálló terepi vezérlő kiegészül hálózati képességekkel.

A helyi működés (gomb → LED, DHT mérés, LCD kijelzés) változatlanul megmarad, viszont az eszköz már képes:

- WiFi hálózatra csatlakozni
- Modbus TCP felügyeleti rendszerként működni
- a fizikai bemenetek és kimenetek állapotát regisztereken keresztül elérhetővé tenni
- külső rendszer (PLC / felügyeleti szoftver) által történő távoli olvasásra és beavatkozásra

Fontos, hogy a logika nem költözik ki a terepi eszközről: a helyi vezérlés továbbra is autonóm, a hálózat csak állapotmegosztásra és távoli beavatkozásra szolgál.

A program felépítése funkcionális egységek szerint

Könyvtárak és hardverdefiníciók

```
#include <WiFi.h>           // ESP32 WiFi stack
#include <ModbusTCP.h>      // Modbus TCP szerver kezelése
#include <DHT.h>            // DHT szenzor magas szintű kezelése (protokollt elrejti)
#include <Wire.h>           // I2C kommunikáció (LCD miatt)
#include <hd44780.h>        // LCD vezérlő
#include <hd44780ioClass/hd44780_I2Cexp.h> // I2C LCD expander kezelés

ModbusTCP mb;  // Az ESP Modbus TCP szerverének létrehozása.


// ================= HARDVER KIOSZTÁS =================
// GPIO-k definiálása hardver absztrakcióként

#define BTN 17         // Fizikai gomb bemenet
#define LED 2          // Fő LED kimenet (vezérelt fogyasztó)
#define WIFI_LED 5     // WiFi státusz visszajelző LED (terepi hibajelzés)
#define DHTPIN 4       // DHT adatvonal
#define DHTTYPE DHT11 // Szenzor típusa

// Szenzor objektum létrehozása
DHT dht(DHTPIN, DHTTYPE);

// I2C LCD objektum létrehozása (automatikus I2C címfelismerés)
hd44780_I2Cexp lcd;
```

Hálózati paraméterezés

```
// ================= WIFI HÁLÓZATI PARAMÉTEREK =================
// Ezek konfigurálják a hálózati kapcsolatot, valamint biztosítják,
// hogy az ESP mindig ugyanazon IP címen legyen elérhető.

const char* ssid = "plc";
const char* pass = "12345678";

IPAddress local_IP(192, 168, 137, 200);
IPAddress gateway(192, 168, 137, 1);
IPAddress subnet(255, 255, 255, 0);
IPAddress primaryDNS(8,8,8,8);
IPAddress secondaryDNS(8,8,4,4);


// ================= WIFI ÚJRACSATLAKOZÁSI LOGIKA =================
// A rendszer nem blokkoló reconnect stratégiát használ.
// Ha a kapcsolat megszakad, a program nem áll meg,
// hanem meghatározott időközönként újrapróbál csatlakozni.

unsigned long lastReconnectAttempt = 0;         // Utolsó reconnect próbálkozás időpontja
const unsigned long reconnectInterval = 300000; // 5 perc újrapróbálkozási ciklus
bool wifiConnected = false;                     // Hálózati állapot logikai jelző

// WiFi csatlakozási rutin
// Ez a függvény bontja az előző kapcsolatot,
// majd új kapcsolatot kezdeményez a megadott SSID-re.
void connectToWiFi() {
  WiFi.disconnect(true);
  delay(100);               // Rövid stabilizációs várakozás
  WiFi.begin(ssid, pass);
}


// ================= WIFI HIBA LED IDŐZÍTÉS =================
// Amennyiben nincs WiFi kapcsolat,
// a WIFI_LED másodperces periódussal villog.
// A megvalósítás millis() alapú, tehát nem blokkolja a fő ciklust.

unsigned long lastBlinkTime = 0;     // Utolsó LED állapotváltás
const unsigned long blinkInterval = 1000; // 1 másodperces villogási periódus
bool wifiLedState = false;           // WiFi LED aktuális állapota
```

Gomb prellegés (debounce) kezelése, LED állapot és mérési változók

```
// ================ BELSŐ ÁLLAPOT RÉTEG ================
// A rendszer egyetlen "igazságforrást" használ a LED állapotára.
// A fizikai gomb, a Modbus és a fizikai LED is ezt az állapotot követi.

bool ledState = false;   // A LED központi logikai állapota


// ================ GOMB PRELL SZŰRÉS ================
// Mechanikus gomb érintkezése záráskor pattog (prell).
// A stabil állapot detektálás millis alapú időszűréssel történik.

bool lastReading = false;       // Utolsó nyers bemeneti érték
bool stableState = false;       // Szűrt, stabil állapot
unsigned long lastChangeTime = 0;
const unsigned long debounceTime = 50;  // 50 ms stabilitási küszöb


// ================ DHT MINTAVÉTELI IDŐZÍTÉS ================
// A DHT szenzor olvasása nem történhet túl gyakran,
// ezért 2 másodperces mintavételezési ciklust alkalmazunk.

unsigned long lastDHTread = 0;
float temp = 0;
float hum = 0;
```

Inicializálás

```
// ==================== SETUP ====================
// ===============================================
void setup() {

  Serial.begin(115200);

  // Hardver inicializálás
  pinMode(BTN, INPUT);
  pinMode(LED, OUTPUT);
  pinMode(WIFI_LED, OUTPUT);

  // ===== Stabil statikus IP konfiguráció =====
  // Az ESP32 WiFi stack sajátossága miatt:
  // - Station mód explicit beállítása szükséges
  // - DHCP cache törlése
  // - Statikus IP konfigurálása csatlakozás előtt

  WiFi.mode(WIFI_STA);
  WiFi.persistent(false);
  WiFi.disconnect(true);
  delay(100);

  WiFi.config(local_IP, gateway, subnet, primaryDNS, secondaryDNS);
  WiFi.begin(ssid, pass);

  // Induláskori maximum 10 másodperces várakozás.
  // Ha ezalatt nem jön létre kapcsolat,
  // a program offline módban folytatódik.
  unsigned long startAttemptTime = millis();
  while (WiFi.status() != WL_CONNECTED &&
         millis() - startAttemptTime < 10000) {
    delay(500);
  }

  wifiConnected = (WiFi.status() == WL_CONNECTED);

  // Szenzor és perifériák inicializálása
  dht.begin();
  Wire.begin(21,22);

  lcd.begin(16,2);
  lcd.backlight();
  lcd.setCursor(0,0);
  lcd.print("Smart Farm Node");
  delay(2000);
  lcd.clear();
```

