Széchenyi István Egyetem
Gépészmérnöki, Informatikai és
Villamosmérnöki Kar

SZAKDOLGOZAT

Rábai Balázs
Villamosmérnök BSc.

2024

SZÉCHENYI EGYETEM
UNIVERSITY OF GYŐR
GÉPÉSZMÉRNÖKI, INFORMATIKAI ÉS VILLAMOSMÉRNÖKI KAR

Szakdolgozat

Okostermosztát rendszer tervezése és kivitelezése a felhasználói kényelem és energiahatékonyság javítására

Rábai Balázs

Villamosmérnöki BSc szak
Automatizálási szakirány

2024

Feladat-kiíró lap szakdolgozathoz

Nyilatkozat

Alulírott, Rábai Balázs (BK2W0W), Villamosmérnök, BSc. szakos hallgató kijelentem, hogy az *Okostermosztát rendszer tervezése és kivitelezése a felhasználói kényelem és energiahatékonyság javítására* című szakdolgozat feladat kidolgozása a saját munkám, abban csak a megjelölt forrásokat, és a megjelölt mértékben használtam fel, az idézés szabályainak megfelelően, a hivatkozások pontos megjelölésével.

Eredményeim saját munkán, számításokon, kutatáson, valós méréseken alapulnak, és a legjobb tudásom szerint hitelesek.

Győr, 2024.11.29.

Rábai Balázs

Kivonat

Okostermosztát rendszer tervezése és kivitelezése a felhasználói kényelem és energiahatékonyság javítására

A dolgozat célja egy innovatív okostermosztát rendszer fejlesztése, amely a felhasználói kényelem növelése és az energiahatékonyság javítása révén hozzájárul a fenntarthatósághoz. A rendszer két fő egységből áll: az *espTouch*-ból és az *espCarryable*-ből. Az *espTouch* egy helyhez kötött központi modul, amely hőmérséklet- és páratartalom-méréseket végez, valamint érintőkijelzőt biztosít a felhasználói beállítások kezelésére. Az *espCarryable* egy hordozható, kvázi valós idejű adatgyűjtésre alkalmas egység, amely szoros együttműködésben dolgozik az *espTouch*-csal.

A rendszer tervezése során a modularitás alapelvként érvényesült. Az egyes hardveres interfészek, kommunikációs protokollok és felhasználói felületek külön modulokként kerültek kialakításra, elősegítve a bővíthetőséget és a meglévő rendszerekkel való kompatibilitást. A kommunikációt WebSocket protokoll valósítja meg, amely gyors és biztonságos adatcserét tesz lehetővé, míg a fűtési rendszerek vezérlését az RS-485 busz és a Modbus RTU protokoll biztosítja.

A rendszer részletes bemutatása magában foglalja a kijelzők és szenzorok működését, valamint az alkalmazott protokollokat, például a One-Wire, SPI, I2C és WebSocket megoldásokat. Az adaptív szabályozásnak köszönhetően az *espTouch* automatikusan választ a hőszivattyús és a gázkazán-alapú fűtési rendszerek között a külső hőmérsékleti körülmények és felhasználói preferenciák figyelembevételével.

A szoftver tervezésekor kiemelt figyelmet kaptak az adatok integrációjára és a felhasználói interfész intuitív megvalósítására irányuló szempontok. A rendszer képes a fűtéskörönkénti hőmérséklet-szabályozásra, valamint a PLC-alapú fűtés rendszer vezérlővel való kommunikációra, miközben egyszerűsíti a működést és minimalizálja a hibalehetőségeket.

Ez az okostermosztát rendszer jelentős előrelépést képvisel a hőmérséklet-szabályozás hatékonysága és élménye terén, növelve a felhasználói elégedettséget, továbbá hozzájárulva az energiahatékonyság javításához.

Abstract

Developing and implementation of a smart thermostat system to improve user comfort and energy efficiency

The aim of this thesis is to develop an innovative smart thermostat system that contributes to sustainability by increasing user comfort and improving energy efficiency. The system consists of two main units: *espTouch* and *espCarryable*. The *espTouch* is a stationary central module that takes temperature and humidity measurements and provides a touch screen display to manage user settings. The *espCarryable* is a portable, real-time data collection unit that works in close cooperation with the *espTouch*.

The system has been designed with modularity as a guiding principle. The hardware interfaces, communication protocols and user interfaces have been designed as separate modules, facilitating extensibility and compatibility with existing systems. Communication is implemented by WebSocket protocol, which allows fast and secure data exchange between the *espCarryable* and *espTouch*, while control of the heating systems is provided by RS-485 bus and Modbus RTU protocol, following industry standards.

A detailed description of the system will include the operation of the displays and sensors, as well as the protocols used, such as One-Wire, SPI, I2C and WebSocket. Thanks to its adaptive control, *espTouch* automatically selects between heat pump and gas boiler-based heating systems based on external temperature conditions and user preferences.

The software was designed with data integration and user interface implementation in mind. The system is capable of per-loop temperature control and communication with the PLC-based heating system controller, while simplifying operation and minimizing the potential for error.

This smart thermostat system represents a significant advance in the efficiency and experience of temperature control, increasing user satisfaction and contributing to improved energy efficiency.

Tartalomjegyzék

1. Bevezetés

1.1. Lakossági felhasználás

A modern, digitális fali termosztátok számos előnyt kínálnak a lakossági és kereskedelmi felhasználóknak. Először is, ezek az eszközök jelentős mértékben hozzájárulnak a kényelem növeléséhez azáltal, hogy lehetővé teszik az épületek hőmérsékletének pontos szabályozását. Az optimális hőmérséklet fenntartása nemcsak az általános komfortérzetet javítja, hanem fontos szerepet játszik az egészség megőrzésében is.

Egy másik jelentős előny az energiahatékonyság. A digitális termosztátok lehetővé teszik a fűtési és hűtési rendszerek pontos időzítését, vezérlését, és csökkenthetik az energiafogyasztást. Ez nemcsak a rezsiköltségek mérsékléséhez járul hozzá, hanem a környezeti terhelést is csökkenti azáltal, hogy kevesebb energiát használnak fel. Így csökken a szén-dioxid-kibocsátás is.

Az ilyen eszközök programozhatósága további előnyt jelent. A felhasználók előre beállíthatják a kívánt hőmérsékletet a különböző napszakokra, például alváshoz, munkába induláshoz vagy hazatéréshez. Ez nemcsak kényelmes, hanem lehetőséget ad arra is, hogy energiafogyasztásukat optimalizálják. Csak akkor fűtenek vagy hűtenek, amikor arra ténylegesen szükség van.

Mindazonáltal a digitális fali termosztátok használata bizonyos kihívásokat is jelenthet. Sok modern modell számos fejlett funkcióval rendelkezik, amelyek használata a technikailag kevésbé jártas felhasználók számára bonyolult lehet. A megfelelő beállítások elvégzése gyakran alapos tanulmányozást igényel, vagy szakember segítségét teszi szükségessé.

Ezen kívül, a digitális fali termosztátok elhelyezése is kritikus fontosságú. Az eszközöknek olyan helyiségekben kell lenniük, ahol a vezérelt fűtéskör működik, hogy pontosan mérhessék a hőmérsékletet, és hatékonyan szabályozhassák azt. Az ideális elhelyezés hiánya csökkentheti a rendszer hatékonyságát, és pontosságát.

Összefoglalva, a digitális fali termosztátok számos előnnyel rendelkeznek, amelyek javítják a kényelmet, növelik az energiahatékonyságot, és csökkentik a környezeti terhelést. Ugyanakkor a felhasználók számára fontos, hogy megfelelően megismerjék ezeknek az eszközöknek a használatát, beállítását, hogy maximálisan kihasználhassák azok előnyeit, miközben minimalizálják a használatukkal járó kihívásokat.

1.2. Ipari felhasználás

A termosztátokat széles körben alkalmazzák az ipar különböző területein, ahol a hőmérséklet precíz szabályozása elengedhetetlen. Ilyen területek például a gyártási folyamatok, ahol a termékek minőségének biztosítása érdekében elengedhetetlen, hogy a hőmérséklet egy meghatározott szinten maradjon. A megfelelő hőmérsékleti körülmények fenntartása nemcsak a gyártás során, hanem a termékek tárolása alatt is kiemelten fontos.

Az ipari berendezések és gépek hőmérsékletének ellenőrzése szintén kritikus jelentőségű. Ezek az eszközök csak egy adott hőmérsékleti tartományban működnek optimálisan, és ezen tartomány túllépése jelentős károkat okozhat. A hőmérséklet megfelelő szabályozása és karbantartása hozzájárul a gépek élettartamának meghosszabbításához. Így jelentős költségmegtakarítást eredményezhet a vállalat számára.

A gyárakban nemcsak a gyártósorok és gépek hőmérsékletének szabályozása fontos, hanem az irodaépületekben dolgozó munkavállalók komfortérzetének biztosítása is. Az ipari környezetben gyakran központi fűtési rendszereket alkalmaznak, amelyeket automatizált rendszerek irányítanak, így a falra szerelt termosztátok használata nem szükséges. Az ilyen rendszerek lehetővé teszik az épület egészének hőmérsékleti szabályozását anélkül, hogy a munkavállalóknak manuálisan kellene beállítaniuk a hőmérsékletet.

Az ipari hőmérséklet-figyelés teljesen automatizált, és előre megtervezett rendszereken alapul. Ezek a rendszerek folyamatosan monitorozzák és szabályozzák a hőmérsékletet, biztosítva a megfelelő körülményeket a gyártási folyamatokhoz, és a munkavállalók számára egyaránt. A fejlett automatizáció révén nincs szükség falra szerelt termosztátokra, mivel a központi rendszerek biztosítják a hőmérséklet állandó és precíz szabályozását.

Összességében a termosztátok, és hőmérséklet-szabályozó rendszerek kulcsfontosságú szerepet játszanak az iparban, hozzájárulva a gyártási folyamatok hatékonyságához, a gépek élettartamának meghosszabbításához, valamint a munkavállalók kényelméhez és biztonságához. Az automatizált hőmérséklet-szabályozás révén az ipari vállalatok jelentős költségmegtakarítást érhetnek el, miközben fenntartják a magas szintű működési hatékonyságot.

1.3. Probléma leírása

Az otthonokban, (és az iparban) alkalmazható termosztátok döntő többségben egy adott fűtéskört tudnak kapcsolni, és nem rendelkeznek olyan okos funkciókkal, amelyek segítenék a fejlesztési lehetőségeket.

Általában a program beállíthatósága bonyolult, illetve az egyes paraméterek akár több gomb kombináción keresztül érhetőek el, kezdetleges karakteres kijelzővel rendelkeznek, amelyek nehezen olvashatóak, emiatt működésük nehezen érthető, beállításuk nehézkes. Ezzel szemben egy érintő kijelzős egység magában hordozza azt a lehetőséget, hogy a fejlesztők által implementált interaktív elemeket/rendszer paramétereket a felhasználó könnyen megváltoztathassa.

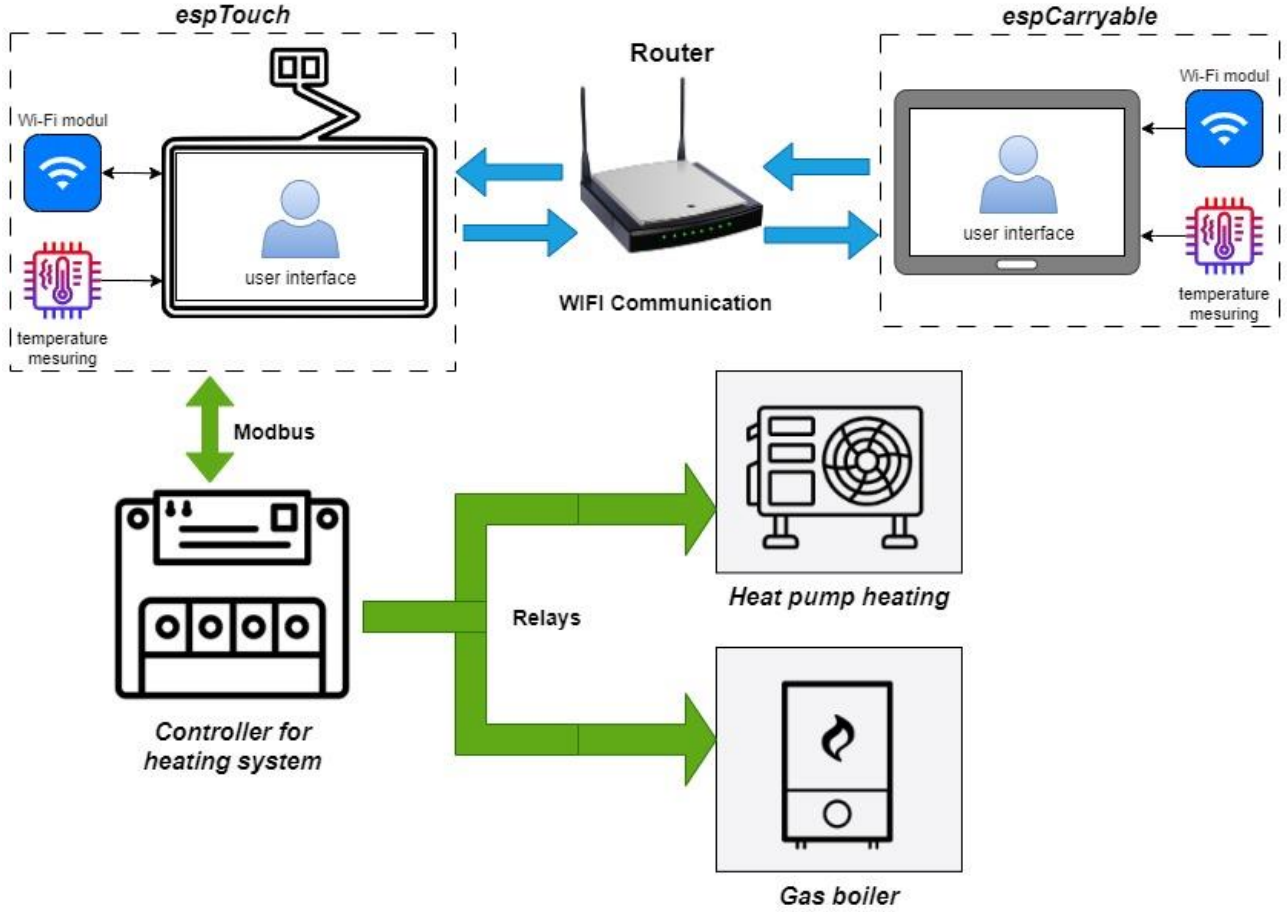
A nem innovált termosztátok hátránya az is, hogy egy fűtéskörre a lakótérben egy termosztátot építenek ki, ami általában a teljes ház hőmérsékletéért felel. A fixen telepített eszközök magukban hordozzák azt a hátrányt, hogy fűtés korszerűsítés esetén komoly kompromisszum megkötésére kényszerülünk.

A fentiek tekintetében, ezen hibák kiküszöbölésére szeretnék egy megoldási javaslatot tenni, hogy hogyan lehetne egy innovatív és univerzális rendszert létrehozni.

1.4. Innovatív megoldási javaslat

A termosztátok kulcsfontosságú szerepet töltenek be az otthonok és munkahelyek kényelmének, energiahatékonyságának és költséghatékonyságának biztosításában. A felhasználók olyan termosztátokat keresnek, amelyek könnyen kezelhetők, megbízhatók és programozhatók, hogy pontosan szabályozhassák a hőmérsékletet és optimalizálhassák az energiafogyasztást.

A megoldási javaslatomban egy olyan termosztát rendszert tervezek, amely több különböző eszközt integrál, így a hőmérséklet szabályozása teljesen autonóm módon történik.



1.ábra. Folyamatábra a tervezett rendszerről

Ennek megvalósításához és demonstrálásához két eszközt tervezek, amelyek WIFI-n keresztül kommunikálnak egymással. A két eszköz az „*espTouch*” és „*espCarryable*” elnevezést kapta, amiknek hardveres és szoftveres felépítését részletesen ismertetem a következő fejezetekben.

Az *espTouch* egy helyhez kötött egység, ami hőmérsékletet és páratartalmamat mér, valamint egy érintő kijelzős interfésszel rendelkezik, amin keresztül be lehet állítani a felhasználói preferenciákat. Ezeket az igényeket az eszköz számos funkcióval elégíti ki, beleértve a felhasználó által beállítható fűtési programokat. Ennek célja a fűtési költségek minimalizálása, és

hatékonyságának növelése, valamint választható a fűtés típusa (*Heat pump heating, Gas Boiler*) is, ha több opció is van a fűtés megvalósítására. Mindemellett a rendszerben ez a periféria képviseli a *szervert*, azaz kommunikálni fog az *espCarryable*- lel a mérési adatokról és egyéb felhasználó igényekről, és a fűtésvezérlővel (*Controller for heating system*), hogy milyen energia alapú fűtést indítson el. A kommunikáció a rendszereszközök között egy belső hálózaton fog lezajlani, míg a fűtésvezérlést (*Controller for heating system*) *RS-485* buszon keresztül lesz elérhető a *szervernek*, amely *Modbus* protokollt használ.

Az *espCarryable* egy hordozható eszköz melynek a feladata az, hogy folyamatosan mérje a környezeti hőmérsékletet, illetve a hozzá tartozó páratartalmat. Erre az eszközre nyomógombokat implementáltam, melyekhez univerzális, és interaktív funkciókat csatoltam, amelyekkel az *espTouch*-nak egyes adatait lehet befolyásolni.

Az *1. ábrán* található (*Controller for heating system*) lesz a felelős azért, hogy a rendszerben definiált kritériumok teljesüljenek attól függetlenül, hogy az milyen működési elven alapszik. Jelen esetemben, egy hőszivattyún, vagy gázkazánon keresztül lehet fűteni az épületet, amelynek vezérlése a külső hőmérséklet függvényében eldönti, hogy villamos vagy gáz alapú energiát használjon fel a szobák felfűtésére. Valamint az egyes alrendszerek áramellátásához relés modulok vannak használatban.

A rendszer lényege, hogy több *espCarryable* legyen üzemelve egy háztartásban, amik szobákként vannak elhelyezve. Ennek célja az, hogy a fűtés működéséért felelős berendezés, azaz az *espTouch*, informálva legyen arról, hogy mely szobában szükséges még fűteni, illetve hol sikerült elérni a felhasználó által beállított hőmérsékletet. Mindezeken túl szeretném úgy megtervezni a szóban forgó rendszert, hogy a már meglévő fűtésvezérlő rendszerekbe is implementálható legyen.

Ez az intelligens termosztát rendszer nemcsak, hogy növeli a kényelmet és az energiatakarékosságot, de egy új szintre emeli a hőmérséklet-szabályozás élményét is.

2. Hardver Elemek

2.1. *espTouch*

A következőkben be fogom mutatni az *espTouch* fizikai felépítését, amik alátámasztják az *1.4. Innovatív megoldási javaslat fejezetben* leírtakat. Kifejtem továbbá az eszköz megvalósításához alkalmazott mikrokontrollert, illetve az ezekhez tartozó egyéb perifériákat.

Az elkészült kapcsolási rajzokat az *I.* és *II. mellékeltben* vizualizáltam.

2.1.1. ESP WIFI modul

A mikrovezérlő programozható, kis méretű integrált áramkörökből épülnek fel, amit kifejezetten speciális feladatok végrehajtására terveztek. Egy ilyen modul, mint például az ESP32 rendkívül sokoldalú és különböző alkalmazási területeken használható, ahol szükség van az alacsony költségű és energiahatékony megoldásokra.

Az ESP32 egy olyan fejlesztői mikrovezérlő sorozat, amelyet az Espressif Systems hozott létre. Ebből a sorozatból választottam az ESP32-WROOM-32, amely az ESP32-D0WDQ6 chip-pel van felszerelve, ami 32 bites Tensilica Xtensa LX6 processzor architektúrával rendelkezik.

A Tensilica Xtensa LX6 kialakítása végett az ESP32-s chip-ek két processzormaggal rendelkeznek, amik magonként akár 240 MHz-s működési frekvenciát is elérhetnek. A két mag lehetővé teszi, hogy a feladatokat párhuzamosan és hatékonyan hajtsa végre. Mindezek mellett kifejezetten pozitív hatást kelt, hogy modul alacsony energiafelvétellel rendelkezik, ami különösen fontos az IoT és hordozható eszközök esetében.

A modul emellett gyorsítótárakkal is el van látva, azaz *cache*-sel, ami 520 KB belső SRAM-t (Static Random Access Memory) tartalmaz, illetve a mikrokontroller 480 KB ROM-val (Read-Only Memory) is rendelkezik.

Ezeken kívül 4 MB flash memóriát integráltak a chip-be, ami egy olyan memóriaterület, hogy tápellátás nélkül is megőrzi a memóriában tárolt adatokat, amiket tetszés szerint újra lehet írni. Ez által flash memória fogja a későbbiekben tárolni a futtatni kívánt programot, valamint egyéb fontos adatokat, mivel a periféria nem rendelkezik beépített EEPROM-mal.

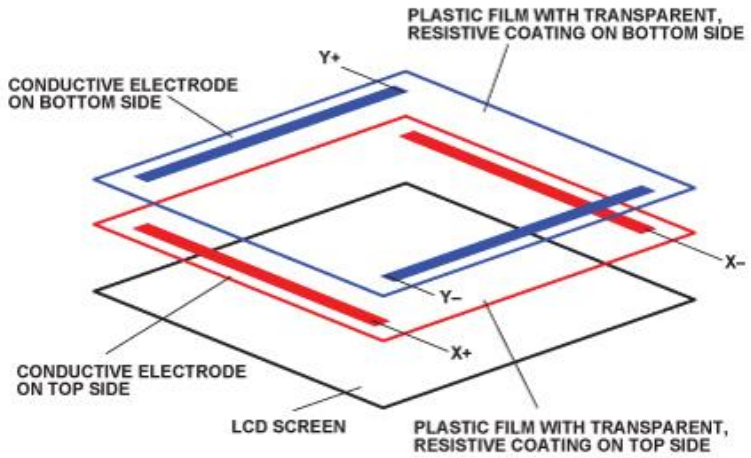
Mindezek mellett az ESP32 mikrokontroller, azaz *MCU* kifejezetten választékos perifériákkal rendelkezik még, amit az *1.táblázatban* lehet megtekinteni.

1. táblázat. ESP32 Hardware specifikáció (forrás: [A1])

Categories	Items	Specifications
Hardware	Module interfaces	SD card, UART, SPI, SDIO, I^2C, LED PWM, Motor PWM, I^2S, IR, pulse counter, GPIO, capacitive touch sensor, ADC, DAC, Two-Wire Automotive Interface (TWAI®, compatible with ISO11898-1)
	On-chip sensor	Hall sensor
	Integrated crystal	40 MHz crystal
	Integrated SPI flash	4 MB
	Operating voltage/Power supply	3.0 V ~ 3.6 V
	Operating current	Average: 80 mA
	Minimum current delivered by power supply	500 mA
	Recommended operating temperature range	–40 °C ~ +85 °C
	Package size	(18.00±0.10) mm × (25.50±0.10) mm × (3.10±0.10) mm
	Moisture sensitivity level (MSL)	Level 3

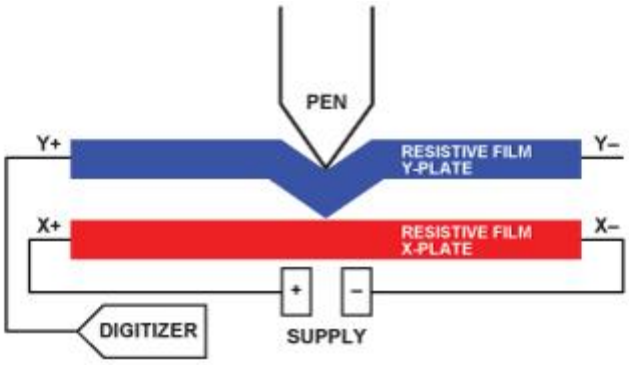
2.1.2. Rezisztívinterfész

A felhasználói interakcióhoz egy LCD kijelzőt használok, aminek az átmérője 7.112 cm, és 240x320 darab pixelből áll, ezen felül színes képet képes produkálni. Az eszközt SPI buszra lehet csatlakoztatni, amivel a felhasznált *MCU* is rendelkezik, valamint rezisztív érintő felülettel van ellátva.



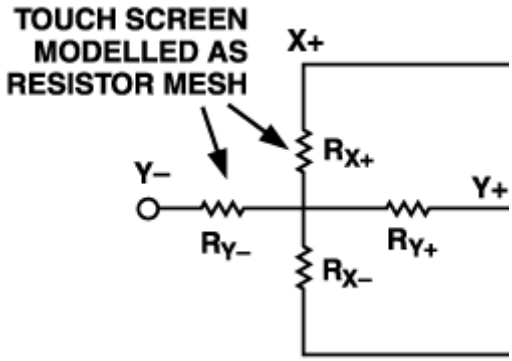
2.ábra. Érintő szenzor kialakítása a kijelző felett (forrás: [A2])

A kijelző két vékony rétegből áll össze, ezen rétegek alkotnak egy koordináta egységet. Ahogy a *2.ábrán* is megtekinthető, a két réteget egymásra merőlegesen kell elhelyezni, mivel a felület lenyomásával fog létrejönni egy koordináta rendszerbeli pont. Ezt úgy kell értelmezni, hogy két azonos rétegbeli elektróda között végtelen sok az elektródákra merőleges egyenes jön létre.



3.ábra. Érintő szenzor működése (forrás: [A2])

A két réteg egyenesei is merőlegesek egymásra, amiből azt a következtetést lehet levonni, hogy egy egyenes a koordináta rendszer egyetlen pontját tartalmazza. Ebből következik, hogy ha egy tetszőleges ponton megnyomjuk a szenzort, ahogy *3.ábrán* is látható, akkor két egymásra merőleges egyenest fogunk egy pontban egymáshoz nyomni, azaz két egyenesben tárolt adat fogja ismertetni a lenyomás helyének a pontját, ahol az egyenesek összeérnek.



4.ábra. Érintő szenzor ellenállás osztása (forrás: [A3])

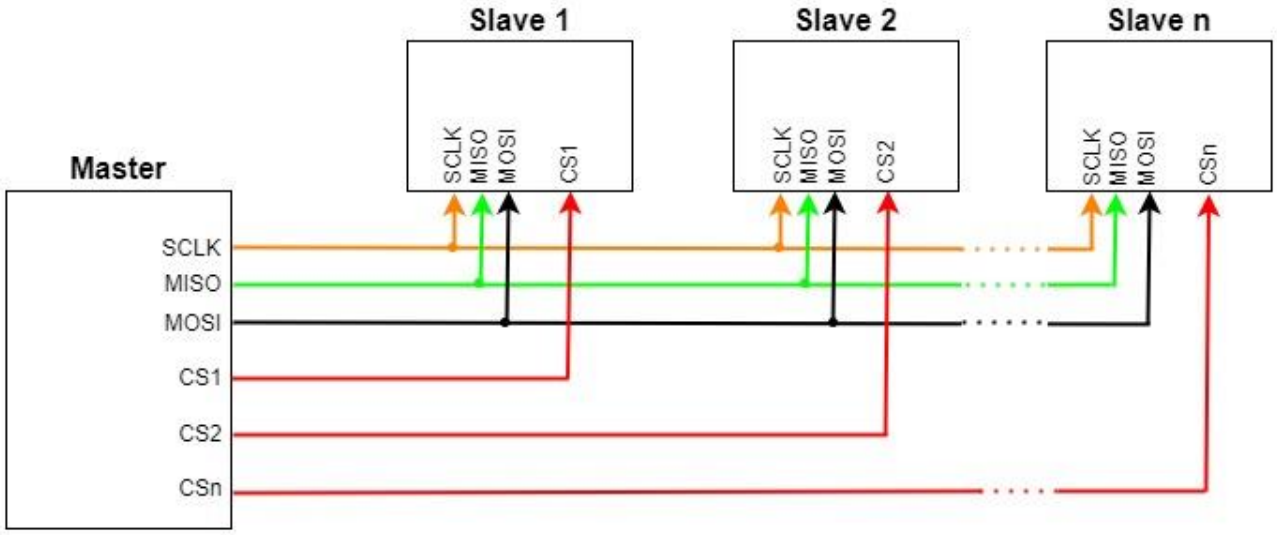
Az említett működési elv fizikailag úgy épül fel, hogy a két régből csak az egyik kap tápellátást, míg a másik rétegen feszültséget mérünk. Tegyük fel, hogy megnyomjuk a kijelzőt egyetlen pontban úgy, hogy csak két egyenes fog érintkezni. A két egyenes ilyenkor két-két szakaszra bomlik fel, és ezen szakaszokat ellenállással lehet helyettesíteni, amiknek aránya függ az érintési pont elhelyezkedésétől. Ezek az arányok úgy alakulnak ki, hogy egy rétegben az elektródák távolságait a lenyomott ponttól elosztjuk egymással. Gyakorlatilag a *2.ábrán* az *Y* réteg az érintett felületen létrehozza a feszültségosztót, amit a *4. ábrán* lehet szemlélni. A *2.ábrán* az *X* réteg kap tápellátást, így az érintett pontban az *X* rétegbeli ellenállások között egy potenciál fog megjelenni, ami az *Y* réteg ellenállásai között is meg fog jelenni. Az ellenállásokat nem ismerjük, csak a mért potenciált, ami közös mindkét rétegben, és a távolságokat az azonos rétegbeli elektródák között. Ezáltal visszaszámolható a mérés alapján az ellenállás nagysága az érintési pont és az elektródák között. Ebből vissza lehet következtetni a szakaszok távolságára is, amik az előbb említett pontok között

húzódnak, valamint egy referencia tengelyen az érzékelt pont koordinátáját fogják meghatározni.

Az interfész működésének megértéséhez a *[15] forrást* használtam fel.

2.1.3. SPI busz fizikai felépítése

Az SPI (Serial Peripheral Interface) busz fizikai felépítése egyszerű, ami lehetővé teszi, hogy több eszköz is csatlakoztatható legyen egy közös buszra. Ez a kialakítás egy szinkron soros kommunikációs protokollt használ, amelyen keresztül a *Master* eszköz (általában egy mikrovezérlő) kommunikálhat a buszra csatlakoztatott *Slave* eszközökkel. A *Slave*-ek különböző perifériák, mint például szenzorok, kijelzők, memória modulok, de akár egy másik mikrokontroller is lehet.



5.ábra. SPI busz általános implementálása

Az SPI busz alapvetően négy vezetéket használ, ahogy azt az *5.ábrán* is láthatjuk és ezek sorra az *SCLK, MOSI, MISO, CS*. A következő ismeretek az *[1] forrásból* származnak.

- *SCLK*: Azaz órajel, amit a *Master* szolgáltat, hogy szinkronizálja a *Slave*-t az adatátvitelhez.
- *MOSI: Master* kimenet, *Slave* bemenet. Ez a vonal a mester eszközről küldött adatokat továbbítja a periféria azaz *Slave* eszköz felé.
- *MISO*: *Master* bemenet, *Slave* kimenet. Ezen a lábon keresztül érkezik az adat a perifériától az *MCU* felé, azaz a *Master* felé.
- *CS: Chip Select, Master* ezen lábon keresztül választja ki azt a modult, amivel kommunikálni szeretne.

A buszhálózaton a *MISO, MOSI, SCLK* lábak a buszon közössé válnak, így csak a *CS* láb szüksége a *Masternek*, hogy melyik *Slave*-vel akar kommunikálni, ahogy az *5. ábrán* látható.

2. táblázat. SPI busz adatátviteli módok az órajel függvényében (forrás: [A4])

SPI Mode	CPOL	CPHA	Clock Polarity in Idle State	Clock Phase Used to Sample and/or Shift the Data
0	0	0	Logic low	Data sampled on rising edge and shifted out on the falling edge
1	0	1	Logic low	Data sampled on the falling edge and shifted out on the rising edge
2	1	0	Logic high	Data sampled on the rising edge and shifted out on the falling edge
3	1	1	Logic high	Data sampled on the falling edge and shifted out on the rising edge

Ezeken túl az SPI buszt négy különböző módban lehetséges használni, amint a *2. táblázatban* is lehet szemlélni.

SPI-ben a *Master* választhatja ki az óra polaritását és az óra fázisát. A CPOL bit állítja be az órajel polaritását az üresjárati állapotban. Az üresjárati állapot az, amikor a *CS* magas, és az átvitel kezdetén alacsonyra vált, vagy amikor a *CS* alacsony, és az átvitel végén magasra vált. A CPHA bit választja ki az órajelfázist. A CPHA bit függvényében az órajel felfutó vagy lefutó élét használják az adatok mintavételezésére, és/vagy eltolására. A *Masternek* kell kiválasztania az óra polaritását és az óra fázisát, az alcsomópont követelményeinek megfelelően. A CPOL és CPHA bit kiválasztásától függően négy SPI üzemmód áll rendelkezésre. Az ismeret az *[1] forrásból* származik.

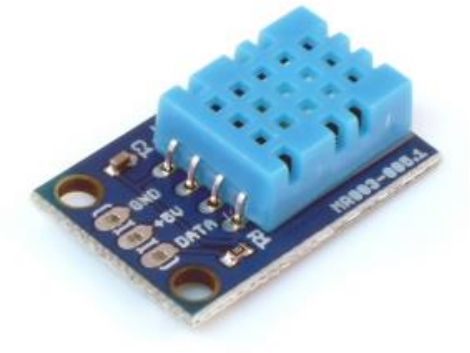
A busz előnyei közé tartozik továbbá, hogy teljes duplex kommunikációra képes, ami azt jelenti, hogy az adatok egyidejűleg mindkét irányba továbbíthatók. Hátrányai, hogy több vezetéket igényel, mint néhány más soros buszhálózat. Továbbá a hozzá tartozó kommunikációs protokoll nem rendelkezik beépített hibajavítással vagy címzési mechanizmussal.

2.1.4. Hő- és páratartalom- kombinált érzékelő modul

A DHT11-s szenzorba egy SI7021 chip van beépítve, amely egy fejlett digitális hőmérséklet- és páratartalom-érzékelő. A kombinált érzékelő megbízható és pontos méréseket biztosít, valamint az adatokat digitális formában továbbítja a csatlakoztatott *MCU*-nak.

Az SI7021 chip hőmérséklet mérésére egy NTC (Negative Temperature Coefficient) hőérzékelőt használ, ezen felül a páratartalom méréséhez egy kapacitív érzékelővel van ellátva. A chip-et gyártás során kalibrálták, ami biztosítja, hogy az érzékelő a teljes működési élettartama alatt pontos méréseket végezzen. A hőmérsékletmérés pontossága általában ±0.5°C, illetve mért értékét 0.1 °C tudja változtatni. Míg a páratartalom mérése ±2% pontossággal történik.

Ezek az adatokat a *[2] forrásból* származnak, amely a DHT11 adatlapjára hivatkozik.



6.ábra. DHT11 (forrás: [A5])

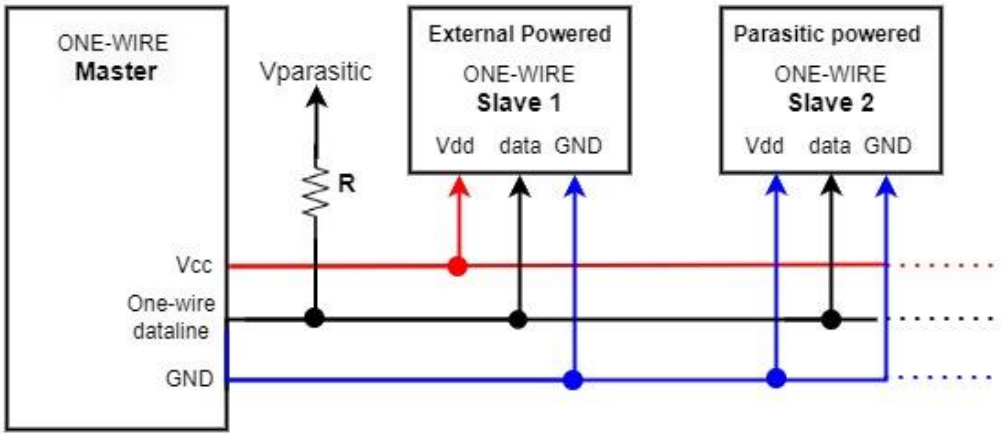
Az érzékelő a mikrokontrollerrel való kommunikációjához One-Wire buszt használ, amit a következőben fogok ábrázolni.

2.1.5. One-Wire busz

A One-Wire busz egy soros aszinkron kommunikációs busz, ami egyetlen adatvezeték és egy közös földvezeték használatán alapul. Ezáltal két vezeték biztosítja a kapcsolatot az adatátvitelhez, és bizonyos esetekben az energiaellátást is biztosíthatja. A One-Wire busz különösen alkalmas egyszerű és költséghatékony rendszerek kialakítására.