Modbus regiszterkiosztás

```
144     // ================= MODBUS REGISZTERKIOSZTÁS =================
145     // Coil 0  → LED állapot
146     // Ists 0  → Gomb állapot
147     // Hreg 0  → Hőmérséklet (x10 skálázva)
148     // Hreg 1  → Páratartalom (x10 skálázva)
149
150     mb.server();
151     mb.addCoil(0);
152     mb.addIsts(0);
153     mb.addHreg(0);
154     mb.addHreg(1);
155
156     // Coil inicializálása a belső logikai állapotból
157     mb.Coil(0, ledState);
158 }
```

WIFI felügyelet, Modbus kiszolgálás aktív kapcsolat mellett

```
164 void loop() {
165
166   // ===== WIFI FELÜGYELET =====
167   // Nem blokkoló hálózati állapotellenőrzés.
168   // Ha nincs kapcsolat, 5 percenként újrapróbál.
169
170   if (WiFi.status() != WL_CONNECTED) {
171
172     wifiConnected = false;
173
174     if (millis() - lastReconnectAttempt >= reconnectInterval) {
175       lastReconnectAttempt = millis();
176       connectToWiFi();
177     }
178   }
179   else {
180     wifiConnected = true;
181   }
182
183   // ===== WIFI HIBA LED KEZELÉS =====
184   // Offline állapotban 1 Hz villogás.
185   // Online állapotban LED kikapcsolva.
186
187   if (!wifiConnected) {
188
189     if (millis() - lastBlinkTime >= blinkInterval) {
190       lastBlinkTime = millis();
191       wifiLedState = !wifiLedState;
192       digitalWrite(WIFI_LED, wifiLedState);
193     }
194   }
195   else {
196     digitalWrite(WIFI_LED, LOW);
197     wifiLedState = false;
198   }
199
200   // ===== MODBUS KISZOLGÁLÁS =====
201   // Csak aktív hálózat esetén kezel TCP kéréseket.
202   if (wifiConnected) {
203     mb.task();
204   }
205
206   // ================================================
207   // ===== MODBUS → BELSŐ ÁLLAPOT SZINKRON =====
208   // ================================================
209   // Ha külső rendszer írja a Coil 0-t,
210   // a belső állapot frissül.
211
212   bool coilState = mb.Coil(0);
213
214   if (coilState != ledState) {
215     ledState = coilState;
216   }
```

Gomb olvasása, LED állapot vezérlése és távoli monitorozása

```
// ================================================
// ===== GOMB → BELSŐ ÁLLAPOT =====
// ================================================
// Fizikai gombnyomás esetén a belső állapot toggolódik,
// majd a Coil regiszter tükrözi azt.

bool reading = digitalRead(BTN);

if (reading != lastReading) {
  lastChangeTime = millis();
}

if ((millis() - lastChangeTime) > debounceTime) {

  if (reading != stableState) {
    stableState = reading;

    if (stableState == true) {
      ledState = !ledState;
      mb.Coil(0, ledState);
    }
  }
}

lastReading = reading;

// ================================================
// ===== FIZIKAI LED VEZÉRLÉS =====
// ================================================
// A fizikai kimenet mindig a belső állapotot követi.

digitalWrite(LED, ledState);
```

DHT11 periodikus mérése, távoli monitorozása, gombállapot jelentése a Modbuson

```
// ================================================
// ===== DHT IDŐZÍTETT OLVASÁS =====
// ================================================

if (millis() - lastDHTread > 2000) {

  lastDHTread = millis();

  float h = dht.readHumidity();
  float t = dht.readTemperature();

  if (!isnan(h) && !isnan(t)) {
    mb.Hreg(0, (int)(t * 10));
    mb.Hreg(1, (int)(h * 10));
    temp = t;
    hum = h;
  }
}

// Gomb státusz jelentése Modbuson
mb.Ists(0, stableState);
```

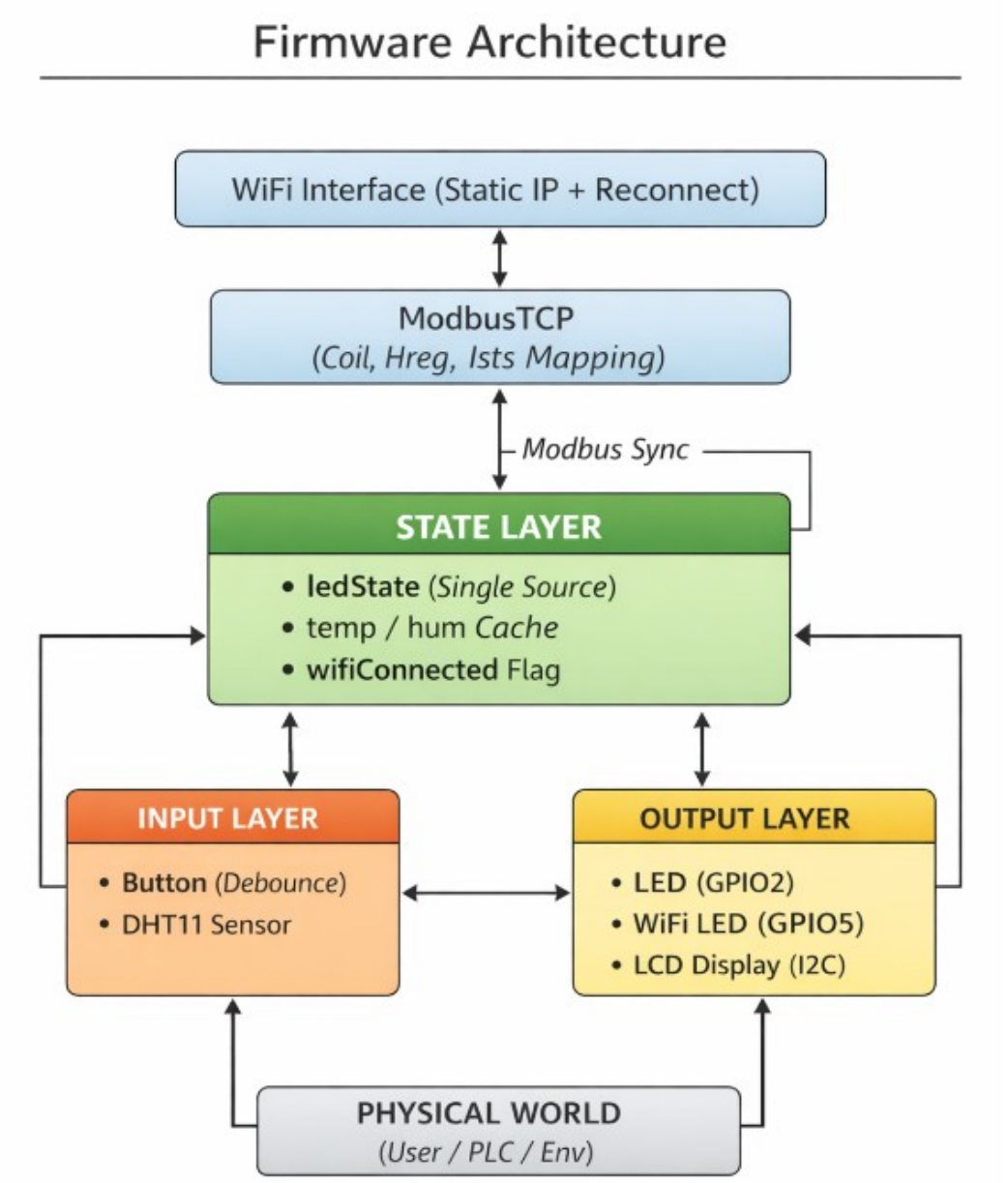
LCD kijelzés

```
273     // ================================================
274     // ===== LCD KIJELZÉS =====
275     // ================================================
276     // Első sor: Hőmérséklet + Hálózati állapot
277     // Második sor: Páratartalom + LED állapot
278
279     lcd.setCursor(0,0);
280     lcd.print("T:");
281     lcd.print(temp,1);
282     lcd.print((char)223);
283     lcd.print("C ");
284     lcd.print(wifiConnected ? "N-ON " : "N-OFF");
285
286     lcd.setCursor(0,1);
287     lcd.print("H:");
288     lcd.print(hum,1);
289     lcd.print("% ");
290     lcd.print(ledState ? "L-ON " : "L-OFF");
291
292     delay(10);  // CPU terhelés csökkentése
293 }
```

A rendszer blokkdiagramja

A blokkdiagram az ESP32-alapú firmware réteges, PLC-szerű architektúráját szemlélteti. A felső szinten a WiFi interfész és a ModbusTCP kommunikáció található, amelyek kizárólag a hálózati adatcserét és a regisztertér kezelését végzik. A rendszer központja a State Layer, amely az összes belső állapotváltozót – például a LED állapotát, a mért hőmérsékletet, páratartalmat és a WiFi státuszt – egyetlen igazságforrásként tárolja.

Az Input Layer a fizikai bemeneteket, míg az Output Layer a kimeneteket kezeli, és mindkettő közvetlenül a belső állapotrétegre támaszkodik. A vezérlés teljes egészében az ESP32-n, azaz terepi szinten marad, így a rendszer hálózattól függetlenül autonóm módon működik. A Modbus és a felügyeleti réteg csupán digitális árnyékként tükrözi a belső állapotot, de nem hordozza a vezérlési logikát. Ez a felépítés biztosítja a determinisztikus, offline működést és az ipari rendszerekre jellemző robusztus architektúrát.



A rendszer blokkdiagramja (Forrás: AI generált)

Felügyeleti szint – Python alapú HMI / felügyeleti ciklus

A terepi ESP32 csomópont a szenzorok és aktorok állapotát Modbus regisztereken keresztül elérhetővé teszi. A felső szint feladata nem a fizikai vezérlés átvétele, hanem a rendszer digitális leképezése és felügyelete.

Ez a réteg a projektben egy Python alapú, ciklikusan működő alkalmazás, amely funkcióját tekintve egy HMI/SCADA-jellegű felügyeleti rendszer.

Feladatai:

- a terepi adatok ciklikus lekérdezése
- állapotok tárolása és feldolgozása
- távoli beavatkozási lehetőség biztosítása
- webes megjelenítés kiszolgálása

Python alapú HMI (Forrás: Saját szerkesztés)

A Raspberry Pi alapú felügyeleti rendszer előkészítése és üzembe helyezése

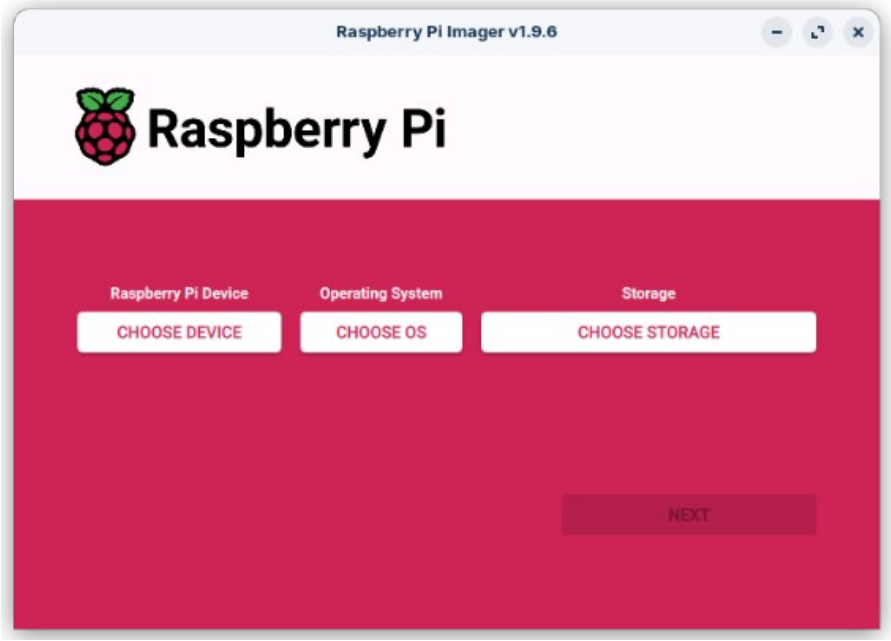
A következő lépcsőfokot a szoftveres felügyeleti logika futtatása jelentette egy önálló, alacsony fogyasztású, ipari környezetben is alkalmazható hardveren. Erre a célra **Raspberry Pi** alapú számítógépet használtunk, amely ideális platformot biztosít a Python alapú, hálózati kommunikációt és webes felületet egyaránt tartalmazó alkalmazások számára. A Raspberry Pi egy jó ár-érték aránnyal rendelkező, viszonylag nagy teljesítményű számítógépet takar, mely rugalmasan illeszthető számos felhasználási területhez. Alapértelmezett esetben egy Debian alapú disztribúciót, a Raspberry OS-t futtat, mely teljesen megfelelt a projekt követelményeinek.