Minden eszköz egyedi 64 bites sorozatszámmal rendelkezik, amely lehetővé teszi több eszköz együttes működését egyetlen buszon. Számos eszköz képes a szükséges energiát buszvezetékből kinyerni, amihez az integrált kondenzátorát használja. Ezt nevezzük parazita táplálásnak, aminek segítségével tovább egyszerűsíti a rendszer kialakítását és csökkenti az energiaellátásra vonatkozó követelményeket. Az egyszerű felépítés és az alacsonyabb vezetékszám felhasználása miatt gazdaságos megoldást nyújt különféle alkalmazásokhoz. Továbbá, ideális a közeli érzékelési és monitorozási alkalmazásokhoz, például a hőmérséklet- vagy nyomásérzékelők esetében, mivel egyszerű és hatékony adatgyűjtési lehetőséget biztosít.

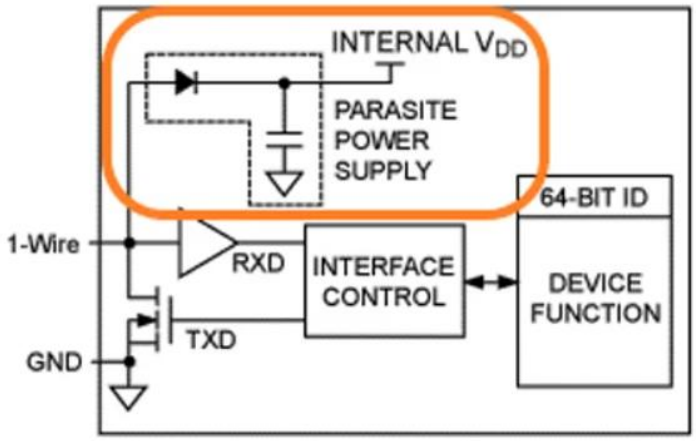
A kommunikáció során a *Master* eszköz (általában egy mikrokontroller) küld egy start jelet, amit az összes csatlakoztatott eszköz érzékel a buszon. Az adatátvitel bitenként történik, ahol a buszvezeték logikai állapota (magas vagy alacsony) határozza meg a továbbított bit értékét. Az eszközök válaszüzenetei szintén a busz logikai szintjének változtatásával történnek, biztosítva a pontos és hatékony adatcserét a rendszerben.



7.ábra. One-Wire busz

A *7.ábrán* látható, hogy az adatvonalat a busz felhúzó ellenállása magasra (*Vparasitic*) húzza, ekkor a *Slave 2* nevezetű egység belső diódája nyitó irányba előfeszül, és feltölti a belső kondenzátorát. Ebben az esetben *Slave 1*-gyel csak is akkor lehetséges a kommunikáció, ha megszűnik a kommunikáció időintervallumára a magas potenciál a *Vparasitic* lábon.

A *Slave 1* egy direkt tápellátású eszköz a One-Wire buszrendszeren, amely saját, dedikált tápellátást kap egy külön tápfeszültségen (*Vcc*) keresztül. Könnyen integrálható olyan rendszerekbe, ahol többféle *Slave* együttesen működik, mivel nem igényli a parazita tápellátás speciális körülményeit, valamint sokkal stabilabb működésűek a direkt táplált *Slave* eszközök.



8.ábra. One-Wire *Slave* belső általános felépítése (forrás: [A6])

Az *8. ábrán* bemutatott One-Wire *Slave* működésének szemléltetésére szolgáló diagram alapján megállapítható, hogy a *Slave* eszköz belső kondenzátorában tárolt töltés biztosítja az energiaellátást a vezérlő áramkör működtetéséhez, amikor a busz alacsony feszültségű állapotba kerül. Ebben az állapotban a dióda záró irányban van előfeszítve, ami megakadályozza az energia áramlását a

kondenzátorból. A busz aktív állapotában van, amikor a *Master* eszköz adatkommunikációt folytat a Slave eszközzel. Az alacsony feszültségű állapot alatt elvesztett töltés később visszanyerésre kerül, amikor a One-Wire adatvonal feszültsége ismét átlépi a 2,8 V-os potenciálú küszöbértéket, ekkor a dióda nyitó irányban van előfeszítve. Ezt a mechanizmust, amely lehetővé teszi az adatvonalon keresztül történő energianyerést és felhasználást, parazita tápellátásnak nevezzük. Utóbbi bekezdések az *[5] forrásból* származnak, amely a One-Wire buszról értekezik.

2.2. *espCarryable*

A következőkben be fogom mutatni az *espCarryable* felépítését, amik biztosítani fogják az *1.4. Innovatív megoldási javaslat fejezetben* leírtakat.

Az *espCarryable* egy hordozható eszköz, hasonló az *espTouch*-hoz, mivel ugyanolyan *MCU-val* van ellátva, és funkcióiban is sok közös van.

Alapvetően ez az eszköz egy OLED kijelzővel rendelkezik, amellyel I^2C buszon keresztül lehet kommunikálni, valamint három nyomógombbal, amivel a felhasználó minimális változtatásait hajthatja végre. Ezek a változtatások kvázi valós időben láthatóvá válnak a kijelzőn, és az *espTouch*-on. Ezen felül lehetőség van arra is, mint az *espTouch*-nál, hogy soros porton keresztül lehessen a belső változóit paraméterezni a megfelelő működés érdekében.

2.2.1. OLED kijelző

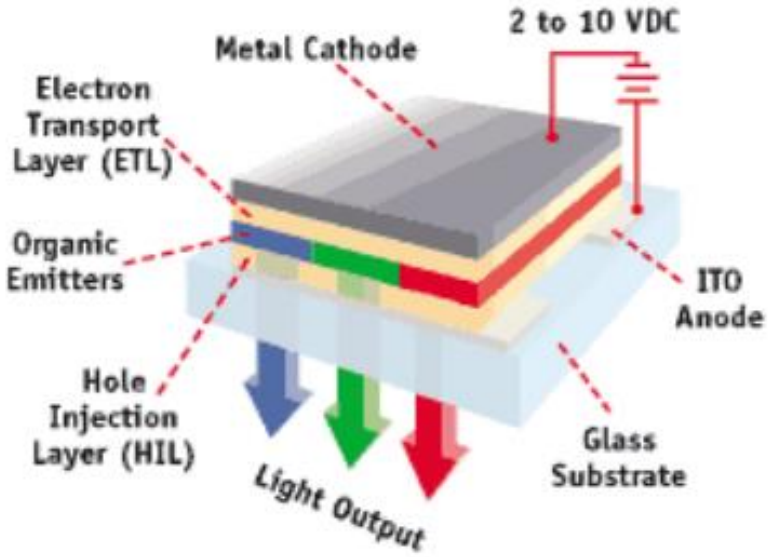
A szerves fénykibocsátó diódák (OLED), mint újfajta kijelző technológia világszerte nagy figyelmet kapnak. Az OLED-ek számos előnnyel rendelkeznek a hagyományos kijelző technológiákkal szemben. Az OLED-k elektrolumineszcencián alapuló energiaátalakító eszközök (elektromosság-fény). Az elektrolumineszcencia olyan szilárd anyag fénykibocsátása, amelyen elektromos áramot vezetnek keresztül.

LCD-vel szemben több előnye is van, amelyek a következők:

- Az OLED-et különböző szögekből lehet nézni anélkül, hogy színvesztést érzékelne a néző.
- Nincs szükségük háttérvilágításra.
- Meghajtófeszültség és az energiafogyasztás alacsony.

Az OLED egy szilárdtest- vagy elektronikai eszköz, amely egy szerves vékonyrétegből áll, ami jellemzően két vékonyrétegű vezető elektróda közé van elhelyezve. Az OLED gyártásához egy tervezett szénalapú molekulát használnak, amely fényt bocsát ki, amikor elektromos áram halad át rajta, amit „electro phosphorescence”-nek neveznek, amely a *[3] forrásból* lett idézve. A kijelzőknek

nagyon kevés tápfeszültségre van szükségük, azaz mindössze 2-10V, valamint a fekete szín megjelenítéséhez sincs szükség működési feszültségre, ami az OLED sajátossága. Ezáltal energiahatékonyabb, és képmegjelenítési képessége is jobb lesz, mint az LCD kijelzőé, ami a működéséhez állandó háttérfényt igényel. Ezek mellett az OLED szerves anyaga segíti feljavítani a fényerő és a fény színének szabályozását.



9.ábra. OLED technológia strukturális felépítése (forrás: [A7])

A *9.ábra* bemutatja az OLED technológia strukturális architektúráját, és működési elvét, amiket a következő bekezdésekben fogok leírni.

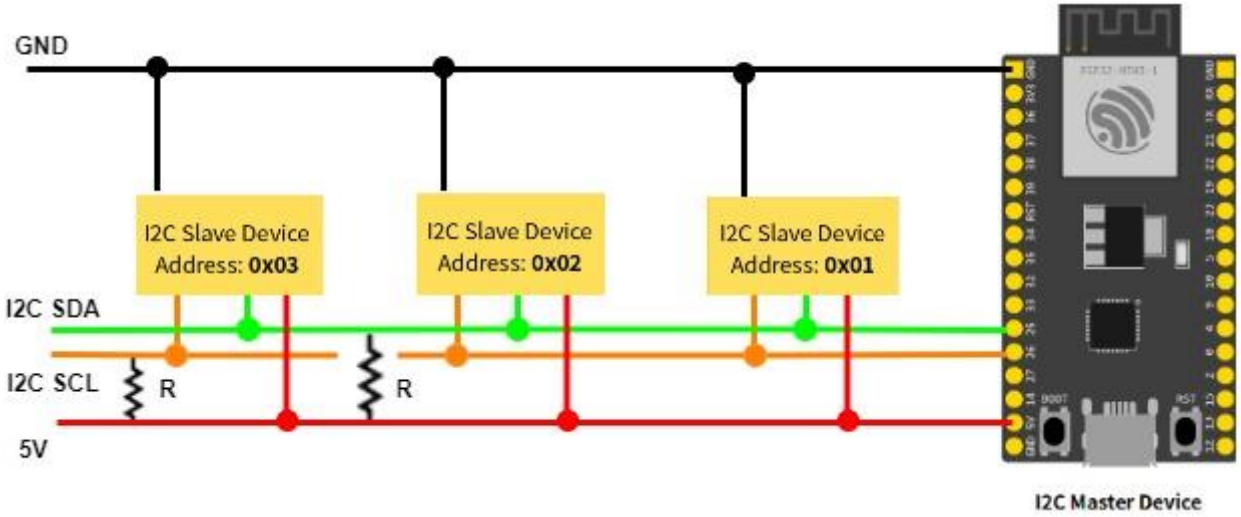
Az aktív rétegek olyan minimális vastagságúak (~10Å és 100 nm közöttiek), hogy az egyes rétegekben lévő elektromos mezők monumentálisak, 105-107 V/cm nagyságrendűek. Ezek a magas, közel a töréspontho tartozó elektromos mezők támogatják a töltések befecskendezését az elektróda/aktív rétegek határfelületein keresztül. A lyukakat az anódról, amely jellemzően átlátszó, az elektronokat pedig a katódról injektálják. Az injektált töltések ellentétes irányban vándorolnak egymás ellen, végül találkoznak és újra egyesülnek. A rekombinációs energia felszabadul, és a molekula vagy a polimer szegmens, amelyben a rekombináció megtörténik, kilépett állapotba kerül. A feszültségek molekuláról molekulára vándorolhatnak végül egyes molekulák vagy polimer szegmensek az energiát fotonok vagy hő formájában szabadítják fel. Fontos, hogy az összes felesleges gerjesztési energia fotonként (fényként) szabaduljon fel.

A töltések rekombinációs helyekre való eljuttatására használt anyagok általában (de nem mindig) gyenge fotonsugárzók (a gerjesztési energia nagy része hő formájában szabadul fel). Ezért megfelelő adalékanyagokat adnak hozzá, amelyek először átadják az eredeti gerjesztésből származó energiát, és az energiát hatékonyabban szabadítják fel fotonokként.

Ezek az ismeretek a *[3] forrásból* származnak, ami az OLED technológiát mutatja be.

2.2.2. I²C busz architektúrája

Az I²C, azaz Inter-Integrated Circuit, egy olyan szinkron soros busz, amely lehetővé teszi a *Master* és a *Slave* közötti adatátvitelt egy rendszeren belül. Ez a kapcsolat szabadon bővíthető, több *Masterrel* és *Slave*-el is rendelkezhet. A busz tápfeszültsége általában 3.3-5V, ami kompatibilis a legtöbb mikrovezérlő kimeneti feszültségével.



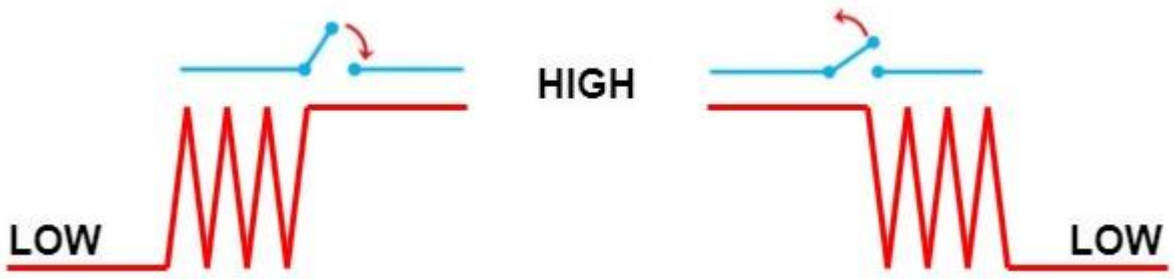
10.ábra. I²C busz minta kialakítása

Felépítése a *10.ábrán* látható, amely nagyon hasonló a One-wire busz kialakításához, mivel egy adat vezetéken (SDA: Serial Data Line) történik a kommunikáció az eszközök között. A különbség az, hogy a *Master* és *Slave* egymáshoz képest szinkronban van a kommunikáció során. Ezt úgy lehet kivitelezni, hogy az adatláb mellett az eszközökön még egy csatlakozási pont van, ez pedig a SCL (Serial Clock Line). Ezáltal a *Master* órajelet generál a *Slave*-nek, hogy a kommunikáció során ne történjen szinkronvesztés, ezáltal adatvesztés.

Az állításaim a *[4] forrásban* található cikkre támaszkodnak, ami az I²C busz alapjairól szól.

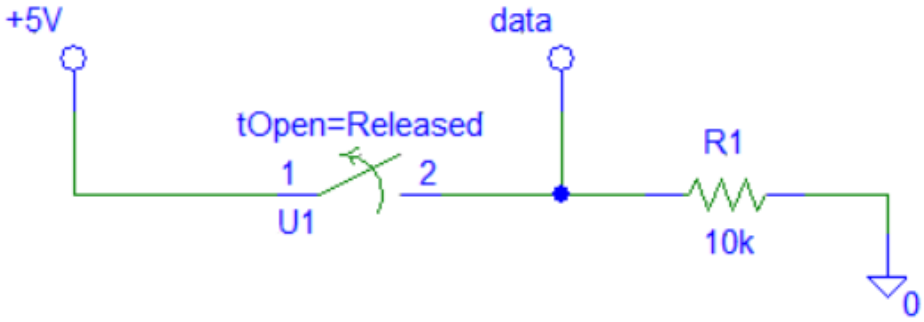
2.2.3. További perifériák

A kapcsoló elemek jelentősége sem elhanyagolható, hiszen ezek fizikai érintkezői nagymértékben befolyásolják az eszköz működését. Ha rosszul vannak bekötve a nyomógombok, akkor a felhasználói interakció hatására az eszköz tönkremenetelét okozhatják. Az *espCarryable* eszközben három darab nyomógombot használok, amik kulcsfontosságúak. A kapcsolók lenyomásakor vagy elengedésekor sokkal több érintkezés történik, mint amennyit a felhasználó szándékozott megtenni, ami általában egy.



11.ábra. Kapcsoló működés

A *11. ábrán* látható, hogyan működik egy kapcsoló, és mire kell gondosan odafigyelni, amikor digitális áramkörben használjuk. Ezeknek az eszközöknek képesnek kell lenniük egy rövid lenyomást 1-es értéknek érzékelni, megfelelő mintavételezési idővel. Ez azonban hibát hordoz magában, mivel előfordulhat, hogy egy-egy lenyomást nem érzékel az eszköz, ha az nem pontosan a mintavételi időben történik. Az én megoldásom erre az első érzékelés utáni késleltetés. Ha az elején tüskeszerű impulzus jelentkezik, a rendszer azt egy lenyomásnak érzékeli, majd késlelteti a program futását addig, amíg a bemeneti jel konstanssá nem válik. Ezzel biztosítom, hogy a felhasználói beavatkozások pontosak és megbízhatóak legyenek.



12.ábra. Kapcsolási rajz a kapcsolók mikrovezérlőhöz illesztéséről

Amint a *data* láb bemenete magasra vált a *12.ábrán*, a rendszer azonnal késlelteti a működését, éppen annyira, hogy a felhasználó ne vegye észre, amíg el nem engedi a kapcsolót.

2.3. Controller for heating systems

A termosztát rendszer úgy van megtervezve, hogy bármilyen fűtést szabályozó berendezést, illetve rendszert képes legyen vezérelni. Azaz, attól függetlenül, hogy milyen technológiával működik, valamint, hogy a szabályozott rendszer realizálva lett vagy sem, a vezérlő *espTouch* alkalmazható lehessen.

Jelen pillanatban egy olyan rendszerhez lett igazítva a kimenete, ami már implementálva lett.

Ez egy hibrid rendszer, ami több levegős hőszivattyúval, és egy gázkazánnal van felszerelve. Ezek a berendezések egymástól függetlenül is el tudják látni a feladataikat, ami a külső hőmérséklettől függ. A döntési folyamatokat egy központi digitális áramkör dönti el.

A rendszer a berendezéseket relés modulokkal kapcsolja a hálózati tápellátásukhoz, attól függően, hogy éppen melyik fűtésmodul kerül használatba/kiválasztásra.