Raspberry Pi (Forrás: https://malnapc.hu/raspberry-pi-5-4gb)

A Raspberry Pi ebben az architektúrában **felügyeleti és vizualizációs rétegként** funkcionál, amely Modbus TCP protokollon keresztül kommunikál az ESP32 alapú alrendszerrel, miközben webes HMI felületet biztosít a felhasználók számára.

Operációs rendszer és alaprendszer telepítése

Az eszköz előkészítése egy stabil, hosszú távon is támogatott Linux alapú operációs rendszer telepítésével kezdődött. Bár lehetőség van egyéni operációs rendszerek telepítésére is, a Raspberry Pi OS biztosította a megfelelő hardvertámogatást, a rendszerstabilitást, valamint a Python fejlesztői környezet egyszerű elérhetőségét. A rendszer telepítését a hivatalos Raspberry Pi Imager programmal végeztük, amely alapvető konfigurációs beállítások megadását követően már egy kész asztali környezetet biztosított a tanulóknak a további munkára.



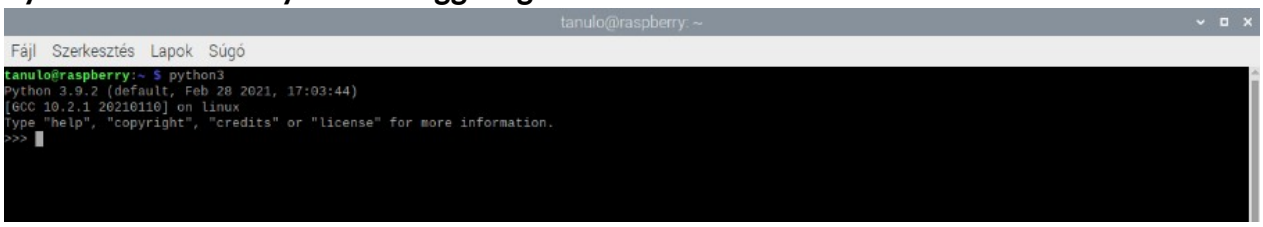
Raspberry Pi OS telepítése (Forrás: Saját szerkesztés)

A rendszertelepítés során megtörtént:

- az alapértelmezett felhasználói fiók konfigurálása,
- a rendszer frissítése,
- valamint a hálózati kapcsolat ellenőrzése.

A megbízható hálózati működés kiemelt fontosságú volt, mivel a teljes felügyeleti logika IP-alapú kommunikációra épül.

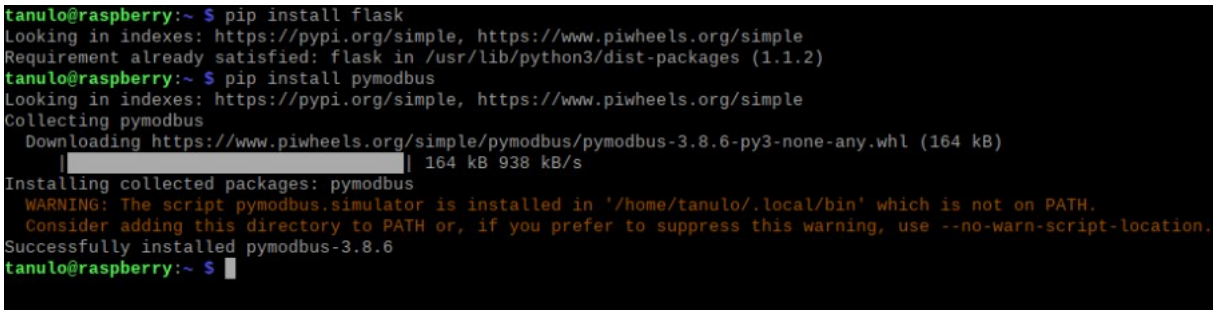
Python futtatókörnyezet és függőségek előkészítése



Mivel a felügyeleti alkalmazás Python nyelven készült, ezért a Raspberry Pi-n egységes és jól karbantartható Python környezet kialakítása történt meg. A rendszerhez tartozó Python 3 verzió biztosította a Flask keretrendszer és a Modbus kommunikációhoz szükséges könyvtárak kompatibilis futtatását.

A környezet előkészítése során hangsúlyt kapott:

- a Python csomagkezelő (PIP) használata,
- a szükséges külső könyvtárak (webszerver, Modbus kliens) telepítése, valamint a verzióütközések elkerülése.



Ez a lépés felhívta a tanulók figyelmét a szoftveres rendszerek reprodukálhatóságára és karbantarthatóságára.

Hálózati konfiguráció és Modbus TCP kapcsolat

A rendszer működésének alapfeltétele a Raspberry Pi és az ESP32 közötti stabil Modbus TCP kapcsolat. Ennek érdekében a Raspberry Pi fix IP-címmel, kiszámítható hálózati környezetben működött, így a felügyeleti alkalmazás mindig elérte az alrendszert.
A tanulók megértették, hogy a hálózati kommunikáció megbízhatósága kulcsfontosságú egy felügyeleti rendszer esetében.
A Modbus kapcsolat ebben a kontextusban nem alacsony szintű vezérlésre, hanem állapotlekérdezésre és parancsközvetítésre szolgált, ami jól illeszkedik a szoftveres PLC-ciklus koncepciójához.

Alkalmazás indítása és párhuzamos működés kezelése

A felügyeleti program indításakor külön szálon fut a ciklikus állapotfrissítést végző logika, míg a Flask alapú webkiszolgáló a felhasználói kéréseket kezeli. Ez a megoldás biztosítja, hogy a webes felület kiszolgálása ne blokkolja a rendszer felügyeleti funkcióit.
A Raspberry Pi erőforrásai elegendőnek bizonyultak:

- a folyamatos Modbus kommunikációhoz,
- a ciklikus állapotfeldolgozáshoz,
- valamint a webes HMI kiszolgálásához.

Ez a működési modell jól demonstrálta a tanulók számára a párhuzamos feldolgozás és a nem valós idejű, de determinisztikus működés közötti különbséget.

Oktatási és rendszerintegrációs tapasztalatok

A Raspberry Pi alapú futtatókörnyezet kialakítása szervesen kapcsolódott a projekt korábbi szakaszaihoz. A tanulók egy olyan komplett rendszert láttak működés közben, amelyben az érzékelők és beavatkozók fizikai szinten jelennek meg, a kommunikáció ipari protokollon keresztül történik, valamint a felügyelet és vezérlés pedig szoftveres eszközökkel valósul meg.
Továbbá a Debian alapú operációs rendszer lehetőséget adott a diákok Linux ismereteinek elmélyítésére. Mivel ipari környezetben nem szükségszerűen grafikus felületen történik ezeknek az eszközöknek a konfigurációja, ezért törekedtünk arra, hogy a diákok minél többet dolgozzanak parancssoros felületen.

A gyakorlati megvalósítás során azonban a diákokkal elértük a Raspberry Pi hardveres korlátait. Amikor a teljes tanulói csapat egyszerre kapcsolódott a webszerverekre, az adatok feldolgozásában lassulás volt tapasztalható. Emellett a többletterhelés szükségessé tette volna aktív hűtés felszerelését, mivel a lapka PC több alkalommal is újraindult túlmelegedés miatt. Bár a projekt szükségleteit kielégítette, fontos tanulságként szolgált a megvalósítás során, hogy egy ipari szintű rendszernél kiemelt szempont kell, hogy legyen a kiszolgáló eszköz erőforrásainak előre tervezése és a várható terhelés felmérése.

A Python kliens szerepe

Az ESP Modbus felügyeleti rendszer, a Python program célja a felügyeleti logika és digitális „árnyék” megvalósítása.

Kapcsolódás az ESP-hez

```python
# ================= FLASK + MODBUS TCP PLC RÉTEG =================
# Ez a program egy "szoftveres PLC ciklust" valósít meg.
# Feladata:
#   • ESP32 Modbus eszköz lekérdezése
#   • LED vezérlési logika kezelése
#   • Szenzoradatok továbbítása a webes HMI felé


# ================= MODULOK IMPORTÁLÁSA =================
from flask import Flask, render_template, jsonify
from pymodbus.client import ModbusTcpClient
import threading, time


# ================= WEB SZERVER OBJEKTUM =================
# Flask kezeli a HTTP kéréseket (HMI felület)
app = Flask(__name__)


# ================= MODBUS KLIENS (ESP32 FELÉ) =================
# Ez a sor létrehozza a TCP kapcsolatot az ESP Modbus felügyeleti rendszerével.
esp = ModbusTcpClient("192.168.0.200", port=502)


# ================= ÁLLAPOTVÁLTOZÓK LÉTREHOZÁSA =================
# Ezek a változók a rendszer állapotát tükrözik a szerver oldalon

led_state = False        # LED logikai állapot (kimenet)
btn_state = False        # Fizikai gomb állapota
prev_btn_state = False   # Előző ciklus gomb állapota (élérzékeléshez)

temperature = 0          # Hőmérséklet cache
humidity = 0             # Páratartalom cache
```

A felügyeleti ciklus szerkezete

```python
# ================= PLC CIKLUS (IPARI LOGIKA MINTÁJÁRA) =================
# Ez a függvény a felügyeleti alkalmazás ciklikusan futó központi része.
# Bár a neve „felügyeleti ciklus”, funkcionálisan nem valós idejű vezérlési ciklust, hanem egy állapotfrissítő, felügyeleti lekérdezési ciklust valósít meg.
# Ez egy végtelen ciklus, amely fix periódusban (200 ms) végzi az I/O beolvasást és a vezérlési logikát.

def plc_cycle():
    global led_state, btn_state, prev_btn_state, temperature, humidity

    while True:

        # ===== DIGITÁLIS BEMENET OLVASÁSA (FIZIKAI GOMB) =====
        # Modbus Input Status regiszter 0
        # Az ESP az aktuális gombállapotot a Discrete Input 0 címre írja.
        rr_btn = esp.read_discrete_inputs(address=0, count=1)

        if not rr_btn.isError():
            btn_state = rr_btn.bits[0]


        # ===== ÉLÉRZÉKELÉS (EDGE TRIGGER) =====
        # Ez a PLC logika:
            # ha a gomb éppen most lett lenyomva → LED állapot vált,
            # a parancs visszakerül az ESP-re
        # Ipari PLC logikában ez "RISING EDGE DETECT".
        # Csak akkor reagálunk, amikor a gomb 0→1 állapotba vált

        if btn_state and not prev_btn_state:
            led_state = not led_state          # LED állapot váltás
            esp.write_coil(0, led_state)       # Fizikai LED frissítés Modbuson

        prev_btn_state = btn_state


        # ===== ANALÓG ADATOK OLVASÁSA =====
        # Holding Register 0 → hőmérséklet
        # Holding Register 1 → páratartalom
        # Az ESP tizedes pontossággal küldi az adatot.

        rr_temp = esp.read_holding_registers(address=0, count=2)

        if not rr_temp.isError():
            temperature = rr_temp.registers[0] / 10.0
            humidity = rr_temp.registers[1] / 10.0


        # ===== PLC CIKLUS IDŐZÍTÉS =====
        # 200 ms ciklusidő (ipari PLC mintára)
        time.sleep(0.2)
```

Webes (HMI) megjelenítés

```python
# ================= WEB OLDAL =================
# A HMI felület betöltése

@app.route("/")
def index():
    return render_template("index.html")


# ================= ÁLLAPOT API =================
# A webes felület innen kapja az aktuális rendszerállapotot

@app.route("/status")
def status():
    return jsonify({
        "led": led_state,       # LED állapot
        "btn": btn_state,       # Gomb állapot
        "temp": temperature,    # Hőmérséklet
        "hum": humidity         # Páratartalom
    })


# ================= VIRTUÁLIS GOMB (WEB HMI) =================
# Ez a fizikai gomb funkcióját utánozza a weboldalon keresztül

@app.route("/toggle_led", methods=["POST"])
def toggle_led():
    global led_state

    led_state = not led_state          # LED állapot váltás
    esp.write_coil(0, led_state)       # Parancs küldése az ESP-nek

    return "OK"



# ================= PROGRAM INDÍTÁS =================
if __name__ == "__main__":

    # PLC ciklus külön szálon fut, hogy a web szerver ne blokkolódjon
    threading.Thread(target=plc_cycle, daemon=True).start()

    # Flask webszerver indítása
    app.run(host="0.0.0.0", port=5000)

```

Szenzoradatok naplózása SQL adatbázisba

A felügyeleti rendszer továbbfejlesztésének következő lépéseként célul tűztük ki, hogy a szenzoradatok ne csupán valós időben jelenjenek meg a webes HMI felületen, hanem strukturált módon, visszakereshető formában is rögzítésre kerüljenek. Ennek megvalósítására egy SQL alapú adatbázis-integráció készült a meglévő Python alkalmazás kiegészítésével.