Mivel több opció is elérhető a fűtés típusa szerint, így szükséges egy olyan termosztát egység, ami képes kommunikálni a fűtést szabályzó rendszerrel. Ennek az az oka, hogy a hőszivattyú képes -5 °C működni, de fűtési hatékonysága az alacsony hőmérséklet miatt csökken egy bizonyos alsó küszöb hőmérséklettől, amit egy fűtőszállal lehet valamelyest kompenzálni. Ezekkel ellentétben egy gázkazán a külső hőmérséklettől függetlenül tud működni, ami lehetőséget ad arra, hogy alacsony hőmérsékleten valamivel energiahatékonyabban lehessen fűteni. Ennek előnye, hogy kevesebb idő alatt lehet felfűteni egy épületet. Így a választható opciók létrehozása egy rendszernél kulcsfontosságú, valamint teljes mértékben növeli a felhasználói kényelmet.

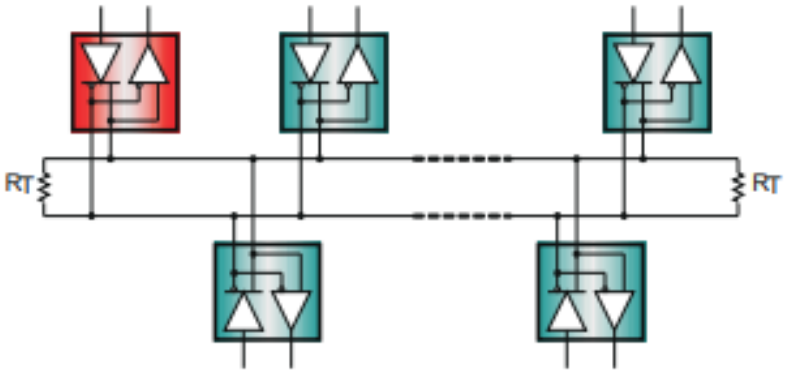
2.3.1. RS-485 hálózat

Univerzális eszközt szeretnék létrehozni, amely a most kiépített rendszernél használható, és a későbbiekben máshol is alkalmazható/adaptálható. A fűtéskapcsolást modbusz kommunikációval fogom megvalósítani az *espTouch* és a *Controller for heating system* között, amelyhez RS-485-s buszhálózatot kell kialakítani. Ez fogja biztosítani az univerzitást.

1983-ban az Electronics Industries Association (EIA) egy új kiegyensúlyozott szabványt hagyott jóvá az *RS-485*-t, amely ipari, orvosi- és fogyasztói alkalmazásokban széles körben elfogadott.

Főbb jellemzői az alábbiak:

- Differenciális jelküldés.
- Legfeljebb 32 *Slave* csatlakoztatható a hálózatra.
- 12m hosszú csavart érpáron maximum 10Mbps sebesség érhető el.
- Maximum 1200m hossz lehet a busz két végpontja között.

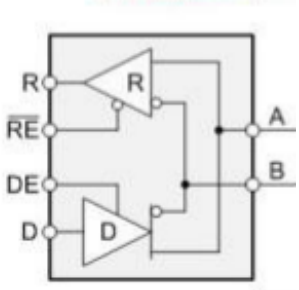


13.ábra. *RS-485* busz Half-duplex struktúrája (forrás: [A9])

Half-duplex felépítésű hálózat esetén, amit a *13. ábrán* látható, csak egy csavart érpárt alkalmazunk, így az adatok küldése különböző időpontokban történhet meg. Mivel egy vezetékpárral lehet realizálni a buszt, így annak lezárásáról gondoskodni kell, hogy adatküldés közben ne történjen, vagy minimalizált legyen a reflexió, azaz a jelvisszaverődés. A hálózat két végére R_T lezáró ellenállásokat kell elhelyezni, amelynek nagysága függ az alkalmazott kábel hosszától, és annak impedancia karakterisztikájától, valamint a kommunikáció sebességétől (baud rate) is.

$$\boldsymbol{\Gamma} = \frac{z_T - z_0}{z_T + z_0} \tag{1}$$

Az *RS-485*-s hálózathoz ajánlott szabványos UTP kábel érpárját használni, mivel annak hullámimpedanciája (Z_0) 120Ω. Az ilyen típusú vezetékpárt szintén 120Ω nagyságú lezáró ellenállással (Z_T) kell lezárni, hogy a reflexiós tényezője nulla legyen, azaz ne következzen be jelvisszaverődés. Egy vezetékpárnak a reflexiós tényezőjét (r) az (1) számú képlet segítségével lehet kiszámolni.

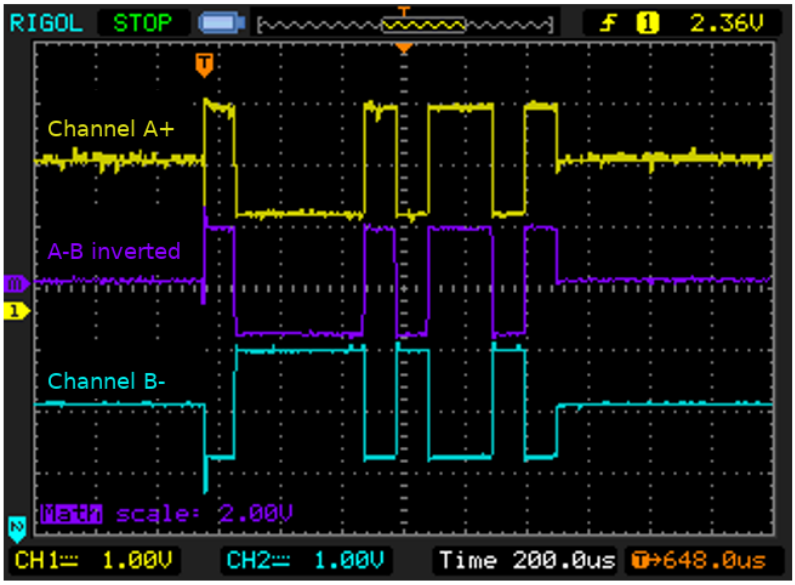


14.ábra. *RS-485* transceiver (forrás: [A10])

A busz konfigurációja *Master*/*Slave* architektúrájú és általában egy *Master* van a hálózatra kapcsolva, és a kommunikációt mindig a *Master* kezdeményezheti. Továbbá minden eszközt a buszra egy transceiver-n keresztül lehet a hálózatra felcsatlakoztatni, hogy a buszkommunikáció egységes legyen. Az *14.ábrán* egy transceiver blokkábrája látható, és csatlakozási pontjai a következőt jelentik:

- **R**: Adatok fogadása
- $\overline{\textbf{RE}}$: Vevő engedélyezése (magas: inaktív, alacsony: aktív)
- **DE**: Adó engedélyezése (magas: aktív, alacsony: inaktív)
- **D**: Adatok küldése
- **A** és **B**: *RS-485*-s buszt alkotó érpár

Működés során mindig csak egy eszköz kommunikálhat, ilyenkor a *DE* és a $\overline{RE}$ magas logikai állapotban vannak. Ezt követően a *D* lábon keresztül lehet a buszra adatfolyamot indítani. Üzenet fogadásához az engedélyező lábakat alacsony állapotba kell kapcsolni és az *R* lábon keresztül lehet feldolgozni a bejövő adatokat.



15.ábra. *RS-485* buszon küldött adatfolyam oszcilloszkóp felvétele (forrás: [A11])

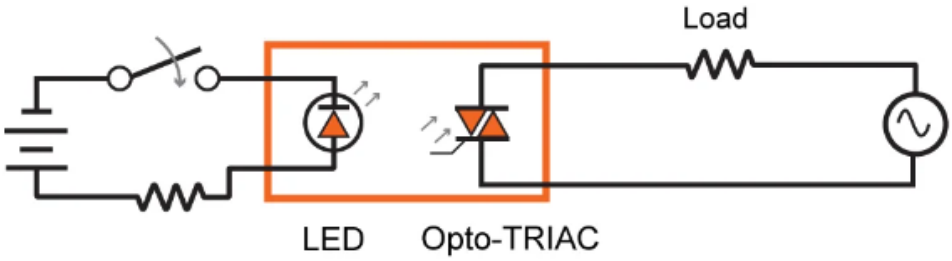
A *15. ábrán* látható a differenciális jelküldés, amely az *RS-485* kommunikációs szabvány egyik fontos tulajdonságára utal, ami azt jelenti, hogy az adatátvitelhez használt jeleket két vezeték között küldik, amelyek ellentétes polaritásúak. A rendszerben az adatok két vezetéken keresztül kerülnek továbbításra, legyenek ezek *A* és *B*. Az egyik vezeték (*A)* a jel pozitív verzióját hordozza, míg a másik vezeték (*B*) az ellentétes, negatív verzióját, és a vevő a két jel közötti feszültségkülönbséget méri.

Előnyös az ilyen típusú jelküldés, mivel a külső elektromágneses zajokra csak minimálisan lesz érzékeny a hálózat. A csavart érpár csökkenti a külső zajok befolyását az adatfolyamra, valamint a két vezetékre ugyan úgy hatnak ezek a külső zajok. Ebből következik, hogy a két jel elenyészően fog torzulni a zaj hatására, ezzel védve a buszon lévő eszközöket, illetve a jelek különbsége továbbra is meg fog egyezni a jelfeldolgozó eszköznél.

Az eddig leírtakhoz a *[6]*, *[7], [14] források* tartalmát használtam fel, hogy ismertessem az *RS-485* busz működését.

2.3.2. Relés modulok

Olyan relét választottam, amelyet a bemenetén keresztül kisfeszültséggel lehet vezérelni, hogy a kimeneti oldalán hálózati feszültséget legyen képes kapcsolni. Ez a relé fajta nem más, mint a szilárdtest relé, röviden *SSR*. Ezeket a reléket arra fogom használni, hogy a *Controller for heating system* a *Gas boiler*-t és/vagy a *Heat pump heating*-t tápellátását kapcsolja.



16.ábra. SSR Relé általános felépítés (forrás: [A12])

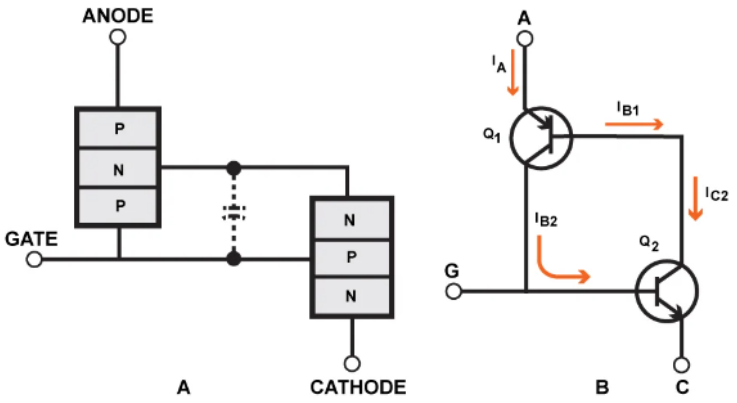
A *16. ábrán* általános kialakítását láthatjuk a SSR-nek, ami képes egyenáram (DC) és váltakozó áram (AC) kapcsolására, valamint némely típus képes mindkettőre. A hagyományos relével szemben a kapcsoló eleme nem mechanikus, és többféle is lehet, mint pedálul a bipoláris vagy MOS tranzisztor, SCR (Tirisztor), TRIAC, vagy ezek kombinációja.

Erről a technológiáról szóló ismeretek a *[8] forrásban* is megtalálhatóak.

17.ábra. *Crydom LR Series LR600240D25R* (forrás: [A13])

Az általam választott SSR relé *a Crydom LR Series LR600240D25R*-s modellje, amely 4-32V (DC) vezérelhető, és 24-280V (AC) képes kapcsolni és maximum 25A-t. A kapcsoló eleme az SCR, amely nagy kapcsolási kapacitást és hosszú élettartamot biztosít. Az adatokat az eszközzel kapcsolatban a *[9] forrásban* lehet megtalálni.

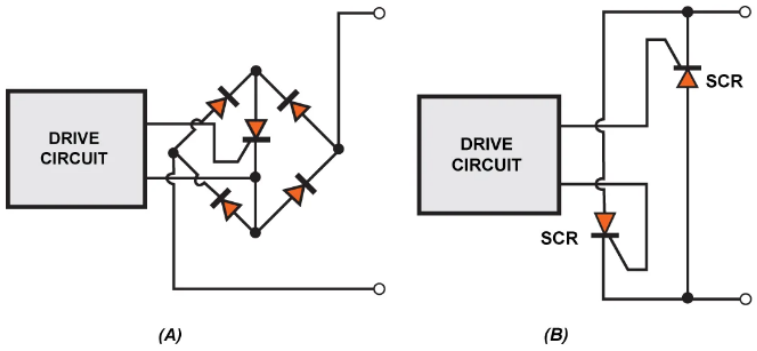
Az SCR kapcsoló azaz Tirisztor négy rétegű félvezető elem, amely struktúrát a PNPN felépítés jellemez. Három lába van, ezek sorra az anód, katód, és a vezérlő láb (gate).



18.ábra. PNPN struktúra megértése (forrás: [A12])

A modul működési elvének megértéséhez szükséges lesz az egymást követő négy réteget több ismert elemre bontani. Tegyük fel, hogy két bipoláris tranzisztorral lehet helyettesíteni a négy réteget, amely a *18. ábrán* látható. Mivel *Q1* PNP kialakítású tranzisztor és *Q2* NPN kialakítású, így ábrán tisztán láthatóvá válik, hogy négy rétegről van szó, mivel mindkét tranzisztor kollektora egyben a másik tranzisztor bázisa is.

Bekapcsoláskor a *Gate*-re I_{B2} nagyságú áramot kell kapcsolni, hogy *Q2* tranzisztor kapcsoló üzembe kerüljön, és teljesen kinyisson. Ebből következik, hogy I_{C2} áram fog *Q2*-n átfolyni, ami egyben az I_{B1} is. I_{C2} és I_{B1} nagysága és iránya is megegyezik, mivel *Q1* tranzisztor PNP felépítésű. Amely azt jelenti, hogy a bázisból kifolyó áram hatására fog kinyitni *Q1*. Ha a bekapcsolási folyamat megtörtént, akkor *Gate* vezérlése nélkül is képes működni a kapcsolás. Ebből következik az, hogy egy SCR elem, akkor fog kikapcsolni, ha az anód és katód közötti áram nullára csökken. Mivel váltakozó áramú hálózatot fog kapcsolni az eszköz, így minden pozitív félperiódus végén az eszköz kikapcsol, és újabb *Gate* vezérlésre van szükség.



19.ábra. SCR alapú SSR teljes periódus kapcsolására (forrás: [A12])

Az említett kapcsoló modul csak a szinusz hullám pozitív félperiódusban tud működni, így ahhoz, hogy a mindkét fél periódusban működhessen, szükséges kiegészíteni az áramkört a következőképpen, amit a *19. ábrán* láthatunk. Azaz egy SCR-t vagy dióda hídba kapcsolunk, hogy a

váltakozó áram teljes periódusát vezesse, vagy egymással párhuzamosan és ellentétesen. Utóbbiból az következik, hogy mindkét SCR csak az egyik félperiódust vezérli, így mikor megtörténik a nulla átmenet, és az egyik kikapcsol, akkor a másik SCR-re vezérlő *Gate* áramot adva bekapcsoljuk azt.

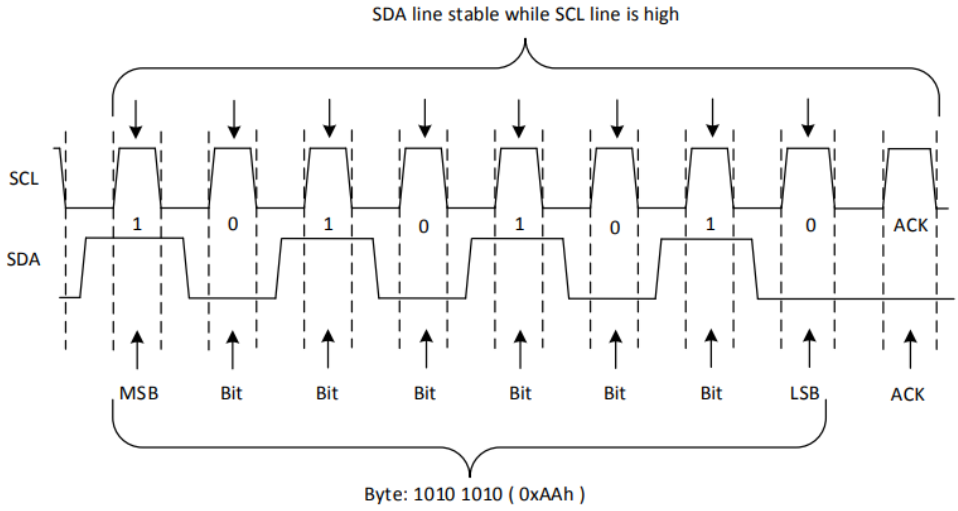
3. Kommunikációs szoftver elemek

A működőképes rendszer felépítéséhez elengedhetetlen az az ismeret, amit a Hardver elemek fejezetben leírtam, továbbá szükséges ezen elemekkel, modulokkal kommunikációt létesítenie a mikroprocesszornak. Ezek a kommunikációs szabályok alapján épülnek fel, amelyeket egy szóval protokolloknak neveznek. Ezen protokollokkal adatokat képes kinyerni a mikrokontroller a szenzorokból, vagy adatot, adathalmazt tud küldeni az adott modulnak, amely végeredménye a felhasználó számára is láthatóvá válik, például egy kijelző képének frissítése.

3.1. Eszköz belső protokolljai

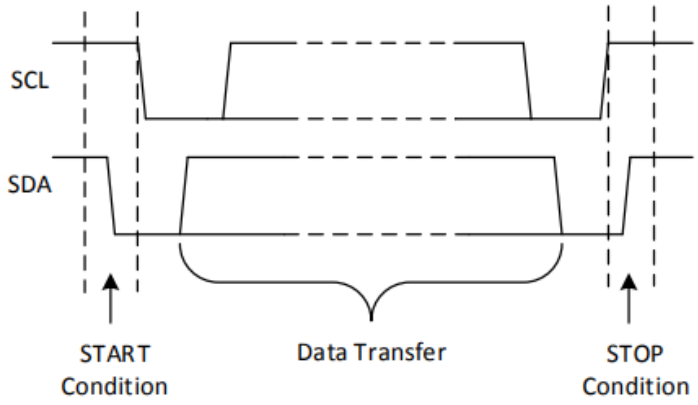
3.1.1. I²C protokoll

Az I²C, azaz Inter-Integrated Circuit hardveres felépítése a *2.2.2. I²C architektúrája* című alfejezetben már ismertetve van, így ebben a fejezetben az ott említetteket is fel fogom használni.