Ez a fejlesztés új dimenziót adott a projektnek, mivel a rendszer így már nemcsak felügyeleti, hanem adatgyűjtési funkciókat is ellát, lehetővé téve azok későbbi elemzését.

Az adatnaplózás szerepe a felügyeleti rendszerben

A szenzoradatok adatbázisba történő mentése lehetővé teszi az időbeli változások nyomon követését, a működés hosszabb távú elemzését, valamint az adatok későbbi feldolgozását. A tanulók számára ez jól szemléltette, hogy a valós rendszerekben a pillanatnyi állapot megjelenítése önmagában nem elegendő; az adatok historikus tárolása alapvető igény.

A fejlesztés során hangsúlyt kapott annak megértése, hogy az adatgyűjtés nem folyamatos vezérlési célokat szolgál, hanem döntéstámogató és elemzési alapot teremt.

Adatbázis-választás és logikai felépítés

Az oktatási környezethez illeszkedve egy egyszerű, SQL alapú adatbázis-struktúra került kialakításra, amely könnyen kezelhető, mégis jól szemlélteti a relációs adatkezelés alapelveit. Az adatbázis egyetlen, jól definiált táblában rögzíti a mért adatokat. Az adatbázishoz MariaDB (MySQL alapú) adatbázismotort használtunk.
A tárolt adatok tipikusan (**sensordata** tábla):

- rögzítés azonosítója (**dataid**, int, auto-incerement, primary key)
- időbélyeg (**time_of_measurement**, datetime, default current_timestamp)
- hőmérséklet-értékek (**temp**, double),
- páratartalom-adatok (**hum**, double),
-

Ez a struktúra lehetővé teszi az adatok időrend szerinti visszakeresését és egyszerű statisztikai feldolgozását.

A Python alkalmazás kiegészítése adatbázis-kezeléssel

A meglévő Flask alapú alkalmazás bővítése során az adatbázis-kezelés egy elkülönített logikai egységként került beépítésre. A szenzoradatok kiolvasása továbbra is a Modbus TCP kommunikáción keresztül, ciklikusan történik, azonban a frissített értékek bekerülnek az adatbázisba is.
Bár a jelenlegi implementációban az adatok beküldése folyamatos az adatbázisba, a tanulók figyelmét felhívtuk, hogy egy komplex rendszerben különböző funkciók eltérő időkritikussággal működnek. Ebből kiindulva az alkalmazás fejlesztésének következő lépcsőfoka lehet az adatok naplózásának ritkítása, hogy az csak néhány másodpercenként történjen meg.
A Python program bővítéséhez először is szükség volt a mysql.connector modul importálására.

```
# ================= MODULOK IMPORTÁLÁSA =================
from flask import Flask, render_template, jsonify
from pymodbus.client import ModbusTcpClient
import threading, time
import mysql.connector #Az SQL kapcsolatot kezelő modul importálása.
```

A következő lépésben létrehoztuk az adatbázis kapcsolat kezeléséhez és az adatok beküldéséhez szükséges objektumokat.

```
# ================= ADATBÁZIS KAPCSOLAT KONFIGURÁLÁSA =================
# Az adatbázis kapcsolatot kezelő objektum létrehozása.
db = mysql.connector.connect(
    host="localhost",
    port="3306",
    user="root",
    password="",
    database="sensordatabase"
)

#Az adstbázist kezelő cursor objektum létrehozása.
dbcursor = db.cursor()
```

Végül pedig az analóg szenzoradatok lekérdezésénél bővítettük a kódot az adatok adatbázis szerverbe küldésével:

```
85          # ===== ANALÓG ADATOK OLVASÁSA =====
86          # Holding Register 0 → hőmérséklet
87          # Holding Register 1 → páratartalom
88          # Az ESP tizedes pontossággal küldi az adatot.
89
90          rr_temp = esp.read_holding_registers(address=0, count=2)
91
92          if not rr_temp.isError():
93              temperature = rr_temp.registers[0] / 10.0
94              humidity = rr_temp.registers[1] / 10.0
95              dbcursor.execute("INSERT INTO sensordata (temp,hum) VALUES (" + temperature + ","+humidity+")")
96
97
98          # ===== PLC CIKLUS IDŐZÍTÉS =====
99          # 200 ms ciklusidő (ipari PLC mintára)
100         time.sleep(0.2)
```

Oktatási jelentőség és tanulói tapasztalatok

Az SQL adatbázis-integráció bevezetése jelentősen bővítette a projekt tanulási értékét. A tanulók nemcsak a valós idejű adatkezeléssel, hanem az adatok strukturált tárolásával és későbbi felhasználhatóságával is megismerkedtek.
A fejlesztés során fejlődtek:

- az adatkezelési és adatmodellezési alapismeretek,
- a rendszerszintű gondolkodás,
- valamint az adatbiztonság és adatkonzisztencia iránti érzékenység.

A szenzoradatok adatbázisba történő rögzítése így szervesen illeszkedett a projekt egészéhez, és tovább erősítette azt a szemléletet, hogy a modern informatikai és ipari rendszerek alapja az adatok tudatos és strukturált kezelése.

Webes felügyeleti réteg – szerepe a rendszerben

A Python alkalmazás biztosítja az adatok elérését HTTP végpontokon keresztül, míg a böngészőben futó HTML + JavaScript felület gondoskodik a megjelenítésről és a felhasználói beavatkozásról.
Ez a réteg:

- nem vezérli közvetlenül a fizikai I/O-t
- az ESP állapotának digitális árnyékát jeleníti meg
- felhasználói parancsokat továbbít a Python alkalmazás felé

Az adatút:

LED állapot blokk

Ez a gomb nem közvetlenül az ESP-t vezérli, hanem a Python felügyeleti rendszert.

```
<!-- ==================== LED KÁRTYA ==================== -->
<!-- Ez a blokk a LED állapotát mutatja és innen lehet vezérelni -->
<div class="card">
    <h2>LED állapot</h2>

    <!-- A LED vizuális visszajelzője
         A színét a JavaScript módosítja:
         zöld = világít
         piros = nem világít -->
    <div id="led" class="led"></div>

    <!-- Virtuális kapcsoló gomb
         onclick esemény → toggleLED() JS függvény fut le
         Ez HTTP POST kérést küld a Flask szervernek (/toggle_led)
         Ez a gomb nem közvetlenül az ESP-t vezérli, hanem a Python felügyeleti rendszert.-->
    <button onclick="toggleLED()">LED kapcsolás</button>
</div>
```

Gomb állapot blokk

```
<!-- ==================== FIZIKAI GOMB ÁLLAPOT ==================== -->
<!-- Ez CSAK kijelzés, nem vezérlő szerv -->
<div class="card">
    <h2>Gomb állapot</h2>

    <!-- A fizikai gomb állapotának szöveges kijelzése.
         A Flask szerver PLC ciklusa frissíti.
         "Nyomva" / "Nincs nyomva" szöveg jelenik meg -->
    <div id="btn">Nincs nyomva</div>
</div>
```

A fizikai gomb állapotának szöveges kijelzése.

Szenzor blokk

```
<!-- ==================== KÖRNYEZETI ADATOK ==================== -->
<!-- DHT11 szenzor által mért adatok -->
<div class="card">
    <h2>Környezeti adatok</h2>

    <!-- Hőmérséklet kijelzés
         Ezek a DHT szenzor értékei számára fenntartott mezők.
         A program ide írja be a /status API-ból érkező értéket. -->
    <div>🌡 Hőmérséklet: <span id="temp">--</span> °C</div>

    <!-- Páratartalom kijelzés -->
    <div>💧 Páratartalom: <span id="hum">--</span> %</div>
</div>
```

Ezek a DHT szenzor értékei számára fenntartott mezők.

JavaScript – élő frissítés

```javascript
// ====================== ÁLLAPOTFRISSÍTŐ FÜGGVÉNY ======================
// Ez a függvény 200 ms-onként lefut.
// Feladata: lekérdezni a Flask szervertől az aktuális rendszerállapotot és frissíteni a weboldalon
látható adatokat.

function update(){

  // HTTP GET kérés a szerver /status végpontjára
  // A szerver JSON formátumban küldi vissza az adatokat:
  // { led: bool, btn: bool, temp: float, hum: float }
  fetch("/status")

    // A válasz átalakítása JSON objektummá
    .then(r => r.json())

    // A kapott adatok feldolgozása
    .then(data => {

      // ===== LED VIZUÁLIS ÁLLAPOT =====
      // A LED kör színének beállítása:
      // zöld ha világít (true), sötétpiros ha nem (false)
      document.getElementById("led").style.background =
        data.led ? "limegreen" : "darkred";


      // ===== FIZIKAI GOMB ÁLLAPOT KIÍRÁS =====
      // A Flask PLC ciklus által olvasott bemenet jelenik meg
      document.getElementById("btn").innerText =
        data.btn ? "Nyomva" : "Nincs nyomva";


      // ===== HŐMÉRSÉKLET KIÍRÁS =====
      // DHT11 szenzor adata az ESP → Modbus → Flask útvonalon
      document.getElementById("temp").innerText = data.temp;


      // ===== PÁRATARTALOM KIÍRÁS =====
      document.getElementById("hum").innerText = data.hum;
    });
}
```

A JavaScript ezeket a változókat kapja vissza, frissíti.

LED kapcsoló gomb működése

```javascript
// ====================== VIRTUÁLIS LED KAPCSOLÓ ======================
// Ez a függvény akkor fut le, amikor a felhasználó megnyomja a weboldalon a "LED kapcsolás" gombot.

function toggleLED(){

  // HTTP POST kérés küldése a szervernek
  // A Flask oldalon ez a /toggle_led útvonalat hívja meg, ami megfordítja a LED állapotát és kiírja
  Modbus-on az ESP-re.
  fetch("/toggle_led", { method: "POST" });

  // Fontos: itt nincs válaszfeldolgozás, a következő update() ciklus már lekéri az új állapotot.
}
```

Folyamat:

Frissítési ciklus

```javascript
// ====================== IDŐZÍTETT FRISSÍTÉS ======================
// A böngésző 200 ms-onként automatikusan meghívja az update() függvényt.
// Ez kvázi PLC ciklus a weboldalon (HMI polling).
// Ez megfelel a Python programban lévő ciklus ütemének.

setInterval(update, 200);
```

Ez megfelel a Python ciklus ütemének, így a rendszer szinkron marad.

A stílus szerepe a rendszerben

A CSS biztosítja, hogy a HMI webes felület:

- jól olvasható legyen gyenge fényviszonyok között is

- egyértelmű vizuális visszajelzést adjon az állapotokról
- elkülönítse az információs blokkokat (HMI panelek logikája)

Ez a struktúra SCADA / ipari HMI (Human–Machine Interface)ek minimalista szemléletét követi.

Teljes CSS – részletesen kommentelve

```css
/* ========================= OLDAL ALAPSTÍLUS ========================= */
body {
    font-family: Arial;
    background: linear-gradient(135deg,#0f2027,#203a43,#2c5364);
    color: white;
    text-align: center;
    margin: 0;
}

/* ========================= KÁRTYA (PANEL) STÍLUS =========================
   Minden információs blokk ezt használja:
   LED, Gomb, Szenzor adatok*/
.card {
    background: #222;
    margin: 20px auto;
    padding: 20px;
    width: 320px;
    border-radius: 12px;
    box-shadow: 0 0 10px #000;
}

/* ========================= LED VISSZAJELZŐ KÖR =========================
   Ez a vizuális LED indikátor*/
.led {
    width: 40px;
    height: 40px;
    margin: 15px auto;
    border-radius: 50%;
    background: darkred; /* Alapértelmezett állapot: LED kikapcsolva. A program dinamikusan felülírja:
    limegreen = bekapcsolva */
}


/* ========================= GOMB STÍLUS =========================
   Virtuális LED kapcsoló*/
button {
    padding: 10px 15px;
    border-radius: 8px;
    border: none;
    background: #00aaff;
    color: white;
    cursor: pointer;
}
```

Ez megfelel a valódi PLC HMI, SCADA terminál és ipari kezelőpanel megjelenítési elveinek.
Miért fontos ez oktatásban?
A tanulók nem csak adatot jelenítenek meg hanem megtanulják, hogy egy HMI (Human–Machine Interface):

- funkcionális vizuális rendszer
- ahol a szín, méret, elrendezés jelentéssel bír

A webes réteg ezzel teljes:

A webes HMI felület tanulói továbbfejlesztése csapatmunkában

A felügyeleti rendszer alapfunkcióinak megvalósítását követően a projekt következő eleme a webes HMI (Human–Machine Interface) felület továbbfejlesztése volt. Ebben a szakaszban a tanulók már nem előre meghatározott megoldásokat valósítottak meg, hanem saját elképzeléseik mentén, csapatmunkában alakították tovább a meglévő HTML és CSS alapú felületet.