20.ábra. Példa az I²C adatcsomagra (forrás: [A14])

A szabvány 7-16 biten történő kommunikációt biztosít, valamint adatátviteli sebessége általában 10 kbit/s-tól egészen 3.4Mbit/s-ig terjed. Ellentétes logikával működik, amely azt jelenti, hogy az alacsony jelszintet logikai 1-nek lehet tekinteni, ezért szükség van egy felhúzó ellenállásra a rendszer helyes működéséhez, még akkor is, ha nincs aktív eszköz a buszon. Egy adatcsomagra a *20. ábrán* lehet példát látni.



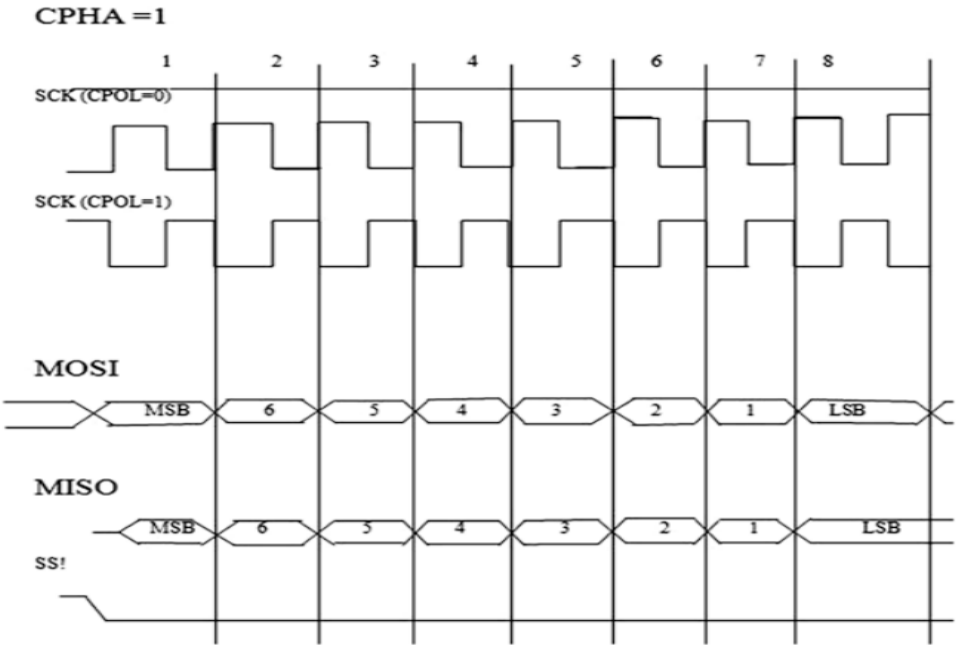
21.ábra. I^{2}C protokoll alapú kommunikáció kezdete és vége (forrás: [A14])

A kommunikáció kezdete egy start frame-mel indul, majd az adatcsomag következik, például 7 bit hosszan. Ezután egy bit jelzi, hogy az eszköz olvasni vagy írni fog-e. A *Master* egy nyugtát vár az utolsónak küldött bit után a *Slave*-től annak érdekében, hogy megkapta-e az adatcsomagot. A kommunikáció lezárását a *Master* egy stop bittel fogja jelezni. Függetlenül az üzenetküldés sikerességétől az SCL és SDA visszatér a magas állapotba a tápfeszültség környékére.

[4]

3.1.2. SPI protokoll

Az SPI kommunikáció során a *Master* az órajel segítségével irányítja az adatátvitelt, amikor kommunikációt kezdeményez egy *Slave*-vel.



22.ábra. SPI protokoll adatcsomag küldésére példa, amikor CPHA=1 (forrás: [A15])

A kommunikáció kezdetén a kiválasztott *Slave CS* lábát alacsony potenciálra állítja be, a többi buszra csatlakoztatott *Slave CS* lábát pedig magasra. Ezt követi a szinkronizáció, amelyet a *Master* biztosít egy előre meghatározott órajellel.

Az adatküldés során 8 bites, azaz 1 byte-s csomagokat képes küldeni a *Master* a *Slave*-nek, vagy a *Slave* a *Master*-nek. Mivel az SPI busz full-duplex működésre képes, így az adatok küldése és fogadása egyidejűleg zajlik a két modul között.

[10]

3.2. Eszközök közti protokollok

3.2.1. Modbus protokoll

A *2.3.1. RS-485 hálózat* című alfejezetben található az, hogy hogyan épül fel egy olyan buszrendszer, amelyen modbus protokollal lehet kommunikálni. A következőkben az ott leírtakra és a *[12] forrásban* található tudásra fogok támaszkodni.

Azért erre a forrásra hivatkozom, mert a program készítése során a protokollt nem én implementálom, hanem egy már kész program könyvtárat fogok használni. A könyvtár a forrás által említett funkciókkal rendelkezik. Forrásból származó ismeretek leírásával a célom az, hogy megismerjem a működését a protokollnak, illetve a saját programomba tudjam integrálni a létező könyvtárat.

A protokoll két kommunikációs módot kínál az adatfolyam felépítésére, ezek a ASCII és az RTU módok.

ASCII mód:

Start of Frame	Device Address	Function Code	Data	LRC Check	End of Frame
1 character (:)	2 characters	2 characters	n characters	2 characters	2 characters (CRLF)

23.ábra. ASCII adatcsomag felépítése (forrás: [A16])

23. ábra a Modbus üzenetek ASCII-framing-jét mutatja. A frame kezdete egyszerűen egy kettőspont (:), a keret vége pedig a két ASCII karaktert igénylő CRLF szekvencia. Az ASCII karakterek mindegyike 7 bit. A többi mezőben lévő összes többi karakternek vagy a 0-9 számoknak, vagy az A-F betűknek kell lennie, mivel az adatokat hexadecimális formátumban kell ábrázolni, de ASCII karakterek formájában kell megjeleníteni. (Például a 03-as funkciókódot két ASCII-karakterként „0” és „3” jelenítené meg.) Ugyanez vonatkozik az adatokra is. Az ASCII mód egyik előnye az időzítés. A karakterek között akár egy másodperc is eltelhet időkiesési hiba nélkül. Ezért

egy jó gépíró szimulálhat egy *Mastert* úgy, hogy egy CRT-terminálon beír egy karakterláncot egy *Slave* számára, és megfigyeli annak válaszát. Az üzenetek azonban meglehetősen tömörnek és hatékonynak bizonyulnak.

RTU mód:

Start of Frame	Device Address	Function Code	Data	CRC Check	End of Frame
4 character times	8 bits	8 bits	n x 8 bits	16 bits	4 character times

24.ábra. RTU mód adatcsomag felépítése (forrás: [A16])

RTU üzemmódban történő működés esetén az időzítés sokkal kritikusabb. Nincs specifikus *Start of Frame* karakter, ehelyett az üzenetkeret négy karakteres jelöléssel kezdődik. Ezt az intervallumot követően az eszközcím, majd a funkciókód és az adatok következnek. Vannak más különbségek is az ASCII üzenetkerethez képest, amint az a *24. ábrán* látható. Az ASCII-keretben alkalmazott *Longitudinal Redundancy Check* (LRC) helyett az RTU-keretben az adatok ellenőrzése robusztusabb, ami *Cyclic Redundancy Check* -kel (CRC) történik meg. A keret végének jelzése szigorúan négy karakteres jelölésen alapul.

Modbus regiszter parcellái

3. táblázat. Modbus funkciók *Master* és *Slave* között (forrás: [A16])

Code	1/16-bit	Description	I/O Range
01	1-bit	Read coils	00001 – 10000
02	1-bit	Read contacts	10001 – 20000
05	1-bit	Write a single coil	00001 – 10000
15	1-bit	Write mulitple coils	00001 – 10000
03	16-bit	Read holding registers	40001 – 50000
04	16-bit	Read input registers	30001 – 40000
06	16-bit	Write single register	40001 – 50000
16	16-bit	Write mulitple registers	40001 – 50000
22	16-bit	Mask write register	40001 – 50000
23	16-bit	Read/write mulitple registers	40001 – 50000
24	16-bit	Read FIFO queue	40001 – 50000

A *3. táblázat* foglalja össze, hogy milyen publikus funkciókkal lehet a *Master* és *Slave* között kommunikálni. Bizonyos funkciókódok meghatározott regisztertartományokat jelentenek. A korai PLC-k többnyire diszkrét bemenetekkel és diszkrét kimenetekkel foglalkoztak, amelyeket a regisztertérképen egy-egy bit képviselt. A Modicon PLC-k esetében a diszkrét kimenetek a 00001-es helyen kezdődnek, a diszkrét bemenetek pedig az 10001-es helyen. Mindegyikhez egy-egy bites

tárolóhely szükséges. A bemenetek, az úgynevezett érintkezők csak olvashatók, a kimenetek, a tekercsek, pedig olvashatók vagy írhatók.

Ahogy a PLC-k összetettsége növekedett, az analóg be- és kimenetek (I/O) kezelésére és a matematikai számítások végrehajtására való képesség is bekerült. A 16 bites regiszterreferenciák I/O tartománya 30001-nél kezdődnek a csak olvasható analóg bemenetek. 40001-nél az általános célú olvasási/írási regiszterek kezdődnek, amelyek analóg kimenetekként is szolgálhatnak, 40001 felett már valóban nincsenek korlátozások.

3.2.2. WIFI kommunikáció

Alapvetően a rendszer különböző önálló elemekből épül fel, amelyeknek kötelező egymással kommunikálnia. Az *espTouch*-nak és az *espCarryable*-nek kötelessége egymást a rendszerben fellépő változásokról értesíteni. Ezen rendszer adatváltozásokat WIFI-s kommunikáción keresztül fogom megoldani HTTP webszerver, és websockets *szerver* kialakításával, mindkét eszközön. HTTP webszerver segítségével a felhasználó távolról is tudja monitorozni a rendszert. A websockets *szerver* lehetővé teszi, hogy automatikusan frissülni tudjon a weboldal. Ezekről fogok a következőkben részletesen beszélni a *[13] forrás* alapján.

A HTTP egy alkalmazásszintű protokoll, amely a TCP/IP protokollcsomagra épül. Úgy tervezték, hogy *kliens-kiszolgáló* környezetben működjön, szigorúan elkülönítve a *kliensek* és a *kiszolgálók* közötti problémákat. Az ügyfél feladata, hogy egy HTTP-kérelem elküldésével adatokat vagy szolgáltatást kérjen a *kiszolgálótól*. A *kiszolgáló* viszont egy HTTP-választ küld, amely az ügyfél által kért művelet végrehajtásának eredményét tartalmazza.

A valós idejű web területén végzett kutatások a HTTP protokoll korlátait a natívan nem aszinkron kommunikációs paradigma továbbfejlesztésével és aszinkron működésre való hangolásával próbálták leküzdeni. Ennek eredményeképpen a bevezetett Websocket protokoll és a HTML 5 szabvány részeként kifejlesztett Websocket API jelenleg az egyetlen valódi aszinkron, full-duplex kommunikációs keretrendszer. A Websockets protokoll és API a TCP sockets kifejezőerejét, és rugalmasságát hozza el a webes alkalmazások számára.

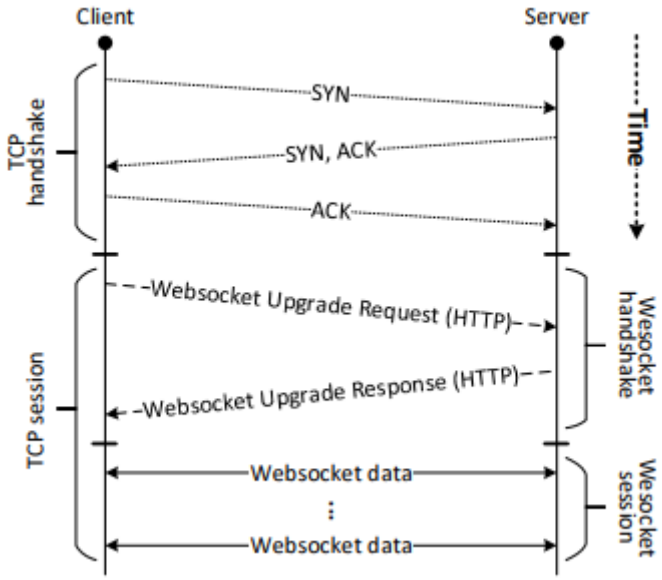
Websocket protokoll

Websocket protokoll egy alkalmazási szintű protokoll, amely a TCP-re épül. Lehetővé teszi a kétirányú adatátvitelt a webes munkamenetekben, és egyfajta életképes alternatívájaként tartják számon a HTTP polling technikáknak. A Websocket protokoll alkalmas a web-alapú, közel valós idejű forgalomra.

A Websocket protokoll visszafelé kompatibilis a meglévő webes infrastruktúrával. A Websocket

handshake megfelel a HTTP 1.1 specifikációnak, továbbá a Websocket-alapú kommunikáció örökölte az összes a meglévő webes infrastruktúra minden előnyét, mint például:

- Webböngészők natív támogatása, beleértve az eredetalapú biztonsági modellt is.
- Proxy és tűzfal áthidalása.
- URL-alapú végpontok, amelyek lehetővé teszik a többszörös, és potenciálisan korlátlan számú szolgáltatás futtatását egyetlen TCP-porton.
- A TCP által előírt hosszkorlátozások megszüntetése protokollban.



25.ábra. Websockets over TCP szekvenciális diagrammja (forrás: [A17])

A kapcsolat létrejöttéhez szükséges a TCP kommunikáció kiépítése, ami egy három kezes kézfogás. Ez azt jelenti, hogy a *kliens* küld a *szerver* felé egy szinkronizációs üzenetet, amire a *szerver* válaszol egy szinkronizációs és egy nyugtázó üzenettel egyszerre. Végül a kapcsolat létrejöttének megerősítésére a *kliens* is küld egy nyugtázó üzenetet a szinkronizációról.

A munkamenet létrehozásához a *kliens* Websocket frissítési kérelmet (Upgrade Request) küld a *kiszolgálónak*, amelyre a *kiszolgáló* Websocket frissítési választ (Upgrade Response) ad. Ettől kezdve a *kliens* és a *szerver* is képes aszinkron full-duplex üzemmódban adatokat küldeni oda-vissza.

A Websocket handshake a HTTP protokoll előnyeit használja ki a munkamenet létrehozásának fázisában. A Websocket frissítési kérelem egy hagyományos HTTP GET-kérelem, amelynek munkamenet végpontja a kérelem URL-összetevőjeként van megadva. A HTTP fejlécek Upgrade és Connection jelzik a *kiszolgálónak*, hogy az ügyfél a munkamenet további részében a hagyományos HTTP-ről a Websocket protokollra kíván váltani.

Ha a *kiszolgáló* támogatja a Websocket protokollt, akkor HTTP-választ ad, amelynek

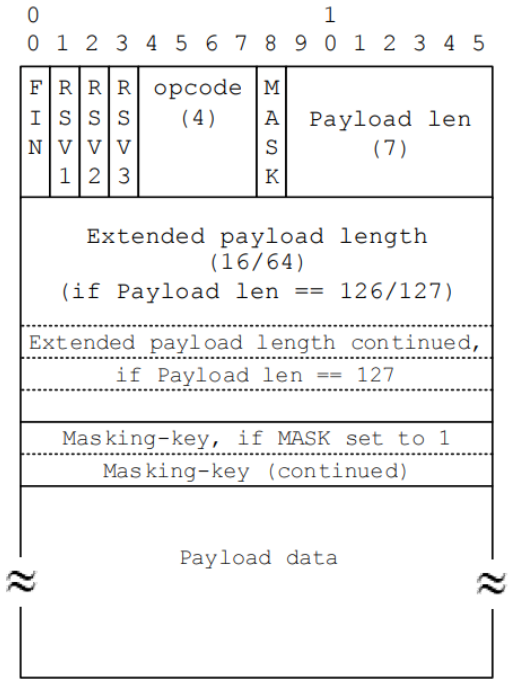
státuszkódja 101 Protokollváltás. Ettől a ponttól kezdve a HTTP-alapú kommunikáció befejeződik, és minden további kommunikáció a Websocket-kereteken alapul. (*25.ábra*)

- Websocket upgrade request:

  GET /chat HTTP/1.1 Host: example.com
  Upgrade: websocket
  Connection: Upgrade
  Sec-WebSocket-Key: example_key== Origin: http://example-webpage.com
  Sec-WebSocket-Protocol: chat
  Sec-WebSocket-Version: 13

- Websocket upgrade responed:

  HTTP/1.1 101 Switching Protocols
  Upgrade: websocket
  Connection: Upgrade
  Sec-WebSocket-Accept: example_acceptation_key=
  Sec-WebSocket-Protocol: chat

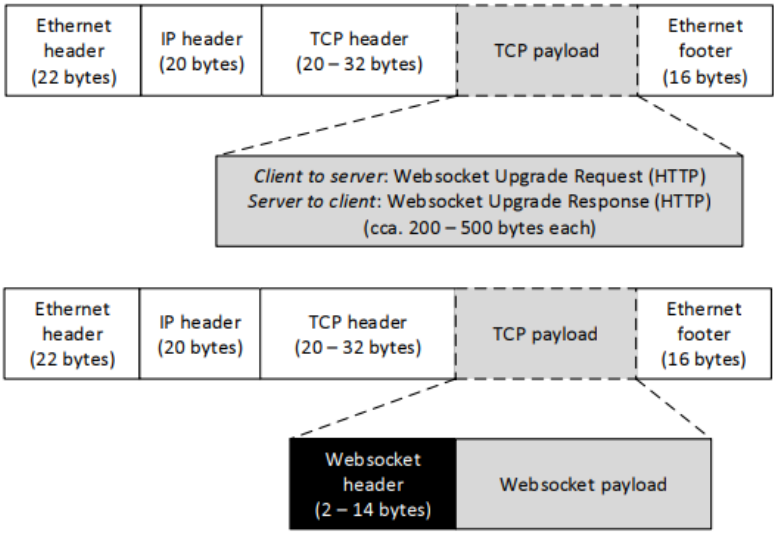


26.ábra. Websocket frame felépítése (forrás: [A17])

A Websocket-munkamenetben az adatátvitel egységeit Websocket-kereteknek nevezzük. A Websocket protokoll támogatja a bináris adatkereteket, a szöveges UTF-8 kódolású adatkereteket és a protokollszintű jelzést szolgáló vezérlőkereteket. A *26. ábrán* látható módon minden egyes hasznos adathoz minimális információ kerül hozzáadásra.