A feladat célja az volt, hogy a tanulók megtapasztalják, miként lehet egy működő, de alapvető megjelenésű felületet felhasználóbarátabbá, áttekinthetőbbé és vizuálisan is informatívabbá tenni, miközben megmarad a rendszer funkcionális stabilitása.

Csapatmunka és feladatmegosztás

A tanulók kisebb munkacsoportokban dolgoztak, amelyekben a feladatok megosztása tudatosan történt. Egyes tanulók elsősorban a struktúra átalakítására (HTML elemek rendezése, logikai csoportosítás), míg mások a megjelenés finomítására (színek, elrendezés, vizuális visszajelzések) koncentráltak.
Ez a munkaszervezési forma lehetővé tette, hogy a tanulók megtapasztalják a közös fejlesztés előnyeit és kihívásait, valamint gyakorolják az egyeztetést és az egymás munkájára épülő fejlesztést.

A felület funkcionális és vizuális továbbfejlesztése

A kiindulási alapként szolgáló webes felület már tartalmazta a rendszer alapállapotainak megjelenítését és az alapvető vezérlési lehetőségeket. A tanulók ezt a felületet továbbfejlesztve:

- átrendezték az információk megjelenítését,
- bővítették a megjelenített szenzoradatok tárházát, előkészítve egy későbbi továbbfejlesztést,
- egyértelműbb vizuális visszajelzéseket alakítottak ki,
- valamint a megjelenési stílust csapatuk igényei szerint módosították.

A módosítások során kiemelt figyelmet kapott az áttekinthetőség és a használhatóság, különösen annak érdekében, hogy a felület valós felügyeleti környezetben is könnyen értelmezhető legyen.

Tanulók továbbfejlesztett munkája 1. (Forrás: Saját szerkesztés)

Egyéni igények és kreatív megoldások megjelenése

A fejlesztési folyamat során a tanulók saját elképzeléseik szerint alakíthatták a felület megjelenését és működését. Ennek eredményeként különböző megoldások születtek, például:

- eltérő színkódolás az állapotjelzésekhez,
- alternatív elrendezések a szenzoradatok megjelenítésére,
- hangsúlyosabb visszajelzések a beavatkozások során.

Ez a szabadság lehetőséget adott a kreativitás kibontakoztatására, miközben a tanulók megtanulták, hogy az egyéni ötleteket össze kell hangolni a rendszer műszaki és funkcionális korlátaival.

70. ábra: Tanulók továbbfejlesztett munkája 2. (Forrás: Saját szerkesztés)

Oktatási jelentőség és fejlesztési tapasztalatok

A webes HMI felület továbbfejlesztése során a tanulók komplex módon alkalmazták korábban megszerzett ismereteiket. A feladat egyszerre fejlesztette:

- a digitális és informatikai kompetenciákat,
- az együttműködési és kommunikációs készségeket,
- valamint a felhasználóközpontú gondolkodást.

A folyamat jól illeszkedett a projekt célkitűzéseihez, mivel a tanulók nem csupán technikai megoldásokat hoztak létre, hanem egy működő rendszer gondolkodtak a felületről, figyelembe véve annak szerepét a teljes felügyeleti architektúrában.

Összegzés – rendszerintegrációs és tanulási eredmények

Ebben a fejezetben bemutatott tevékenységek a projekt egyik legkomplexebb és pedagógiailag legértékesebb szakaszát alkották. Ebben a fázisban a tanulók egy teljes, működő felügyeleti rendszert valósítottak meg, amelyben a hardveres elemek, a hálózati kommunikáció, a szoftveres vezérlési logika és a webes megjelenítés egységes rendszerként működött.

A Raspberry Pi alapú futtatókörnyezet előkészítése és a Flask alapú alkalmazás üzembe helyezése lehetőséget adott arra, hogy a tanulók valós informatikai és ipari jellegű

környezetben alkalmazzák programozási és rendszerüzemeltetési ismereteiket. A Modbus TCP kommunikáció használata különösen fontos tapasztalatot jelentett, mivel a tanulók egy iparban elterjedt protokollal dolgozhattak, és megértették annak szerepét egy felügyeleti architektúrában.
A webes HMI felület továbbfejlesztése során a tanulók nem csupán technikai megoldásokat készítettek, hanem felhasználóközpontú szemléletet is elsajátítottak. A csapatmunkában végzett fejlesztések során megjelent az egyéni kreativitás, az önálló döntéshozatal, valamint a közös szakmai egyeztetés gyakorlata. A felületek testreszabása hozzájárult ahhoz, hogy a tanulók sajátjuknak érezzék a rendszert, és felelősséget vállaljanak annak működéséért.
Pedagógiai szempontból kiemelendő, hogy a tanulók a teljes fejlesztési folyamat során folyamatos visszacsatolást kaptak döntéseik következményeiről. A rendszer működésének tesztelése, a hibák feltárása és javítása, valamint a megoldások közös értékelése erősítette az önreflexiót és a problémamegoldó gondolkodást. A hibák kezelése nem kudarcként, hanem a tanulási folyamat természetes részeként jelent meg.
A 6. fejezetben megvalósított tevékenységek hozzájárultak több kulcskompetencia fejlődéséhez, különösen:

- a digitális kompetencia,
- az együttműködési és kommunikációs készségek,
- az algoritmikus és rendszerszintű gondolkodás,
- valamint az önálló tanulás és felelősségvállalás területén.
-

Összességében ez a fejezet jól példázza, hogy a projektalapú, gyakorlatorientált megközelítés miként képes összekapcsolni az elméleti ismereteket a valós alkalmazási környezettel. A tanulók nem csupán egy működő technikai rendszert hoztak létre, hanem olyan tapasztalatokat szereztek, amelyek hosszú távon is megalapozzák szakmai fejlődésüket és pályaorientációjukat.