Minden Websocket-keret legalább 2 bájt hosszú fejléccel kezdődik, amelyet a hasznos adatok elé kell illeszteni. A hasznos adatok hosszától és a kommunikáció irányától függően a fejléc hossza akár 14 bájtra is nőhet.

Az első bájt vezérlőbiteket és 4 bites műveleti kódot (*opcode*) tartalmaz, amely meghatározza, hogy a hasznos adatot bináris adatnak, szöveges adatnak vagy protokollszintű jelzésnek kell-e értelmezni. A következő bit egy MASK, amely 0-ra állítva jelzi, hogy a hasznos adatok tiszta formában kerülnek-e átvitelre, egyébként pedig titkosítva. A következő 7 bit a hasznos adat hosszát jelzi. A hasznos adat hossza legfeljebb 125 bájt, ez az érték közvetlenül a Payload len mezőben van megadva. Ha a hasznos adat hossza 126 és 65535 bájt között van, akkor Payload len mező értékét 126-ra kell állítani, és további két bájtot adunk hozzá a tényleges hasznos adat hosszához. Végül, ha a hasznos adat 65535 bájtnál hosszabb, a Payload len mező értéke 127 lesz, és további 8 bájtot adunk hozzá a tényleges hasznos adat hosszához.



27.ábra. Websocket csomag az internetes csomagok között (forrás: [A17])

A *3. ábra* a protokollmezők beágyazását mutatja az internetes protokollhalmazba. Függetlenül attól, hogy a két fél közvetlenül a TCP protokollon vagy a Websocket protokollon keresztül kommunikál, az alacsonyabb szintű protokollmezők, beleértve a TCP fejlécet is, mindig jelen vannak minden hálózati csomagban. Tehát csak a Websocket-alapú kommunikáció által a sima TCP-alapú kommunikációhoz képest okozott többletköltséget, amely magában foglalja a Websocket handshake során továbbított két HTTP-üzenetet, valamint a munkamenet során továbbított minden egyes kerethez tartozó Websocket-keretfejlécet. A Websocket handshake overhead-je fix hosszúságú, és jellemzően néhány száz bájtot számlál. Mivel ez a kézfogás munkamenetenként egyszer kerül végrehajtásra, jelentősége minden egyes új, hasznos adatokat szállító Websocket-keretnél csökken.

Amikor adott mennyiségű hasznos adatot továbbítanak a sima TCP protokoll segítségével, ezek az adatok közvetlenül TCP hasznos payload-ként vannak beágyazva. Ha azonban Websocket-keretként kerül átvitelre, akkor a TCP hasznos payload a hasznos adatokból és a Websocket-keret fejlécéből is áll, amint az a *27. ábrán* látható.

4. Perifériák tervezése

Az *1.4. Innovatív megoldási javaslat* című alfejezetben nagyvonalakban bemutattam a tervezni kívánt rendszert, és ebben a fejezetben ezt fogom részletezni, pontosítani.

4.1 Rendszerterv készítése

A rendszereszközök implementációjához elengedhetetlen, hogy követelményeket fogalmazzak meg, hogy mit várok el az egyes eszközök működésétől, és mik azok a funkciók, amik szükségesek a megfelelő működéshez. Több követelményt is meg fogok fogalmazni, amelyek biztosítani fogják a helyes és stabilis működést. Ezek a következők:

- Rendszer követelmények
- Eszköz követelmények

Minden követelményt jellemzően rövid, lényegre törő mondatokkal fogom leírni, hogy a további tervezés és az implementáció során előre meghatározott lépések alapján készítsem el az egyes komponenseket.

4.1.1 Rendszer követelmények

A rendszer követelmény alatt azt értem, hogy mit várok el a teljes hálózat működésétől. Ezekből a követelményekből fogom meghatározni az eszközökre vonatkozó elvárásaimat.

Az eszközök feladata, hogy megbízhatóan a felhasználó preferenciái alapján egy adott háztartás belső hőmérsékletét a megfelelő hőmérsékletre szabályozza.

- Vezeték nélküli kapcsolat a *szerver* és *kliensek* között állandó és stabilis legyen.
- Eszközök távoli elérése monitorozás céljából.
- Rendelkezzen olyan beavatkozószervekkel, amelyek a felhasználót segítik a beállításokban.
- A fő eszköz(*szerver*/*espTouch*) képes legyen kommunikálni a *Controller for heating system-mel*, hogy az adott légterekben a beállításoknak megfelelő hőmérséklet legyen.
- *Kliens* (*espCarryable*) eszközzel dinamikusan bővíthető legyen a hálózat.
- Szoftveres meghibásodás esetén térjen vissza az utolsó stabil állapotába.

4.1.2 Eszköz követelmények

A hálózatot egy *espTouch*, és több *espCarryable* (akár nulla is lehet ebből), valamint egy router alkotja. A router független az általam tervezett rendszertől, és egyetlen *espTouch*-al is működtethető. Ebből következik, hogy az eszközök követelményeit úgy kell meghatározni, hogy azok kielégítsék a rendszer követelményeket.

Ezen elvárásokat kétfelé bontom, hogy az eszközök közti különbséget érzékeltessem, és az implementálás során tudjam, hogy mik azok a komponensek, amelyek különbözni fognak.

Közös követelmények:

- A DHT hőmérsékletmérő szenzor mérései pontosak legyenek.
- Állandó kapcsolat a router-rel.
- Weboldal működtetése, élő adat szolgáltatása, és ezek feldolgozása, mentése az adatbázisba.
- Helyes működéshez szükséges adatok beállítására szolgáló kezelőfelület.
- Jelezze ki az időt, kívánt és szenzorral mért hőmérsékletet, aktív fűtéstervet, páratartalmat, és a fűtéskör állapotát.
- Rendelkezzen dinamikusan bővíthető adatbázissal.

***espTouch*-csal szembeni elvárások:**

- Dinamikusan kezelje a csatlakozott *Klienseket* (*espCarryable*).
    - Weboldalon megjelenjen a *Kliens(ek)* elérhetősége(i).
    - Grafikus interfészen is látható legyen a *Kliens*(ek), és az(ok) adatai.
- Grafikus felületen keresztül lehessen beállítani:
    - Kívánt hőmérsékletet.
    - 24 órás fűtéstervet, órákra bontva.
    - Több különböző fűtésterv legyen elérhető.
    - Fűtés típusát (Gáz alapú vagy automatikus, ami a külső hőmérséklet függvényében fogja eldönteni, hogy a Gázkazán vagy a hőszivattyú lesz-e az aktív fűtőszerv.)
- Dinamikusan lehessen beállítani a fűtéskörök számát.

***espCarryable*-lel szembeni elvárások:**

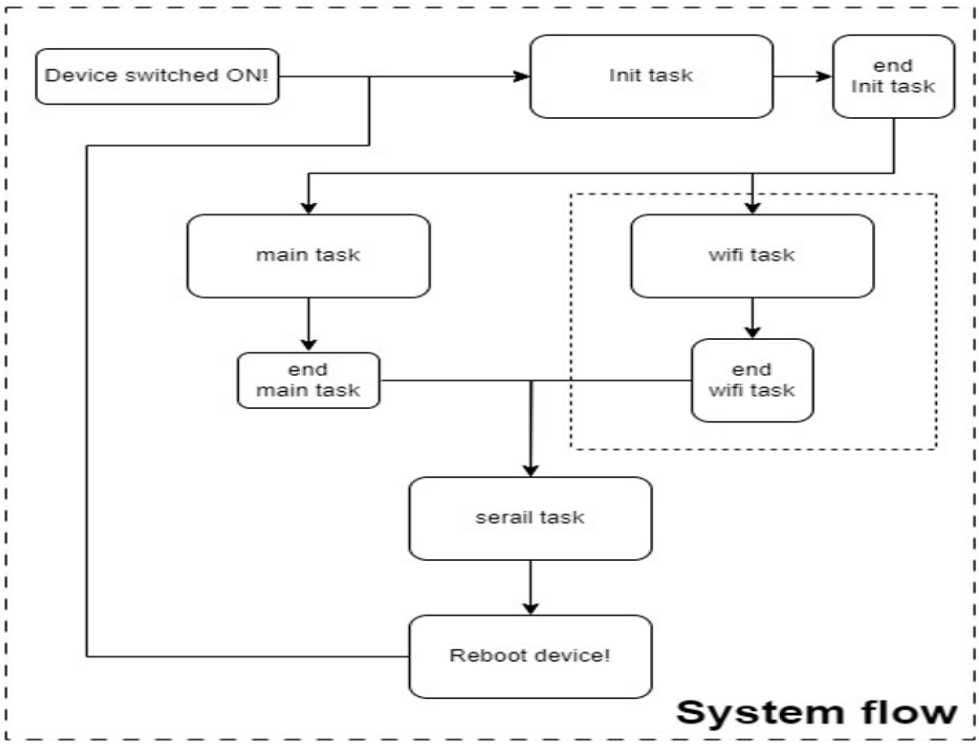
- Kapcsolódjon az *espTouch* webszerverére.
- Az aktív fűtéstervet és kívánt hőmérsékletet nyomógombos beavatkozó szervvel változtatni lehessen.

4.2. Hálózati modulok főszekvenciájának tervezése

A két eszköz központi modulja egy-egy ESP32-s mikroprocesszor, amely rendelkezik két processzormaggal, így ennek tudatában kell az eszköz követelményekből megtervezni a modulok szekvenciáit. Az elvárásokat halmazokra csoportosítom, és egy-egy követelményhalmaz fog alkotni egy-egy feladatot (*task*-ot).

A halmazok a következőek:

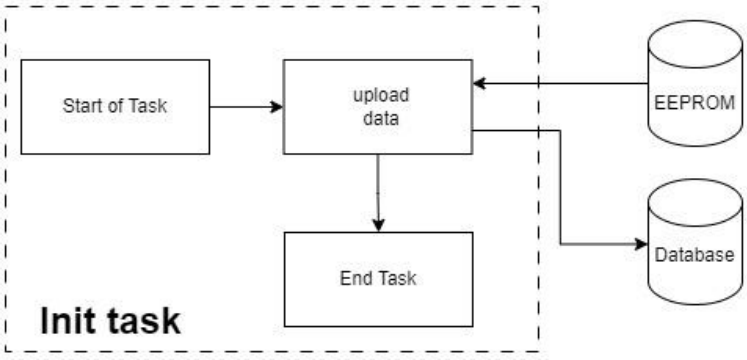
- Inicializáló feladat (*Init task)*
- Főfeladat (*main task*)
- Kommunikációért felelős feladat (*wifi task*)
- Eszköz beállításáért felelős feladat (*serial task*)



28.ábra. Rendszer működésének folyamatábrája

28. ábrán látható az általam megtervezett rendszer folyamatábrája. Amikor az *espTouch* vagy egy *espCarryable* bekapcsol, akkor ennek megfelelően kell működnie. Ha az *Init task* befejeződött, akkor az adott eszköz a *main task*-t és a *wifi task*-t fogja elindítani. Addig fognak futni ezek a programok, amíg a felhasználó ki nem kapcsolja az adott eszközt, vagy ha olyan beállítást akar állítani, amit majd az újból bekapcsoláskor az *Init task* be tud állítani az adatbázisban.

4.2.1. Inicializáló feladat



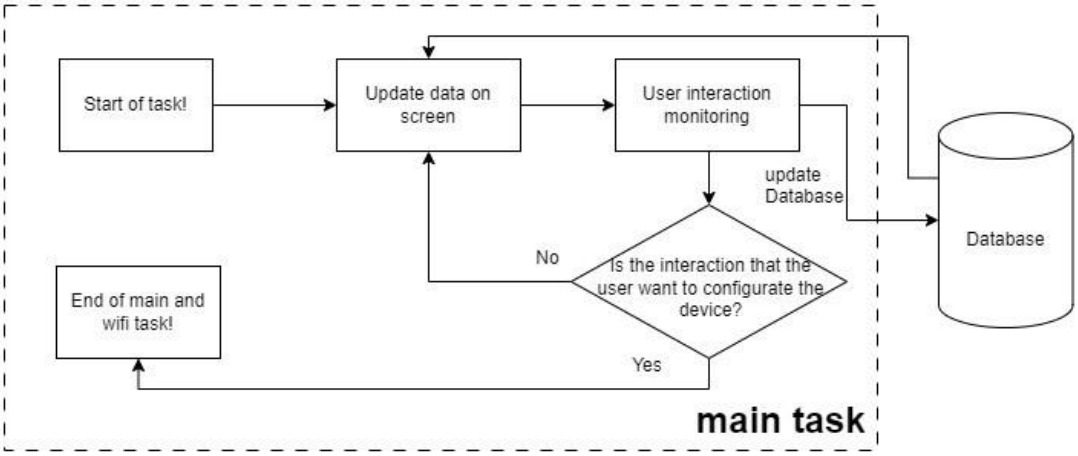
29.ábra. *Init task* folyamatábrája

Minden egyes bekapcsoláskor a rendszerelemek az EEPROM-ba mentett specifikus adatokkal fogják létrehozni az adatbázist, valamint ebből a memóriamodulból fogják feltölteni adatokkal a már létrehozott adatbázist. Amikor az adatbázis elkészült, akkor véget ér a feladat és a *main*, illetve a *wifi task* fog elindulni. (*29. ábra*) Az adatbázis adatai a felhasználó által beállított adatok lesznek, amiket a *serial task* futása során tud a felhasználó beállítani. Amíg üres az adatbázis, addig az eszközök egy *default* (általános) beállítás alapján fognak elindulni. A *default* beállítás önmagában annyit tesz lehetővé, hogy az *espTouch*/*espCarryable* elindítható állapotba kerüljön, hogy a felhasználó be tudja azt későbbiekben konfigurálni. Ez a feladat lényegében egy jól megszokott *bootloader*-ként értelmezhető.

A konfigurációs adatokról a *serial task* bemutatásakor lesz szó részletesseben.

Az adatbázist az eszközökre implementált kód nyelvén fogom elkészíteni, amelyet a későbbiekben fogok részletezni.

4.2.2. Főfeladat



30.ábra. *main task* folyamatábrája

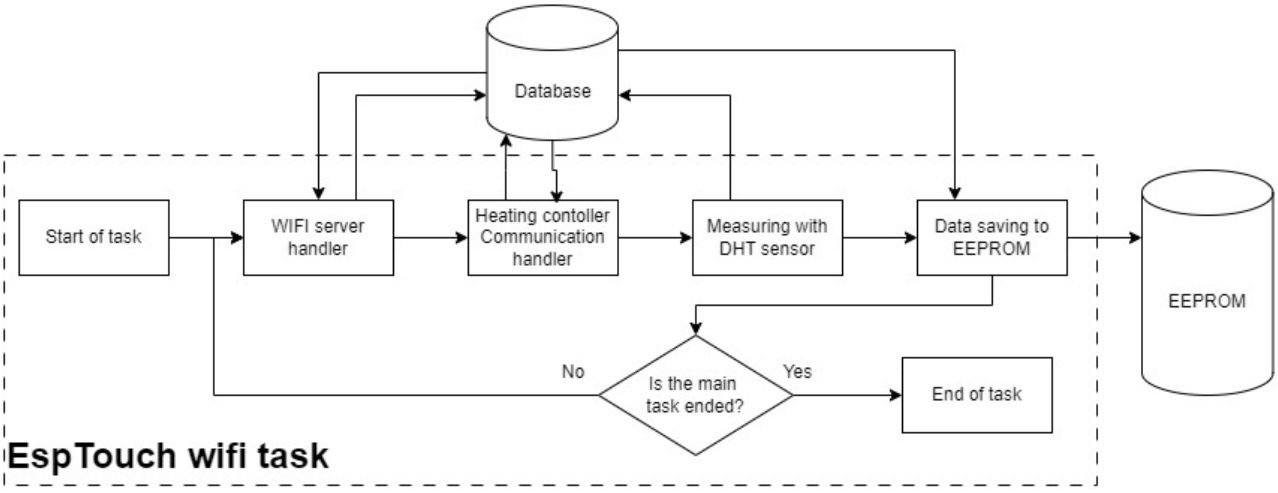
30. ábrán látható az általam megtervezett feladat folyamatábrája, amely biztosítani fogja, hogy az adatok kvázi valós időben frissüljenek a felhasználó számára, amely visszacsatolásként szolgál a rendszer helyes működéséről a felhasználó felé (*Update data on srceen*). Továbbá az *espTouch/espCarryable* rendelkezik beavatkozó szervekkel is, így a feladat ezen interfészeket is monitorozni fogja. Ha valamilyen beavatkozás történik, akkor a *user intreacion monitioring* frissíteni fogja az adatbázist.

Ez a feladat az ESP32 mikroprocesszor egyik processzormagján fog futni, amely biztosítani fogja a kvázi valós időben történő kijelző frissítését. Amikor a felhasználó interakcióba lép az eszközzel a beavatkozószerveken keresztül, akkor annak hatása a kijelzőn azonnal meg fog jelenni.

4.2.3. Kommunikációért felelős feladat

Mivel egy processzormag teljes mértékben le van foglalva, így a kettőből egyre maradnak azon elvárások, amelyek eleget fognak tenni az eszközök követelményeinek. Mivel a két modul egy rendszert fog alkotni és ennek a hálózatnak központi eleme az *espTouch* (*server*), így a *wifi task*-ban is lesz különbség.

espTouch wifi task:



31.ábra. *espTouch wifi task* folyamatábrája

A *31. ábrán* látható a megtervezett szekvencia. A funkciók a folyamatábrán komplexek, és azért felelnek, hogy az eszköz követelményeket teljesítsék. Fontos megjegyezni, hogy egyik funkcióban sem szabad túl sokáig várakozni.

WIFI server handler felelős azért, hogy stabil kapcsolatot teremtsen a routerrel, ezzel biztosítva bármiféle *kliens (*általában *espCarryable,* de lehet böngésző is, például Microsoft Edge*)* csatlakozását a *szerverre.* Minden egyes eszköznek egy HTML weboldalt kell üzemelnie websockets kommunikációval. A *fő szerver* (weboldal) az *espTouch-n,* a *kliens szerverek* pedig az *espCarryable-*

n működnek. Amikor a *kliens* csatlakozik a *szerverre*, akkor adatküldés és fogadás indul el az eszközök között. Ezen megosztott adatokat mindkét fél a saját adatbázisában fogja eltárolni, hogy a *main task* hozzáférhessen az új adatokhoz.

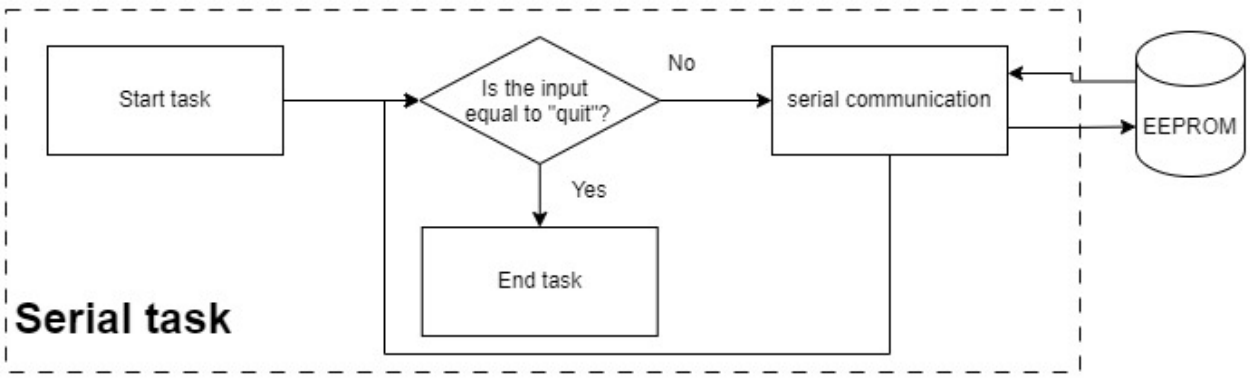
Heating controller communication handler feladata az, hogy üzenjen a *controller for heating system*-nek, ha bármilyen beavatkozás szükséges. Ezek a beavatkozások a fűtés indítása vagy leállítása lehetnek, továbbá, hogy milyen típusú fűtőeszközzel működjön a rendszer (Gázkazán vagy hőszivattyú).

Measuring with DHT *sensor* feladata, hogy a DHT11-s szenzorral megmérje a szoba hőmérsékletét. Az ESP32-höz van csatlakoztatva a szenzor, amihez One-Wire buszt használok.

Data saving to EEPROM feladata, hogy a fontos adatokat elmentse az EEPROM-ba, ha megváltozik az adott adat, vagy ha a felhasználó mentési funkciót kér a *main task*-n keresztül.

A *wifi task*-nak akkor lesz vége, ha a *main task* befejeződik. Ez akkor lehetséges, ha a felhasználó konfigurálni szeretné az eszközt.

4.2.4. Eszköz beállításáért felelős feladat



32.ábra. *serial task* folyamatábrája

A *serial task* feladata, hogy a termosztát rendszert a kialakított vagy még tervezett fűtésrendszerhez lehessen alakítani. A felhasználó vagy a fűtésrendszert kivitelező személy beállíthatja ezen feladaton keresztül, hogy milyen előre meghatározott paraméterekkel dolgozzon az *espTouch* és az *espCarryable*. A szükséges paraméterek azok, amikkel egy dinamikus adatbázis hozható létre, és ezzel együtt dinamikus rendszert lehet működtetni.

A rendszer működéséhez kell egy router, így ennek paramétereit lehet elmenteni az EEPROM-ba (SSID, jelszó). Mindkét típusú eszköz egy-egy webszervert is működtetni fog, így a kommunikációhoz szükséges átjáró megadására is lehetőséget kell biztosítanom. Tehát egy port számot is meg kell adni, amit érdemes úgy megválasztani, hogy könnyen hozzáférhetőek legyenek a webszerverek a felhasználónak, például: 80-s.

A fő egység az *espTouch,* egyszerre több fűtéskört is képes monitorozni, és mellette ez az eszköz is része az egyik fűtéskörnek. Így a *Serial task*-ban be kell tudni állítani, hogy hány fűtéskörrel rendelkezik a háztartás, amelyben el lesz helyezve, és hogy az *espTouch* ezek közül melyiknek a része. Az *espCarryable*-nél ezeknek a paramétereknek egy saját egyedi azonosítónak kell lenniük, és egy olyan azonosítónak, amely meghatározza, hogy a *Kliens* eszköz melyik fűtéskörhöz tartozik.

Fontos beállítani továbbá azt is, hogy az *espTouch* milyen paraméterekkel tudjon kommunikálni a *Controller for heating system*-mel, mivel a két eszköz között *RS-485* busznak kell biztosítania a fizikai összeköttetést, amely modbusz protokollt használ. Így a beállítandó elemek lennének az olvasni vagy írni kívánt regiszternek a címe és annak száma, továbbá a *Controller for heating system* azonosító száma. Fontos megjegyeznem, hogy ezeket a paramétereket csak az *espTouch* esetében lehetséges meghatározni.

Egy rendszeren belül több egyforma eszköz is lehetséges és ezek sokféle módon meg vannak különböztetve egymástól. Ettől függetlenül a felhasználó számára is fontos lenne megkülönböztetni az eszközöket. Így lehetőséget biztosítok arra is, hogy különböző nevekkel (*NickName*-mel) lehessen ellátni a két modult.

Természetesen egy fűtésterv 24 elemből áll, azaz minden órához tartozik egy hőmérséklet. A rendszer ezeket az értékeket akarja elérni működése során, így fontos, hogy az eszközökön az órát is be lehessen állítani. Ez alatt azt értem, hogy hány óra hány perccel *boot*-oljanak be az egyes eszközök.

Azért, hogy biztosak legyenek a beállítást végző személyek, lehetőség van arra is, hogy minden beállítható értéket ki lehessen listázni, és ha szükséges, akkor minden paramétert egy paranccsal ki lehessen törölni. Azaz egy *default* értéket vegyenek fel, amelyekkel az eszközök hibátlanul be tudnak majd *boot*-olni.

(*32. ábra*)

4.3. *espTouch* periféria felhasználói interakcióinak tervezése

Az *espTouch* kapcsolási rajzai az *1.* és *2. számú* mellékletben tekinthetők meg, a rajzok ismertetik az eszköz felépítését.

Az *espTouch* központi eleme egy ESP32-s mikroprocesszor, amelyhez egy rezisztív interfész és egy LCD kijelző van csatlakoztatva. Az érintőkijelzőt és a processzort SPI busz köti össze. Ez a beavatkozó szerv lesz a felelős a felhasználói interakciókért. Továbbá egy DHT11 hőmérséklet és páratartalom mérő szenzorral van ellátva, amely One-Wire buszon keresztül csatlakozik a mikroprocesszorhoz. Ez a DHT11-s szenzor a *2.1.5. One-Wire busz* című fejezet szerint csatlakozik a központi elemhez, és a modul adatlába az ESP32 27. lábára csatlakozik, amely egy ki/be -meneti

port.

4.3.1. Beavatkozószerv funkcióinak megtervezése

A *4.1. Rendszerterv készítése* című alfejezetben ismertetett követelmények alapján tervezem az interfész design-t, hogy a felhasználó a preferenciáit kényelmesen, és egyszerűen tudja beállítani.

Tervezéshez a SquereLine Studio-t használom, amely kifejezetten a projektemhez és ehhez hasonló projektek készítésében nyújt segítséget. Az alkalmazásban ki lehet választani a hardvert és a kijelzőt is, amelyre a design készül. Továbbá előre meghatározott elemekkel és funkciókkal lehet használni. Az elkészült design-ból az alkalmazás képes programkódot generálni, amely animáció hűen tud viselkedni, de szükséges logikai összekapcsolásokat már a programozónak kell megcsinálnia.

Összesen négy oldalas GUI-t (graphical user interface) tervezek az *espTouch*-ra, amely biztosítani fogja, hogy a felhasználó be tudja állítani a fűtésterveket, és akár az egyéni kívánt hőmérsékletet, fűtés típusát. Továbbá a GUI az adatbázis fontos elemeit fogja megmutatni.

33.ábra. *espTouch* főoldala

33.ábrán az eszköz fő oldala látható, ahol a felhasználó a kör alakú csúsztatható gomb segítségével beállíthatja, hogy mekkora legyen a kívánt hőmérséklet. Ennek értékét középen a legfelső szám fogja megmutatni, alatta pedig a mért hőmérséklet és páratatalom lesz látható. Ha megváltozik a főoldalon a beállított hőmérséklet, akkor a következő egész óráig az eszközön ez a kívánt hőmérséklet lesz látható, és *Auto* felirat *Manual* feliratra változik meg. Így lényegében csak átmenetileg fog megváltozni a kiválasztott programterv. Ha a kívánt hőmérséklet kisebb, mint a mért hőmérséklet, akkor az *OFF* feliratot *ON*-ra fogja váltani. Továbbá itt lesz látható az idő, és az eszköz IP címe is, ha sikerült a külső routerhez csatlakoznia. Két gombot terveztem még erre a kijelzőre, amik két különböző GUI oldalt jelenítenek meg.

4.3.2. A *Programs* aloldal

34.ábra. *Programs* aloldal

34. ábrán be lehet állítani a programterveket, ebből összesen öt van. A beállítás nagyon egyszerű, a *prog 1* mezőben a programtervet lehet kiválasztani, és mellette ugyan így azt az időpontot, ahol a kívánt hőmérsékletet szeretné a felhasználó változtatni.

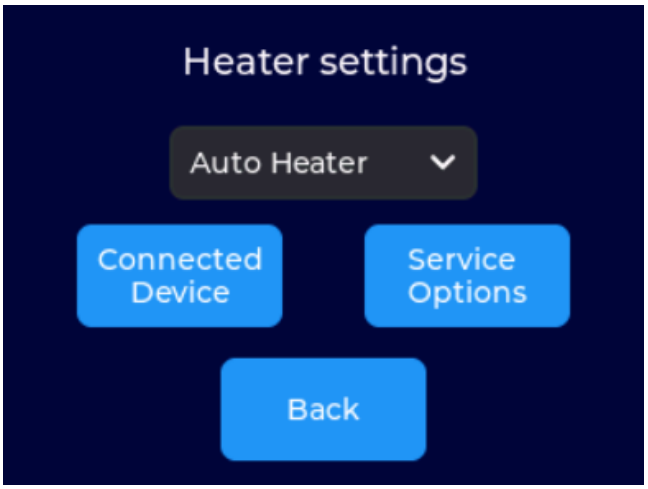
35.ábra. programterv és időpont választó interakció

A programterv és időpont a *35. ábra* szerint van létrehozva. A *Programs* oldalon mindig csak a kiválasztott fog látszódni, az ábrán ez a *Item 3*. Függőleges interakcióval meg lehet változtatni a kiválasztott elemet.

Ha kiválaszt a felhasználó egy időpontot, akkor a *Wanted Temperature* alatt a csúszó gomb pozíciójával meg lehet változtatni az időponthoz tartozó hőmérséklet értékét, amely a grafikonon azonnal láthatóvá válik. A csúszó gomb melletti szám jelezni fogja, hogy a kiválasztott időpontban milyen érték került beállításra.

A *Save* gomb lenyomásakor az összes változás az EEPROM-ba kerül mentésre azért, hogy amikor az eszköz újra indul, akkor a változások maradandóak legyenek. Ha a felhasználó nem nyomja le a gombot, akkor a változások szerint fog az eszköz működni mindaddig, amíg újra nem indul az *espTouch*. A *Back* gombbal a főoldalra lehet visszalépni.

4.3.3. Az Options gomb következménye



36.ábra. *Options* aloldal

36. ábrán több funkció is elérhető lesz, ezek a következők:

- *Back* gomb: visszaváltja kijelzőt a főoldalra.
- *Service Options* gomb lenyomásakor *serial task-t* fogja betölteni, a kijelző elsötétül és a *main,* illetve a *wifi task* befejeződik.
- Ki lehet választani a fűtés típusát (*Auto Heater/ Gas Heater*). A típusra „kattintva” egy lenyíló ablak fog megjelenni, ahol ki lehet választani a preferált típust.
- *Connected Device* gomb megnyomásakor egy újabb ablak töltődik be.

Connected Device **oldal**

37.ábra. *Connected Device* aloldal

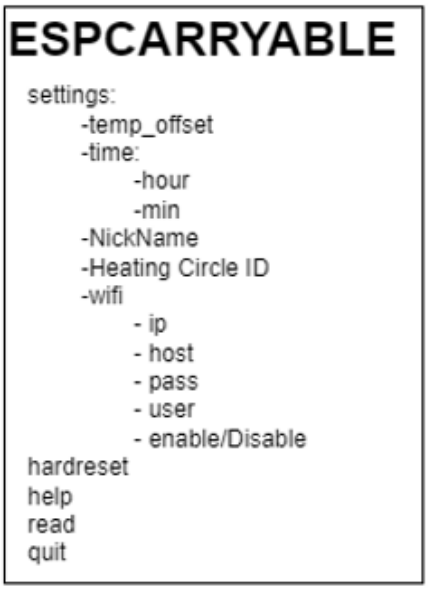
37. ábrán látható oldalon a *szerverhez* kapcsolódott *espCarryable* adatait lehet vizsgálni. Többek között azt is, hogy az adott eszköz mekkora hőmérsékletet mér, mi a MAC címe, és hol helyezkedik el/hogy hívják a perifériát. Ez az oldal visszacsatolásként szolgál a felhasználónak, hogy

minden használt *espCarryable* csatlakozva van-e a központi elemhez, az *espTouch*-hoz.

4.3.4. *serial task* felhasználói oldalának tervezése

A *serial task* mind a két eszköznél ugyan úgy fog felépülni, de más tartalom lesz elérhető a felhasználó számára. Így úgy kell megterveznem, hogy használat szerint megegyezzenek, de tartalmát implementáláskor könnyűszerrel lehessen bővíteni.

Mivel a soros porton való kommunikáción keresztül lesz elérhető ez a funkció, így a felhasználó a csatlakozott számítógépen egy konzolban fogja majd látni az adatokat. Ezen keresztül tud kommunikálni az eszközzel parancsszavak felhasználásával.



38.ábra. példa az *espCarryable* menürendszer felépítésére

A *38. ábrán* látható a tervezett menürendszer. Alapvetően a menünek tetszőlegesen bővíthetőnek kell lennie és fa struktúrát kell követnie. Ez azt jelenti, hogy minden funkciónak egy kijelölt menühöz kell tartoznia. Egy vagy több menüelem része lehet egy másiknak, amely a szülő/ős menü lesz. Kivételt fog képezni a fő menü, amely a *serial task* betöltésekor megjelenik a felhasználó számára. Ezen menürendszer implementálásáról a későbbiekben lesz szó.

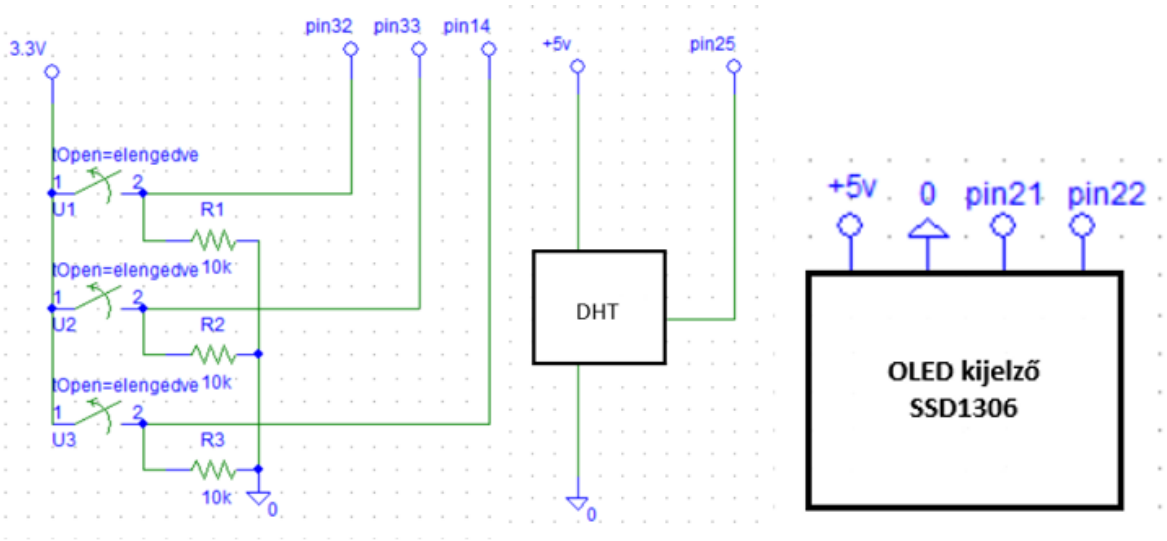
Mivel minden hívható applikációnak megvannak a saját tulajdonságai, így ezen tulajdonságok alapján lesznek meghatározva az elérhető funkciók, például *settings* egy almenü, amely a beállítható paramétereket foglalja magába, egyben ez lesz a parancsszava is az almenünek.

Mivel parancsszavakkal elérhetőek az egyes funkciók, így két típusú elérést biztosítok a felhasználónak. Vagy parancsszavanként halad célja felé, vagy minden parancsszót felhasznál az elérni kívánt funkcióhoz, például „*settings wifi ip 192.168.0.103*”. Utóbbiból az kell, hogy következzen, hogy a megadott IP címet az eszköz elmenti az EEPROM-ba, ha a szintaxisa megfelelő.

A parancsszavankénti megközelítést azért adom hozzá a terveimhez, mert a felhasználók általában nincsenek tisztában eszközük pontos működésével. Ebből következik, ha valamit be szeretne állítani, akkor „*help*” paranccsal az aktuális menü elemeit fogja kilistázni, és ezek alapján fel tudja térképezni a modul belső funkcióit.

4.4. *espCarryable* felhasználói interakcióinak tervezése

4.4.1. *espCarryable* fizikai kialakítása



39.ábra. *espCarryable*-t kiegészítő kapcsolások

A *39. ábrán* látható, hogy milyen elemekkel egészítem ki a *3. mellékletben* szereplő modul kapcsolását. Az *espCarryable*-t három nyomógombbal, egy hőmérséklet érzékelővel (DHT), és egy kisméretű *OLED kijelzővel* láttam el.

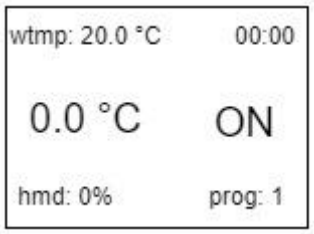
A három darab nyomógomb 5 funkciót fog ellátni. A kívánt hőmérséklet szabályozását az *U1* és *U2* kapcsolók végzik pozitív és negatív irányban, egytizedes tűréssel. A harmadik kapcsoló *U3* lenyomásával a programot lehet majd változtatni. Ha az *U3*-t nyomva tartjuk, akkor *U1* és *U2* segítségével tudjuk a kívánt programot kiválasztani. Ha a felhasználó *U1* és *U2* nyomógombokat egyszerre nyomja le, akkor a programnak a *serial task*-t kell elindítania, és minden más futó *task*-nak le kell állnia. (*39.ábra*)

A hőmérséklet mérésért felelős DHT szenzor a mikroprocesszor 25-s lábára fog csatlakozni. Ezen a pin-n keresztül fog az *espCarryable* kommunikálni a hő és páratartalom mérő szenzorral One-Wire buszon keresztül. (*39.ábra*)

A felhasználó számára fontos adatokat egy OLED kijelzőn fogom megjeleníteni, amely I^2C buszon keresztül kapcsolódik az ESP32-höz. A kommunikációhoz szükséges lábak az SCL és az

SDA, az *espCarryable* 21-s lába lesz az SCL, a 22-s pedig a SDA. (*39.ábra*)

Kijelző design tervezése



40.ábra. OLED kijelző dizájnja

A kijelzőn több adatot is tervezek megjeleníteni, ezek a kívánt hőmérséklet(*wtmp*), az óra (*00:00*), a szenzor által mért hőmérséklet (*0.0 °C*) és páratartalom (*hmd:*), a fűtés rendszer állapota (*ON*), és végül a kiválasztott aktív fűtésterv (*prog:*).

Ezen adatok az eszköz monitorozására, illetve a beiktatott változások ellenőrzésére alkalmasak, hogy megváltozott-e az adott adat az adatbázisban. Mivel egy rendszer része lesz az eszköz, így nem csak nyomógomb segítségével lehet beavatkozni, hanem a server maga is megváltoztathat adatokat. Így a kiválasztott adatok kvázi valós idejű frissítése a kijelzőn segít ellenőrizni a rendszer helyes működését.

(*40. ábra*)

5. Modulok szoftveres implementációja

Az ESP32 mikrovezérlő (MCU) különböző módszerekkel programozható, például C vagy C++ nyelven, illetve Python nyelven is. A probléma az, hogy a piacon számos ESP32 modul található, amelyekhez egyedi nyomtatott áramkört kell tervezni annak érdekében, hogy a processzorra *flash*-elni lehessen az elkészült programot. Az ESP32 UART-n keresztül lehet *flash*-elni. Ennek protokollja nem kompatibilis az USB-s kommunikációs protokollal, így szükség van egy *bridge*-re, amely biztosítja a konvertálást a két kommunikációs protokoll között. A *bridge*-t egy chip fogja biztosítani, amely a CP2102-s lesz, amelyet a Silicon Labs fejlesztett ki, adatlapja a *[11] forrásban* megtalálható. A chip lehetővé teszi, hogy a számítógép felismerje, mint csatlakoztatott eszközt, de ehhez telepíteni kell az adott számítógépre a chip illesztő programját, amit a Silicon Labs weboldalán keresztül lehet elérni.

A program elkészítéséhez, annak lefordításához, és MCU-ra *flash*-eléséhez további programokra van szükség. Programozásához számos fejlesztői környezet áll rendelkezésre, amelyek különböző programozási nyelveket és eszközöket támogatnak. Az Arduino IDE egyszerű és könnyen használható, támogatja a C/C++ nyelvet, és széles körben elterjedt. A PlatformIO szintén C/C++ alapú, és modern, rugalmas fejlesztői környezet, amely integrálható a Visual Studio Code-dal is. Az Espressif IoT Development Framework (ESP-IDF) az Espressif Systems hivatalos keretrendszere, amely mélyebb hardverszintű programozást tesz lehetővé C nyelven.

A MicroPython egy könnyű Python implementáció, amely gyors prototípus-készítést és oktatási célokat szolgál. A NodeMCU firmware lehetővé teszi az ESP32 programozását Lua nyelven, egyszerűsítve a fejlesztést. Az Espruino JavaScript interpretáló segítségével JavaScript nyelven is programozhatunk, ami szintén gyors prototípus-készítést tesz lehetővé. Végül, az Eclipse IDE, az ESP-IDF pluginnel kombinálva, egy robusztus és bővíthető fejlesztői környezetet kínál C/C++ nyelven. Ezek az eszközök különböző előnyöket kínálnak, így a választás az alkalmazás követelményeitől és a fejlesztő preferenciáitól függ.

Az Arduino IDE sokak által kedvelt, és rendelkezik egy platformmal, amely a fejlesztők számára elérhető. Erre a platformra különböző megoldásokat lehet feltölteni, illetve letölteni, amelyek ESP32-vel is kompatibilisek. Továbbá az Arduino IDE-ba beépített fordító és *flash*-elésre szolgáló applikáció is van. Ezek miatt a tulajdonságok miatt választottam ezt az alkalmazást, hogy a megírt programot *flash*-eljem a mikroprocesszorra. A könnyű program készítése és a jobb átláthatóság érdekében viszont a Visual Studio Code-t fogom használni.

5.1. Adatbázisok felépítése

Az eszközök adattárolására olyan adatbázist készítek, amelyet az inicializáláskor az eszköz maga képes felépíteni, és működése közben dinamikusan tud változni. Az automatikus adatstruktúra felépítéséről a *4.2.1. Inicializáló feladat (Init task)* című alfejezetben megtervezett *Init task* fog gondoskodni, amely az EEPROM-ban tárolt adatok alapján fog dolgozni. Ahhoz, hogy tudjam melyek a fontos építő adatok, ahhoz meg kell határoznom, hogy milyen adatokkal fog dolgozni mind a két eszköz.

5.1.1. Az *espTouch* adatbázisa

Adatbázis építést C/C++ nyelven fogom implementálni, így külső adatbázist kezelő alkalmazás használatára nincs szükségem. Programozás során objektumorientált programozási módszert fogok használni, amely azt jelenti, hogy az adat struktúráimat egy halmazba gyűjtöm a hozzájuk tartozó funkciókkal együtt. Ennek előnye, hogy Globális változóktól mentesen tudok az objektum belső változóihoz hozzáférni, valamint egy objektum lehet részese, leszármazottja vagy őse/szülője egy másiknak.

Az eddigiek alapján a következő objektumokat kell létrehoznom, hogy megfelelő adatbázissal rendelkezzen az eszköz, valamint a következő belső változókkal fognak ezek az osztályok rendelkezni.

- Fűtést kezelő objektum (*Heathandler*)
- Fűtésprogram, és kívánt hőmérséklet tartalmazó osztály (*programs*)
- WIFI kommunikációs adatok tárolására szolgáló objektum (*wifi_data*)
    - Tartalmazza a Router csatlakozási adatait (SSID, jelszó).
    - Webszerver működési portját.
- Idő és időzítésért felelős osztály (*Clock*)
    - Időt, mint óra és perc.
    - Két flag-t, amelyek olyan információval bírnak, hogy változott-e az óra és perc értéke.

Ezeket az objektumokat fogja tartalmazni a *DataHandler* osztály, amelyet a program működése során dinamikusan tud majd bővíteni, illetve zsugorítani. Ezek a beágyazott objektumok csak egy-egy elérési utat biztosítanak a működéshez szükséges adatokhoz.

Fűtésprogram, és kívánt hőmérséklet tartalmazó osztály

A *programs* osztály feladata, hogy tárolja, és frissítse az ott megtalálható adatokat. Különböző flag-eket tartalmaz, ezek segítenek a változások lekezelésében.

Attribútumai a következők:

- *activ_program_index*: Ez a változó fogja az aktív fűtésprogram számát tartalmazni, amit a felhasználó ki tud választani.
- *ProgHour_index*: Minden órához tartozik egy hőmérséklet érték, amely a fűtéstervben megtalálható. Ez a változó aktuális órához tartozó értéknek útvonalát fogja biztosítani.
- *Programs*: A változó egy előre meghatározott méretű tömb, amely fűtésprogramokat képes eltárolni. Olyan tömbökről van szó, amelyek 24 elemből állnak, és bennük minden egyes elem egy-egy hőmérséklet értékét tárolja.
- *Wanted_temp*: A kívánt hőmérséklet értéke, ha a felhasználó a GUI oldalán állítja át, akkor ez a változó fog változni, nem pedig a *Programs* változóban fog változni a megfelelő elem. Így biztosítom, hogy az adatváltozás a Programs változóban akkor jöhessen csak létre, ha a felhasználó azt a megfelelő beavatkozási oldalon teszi.

Flag-ek:

- *active_program_index_changed*
- *ProgHour_index_changed*
- *program_changed*
- *wtmp_wtmp_changed*
- *server_update_wtmp*
- *server_update_prog_index*

Ezek a flag-ek *bool* típusúak, és biztosítják a megfelelő logikai összeköttetést az egyes funkciók között. A GUI változói az adatbázis elemeit a saját változóiba másolja. Ha valami megváltozik felhasználó interakció során vagy *klienst*ől kapott üzenet során, akkor ezen flag-ek állításával el lehet érni azt, hogy a GUI változói csakis akkor frissüljenek, ha valamilyen változás történt a rendszerben. Azért előnyös ezeket a flag-eket használni, mert így fel lehet gyorsítani a program működését, mivel nincs felesleges adatmásolás.

Funkciók:

A *programs* osztálynak alapvetően rengeteg funkciója van, ezek lényegében arra szolgálnak, hogy a változókat és flag-eket be lehessen állítani, és ki lehessen olvasni a tartalmait.

Fűtést kezelő objektum

Az *HeatHandler* osztály feladata, hogy minden csatlakozott eszközt a megfelelő fűtéskörhöz csoportosítsa, és eltárolja ezen eszközök adatait. El kell tudnia tárolnia az *espTouch* szenzora által mért hőmérsékletet és azt, hogy a szenzor melyik fűtéskörön fog elhelyezkedni.

Attribútumai:

- *HeatingCirclesHandler*: Olyan objektumtömb, amely a csatlakozott eszközök és szenzorok adatainak tárolásáért felel.
- *number_of_HeatingCircles*: A létrehozott objektumtömb nagyságát, elemszámát fogja tartalmazni.
- *status*: A változó egy *bool* tömb, amelynek nagysága megegyezik a *HeatingCircles* változóval, és tartalmazni fogja, hogy melyik fűtéskörön szükséges fűtést bekapcsolni.
- *Heating_mode*: A kiválasztott fűtés típusát fogja eltárolni, amely vagy Gázkazán általi fűtés lesz, vagy Hőszivattyú általi.
- *modbus*: Ez a változó a *modbus_data* objektumhoz tartozik, amely a modbus kommunikációhoz szükséges adatokat fogja tárolni.

Flag-jeit illetően egy darab van, *heating_mode_changed*. Ez arra szolgál, ha a fűtésmódja megváltozik, akkor a rendszernek ezt azonnal közölnie kell a *Controller for heating system*-mel, hogy a megfelelő típusú fűtésmóddal fűtsön, ha szükséges.

Funkciói:

- *set_modbus_communication*: Ezen függvénnyel lehet eltárolni a modbusz kommunikációhoz szükséges adatokat.
- *add_HeatingCircles*: Hozzá lehet adni fűtéskört a rendszerhez, amely előre meghatározott számú, mivel az eszköz konfiguráláskor be lehet állítani, hogy hány fűtéskörrel rendelkezik a teljes rendszer.
- *add_device_to_HeatingCircles*: A csatlakozott *kliens* adatait lehet ezen a függvényen keresztül hozzá adni az adatbázishoz.
- *remove_device_from_HeatingCircles*: Ha egy *kliens* lecsatlakozik a hálózatról, akkor ezen funkció segítségével fog törlődni véglegesen az adatbázisból.
- *add_Measuringsensor*: A konfiguráció során meg kell határozni, hogy az eszköz melyik fűtéskörön fog elhelyezkedni. Ez a funkció a meghatározott értékhez tartozó fűtéskörön fogja eltárolni saját adatait és méréseit.
- *get_HeatingCircles_status*: A kívánt hőmérsékletet átadva, meghatározza minden fűtéskörön azt, hogy kell-e fűteni vagy sem. Minden fűtéskörre vonatkozó eredményt a

status tömbbe fogja menteni a függvény.

További funkcióival a változókat, és az objektumváltozókat lehet elérni.

Fűtésköröket kezelő objektum

A *HeatingCirclesHandler* objektum feladata, hogy dinamikusan hozzá lehessen adni az adatbázishoz a csatlakozott *klien*sek adatait, és eltárolja az *espTouch* adatait.

Belső változói:

- *measuring*: *TemperatureMeasuring* objektum típusú, itt lesznek az eszköz adatai eltárolva.
- *devices*: *DeviceComponents* osztály típusú, ahol a *klien*sek adatai lesznek eltárolva.

A változók csak elérési utat fognak biztosítani az adatokhoz, így funkcióival visszakapjuk az adatokra mutató elérési utat, ezáltal adatokat lehet módosítani, vagy új *klien*seket lehet hozzáadni az objektumhoz.

A *measuring* struktúrában az eszköz adatai, szenzor által mért hőmérséklet és páratartalom lesz eltárolva. Az eszköz saját adatai:

- Melyik fűtéskörön helyezkedik el.
- Mi a neve az eszköznek.
- Szenzor *offset*, amely megmutatja, hogy mennyivel kevesebbet vagy többet mérjen az DHT szenzormodul.

A *devices* pointer egy elérési útvonalat biztosít a csatlakozott *klien*seket tároló objektumhoz, amely *Components* osztály típusú. Ez a típus elfogja tárolni a *klien*sek által mért hőmérsékletet, a nevét, egyedi azonosítóját, MAC címét, IP címét. Az objektum létrehoz ezen adatokból egy panelt, amelyen ezek az adatok szerepelnek. A panel a GUI-ban fog megjelenni a *4.3.3. Az Options gomb következménye* című alfejezet *34. ábrája* szerint. Minden panel természetesen egyszer fog megjelenni a grafikus felületen.

A *Components* típusú objektum belső funkcióival lehet az adatokat frissíteni, lekérdezni, és megjelenési pozíciót frissíteni. Utóbbi fogja biztosítani, hogy minden panel egymás után helyezkedjen majd el a GUI *Connected Device* oldalán.

5.1.2. Az *espCarryable* adatbázisa

Az *espTouch* működéséhez szükséges adatbázis összetett és rendelkezik minden olyan elemmel, amely az *espCarryable* adatbázis felépítésének is megfelel. Lényegi különbség az, hogy a *kliens* eszköznek csak saját adatait és az *espTouch* által közös adatait kell eltárolnia.

Ezekből kiindulva a következő adatbázist hasonlóképpen tervezem felépíteni, mint az *espTouch*

esetében:

- *measuring*: *TemperatureMeasuring* típusú. Az eszköz adatait fogja eltárolni.
    - Változói: eszköz név, eszköz azonosító, eszköz fűtéskör azonosítója, szenzor által mért adatok (hő, páratartalom).
- *progs*: *programs* objektum típusú, amely a fűtésrendszernek a fontos adatait fogja eltárolni.
    - változói: kívánt hőmérséklet, aktív fűtésprogram indexe, és a fűtéskör fűtés állapotát jelző *bool* változó.
- *Clock:* idő és időzítésért felelős objektum változója.
    - változói: idő, mint óra, perc.
- *wifi*: A Router, és *espTouch* webszerver kapcsolódásához szükséges adatok tárolása.
    - változói: SSID, jelszó, *szerver* IP címe, kommunikációs port.

Az adatbázis objektumainak funkció az adott belső változó elérésére, vagy annak beállítására szolgál.

Az *espCarryable* szerepe a rendszerben az adatszolgáltatás a *szervernek*, illetve a felhasználó szempontjából a rendszer monitorozása. Ezen kívül minimális változásokat tud a rendszer működésében véghez vinni a felhasználó beavatkozásaival. Minimális változást jelent a kívánt hőmérséklet, és aktív fűtésprogram változtatása, amit nyomógombok segítségével lehet elérni. Ezen funkciók segítségével lehet a felhasználó kényelmét növelni, valamint a rendszer komplexitását és hatékonyságát is.

Az eszköz változást is létre tud hozni a saját rendszerében. Az ilyen változókhoz tartoznak flag-ek, amelyek biztosítani fogják, hogy akkor küldje a főszerver a változást, ha a flag-ek azt jelzik.

5.1.3. EEPROM szerepe az adatbázis felépítésében

Az EEPROM a *flash* memóriához hasonlóan tápfeszültség nélkül is képes megőrizni a regiszterekbe írt biteket. Az ESP32 csak *flash* memóriával rendelkezik, amely *flash*-eléskor négy partícióra szerveződik. Ezek a partíciók egyenként felelősek a Bluetooth és WIFI működtetéséért, valamint a feltöltött program tárolásáért. Az utolsó partíció viszont lehetőséget biztosít arra, hogy EEPROM memória típust emuláljak.

A programban szereplő változóknak típusai vannak, ezek a típusok az egyes elemek megkülönböztetésére szolgálnak. Minden típus változóinak értéke bit szinten van eltárolva. Minden változónak van egy memória címe, ami az adott típus változójának értékére fog mutatni. Mivel minden változó előre meghatározott méretű, így elegendő az első bájtjának helyét ismerni a

memóriában.

Automatikus indításhoz, és megfelelő rendszerműködtetéshez a következő változók szükségesek:

- modbus konfigurációs változók:
  - *Slave* ID, regiszter címe és száma, ezek *unsigned int* típusúak.
- Fűtéskörök létrehozásáért felelős változók, amik szintén *unsigned int* típusúak.
  - fűtéskörök száma, és szenzor helye a fűtéskörökön. (*espTouch*)
  - egyedi azonosító, és eszköz helye a fűtéskörön. (*espCarryable*)
- WIFI konfigurációs adatok:
  - SSID, jelszó, *szerver* IP címe az *espCarryable*-nél, ezek karakterenként vannak eltárolva, amelynek típusa *char.*
  - kommunikációs port szám, amely *unsigned int* típusú.
  - WIFI kommunikációt engedélyező változó, amely *bool* típusú.
- Fűtésprogramok, amelyekből összesen 5 darab lesz. Egy fűtésterv 24 elemből áll, amely valós számokat tartalmaz, így az elemek egy-egy *double* változóban lesznek eltárolva. (Ez csak az *espTouch*-nál lesz eltárolva.)
- Az órát és percet *unsigned int*-ben fogom tárolni.
- A hőmérő által mért hőmérséklet korrigáló *offset* -ét *double*-ben fogom tárolni.
- Az eszköz elnevezését pedig karakterenként (*char* típus) fogom eltárolni.

A gyors memória olvasáshoz érdemes a változókat szorosan egymás mellé elhelyezni úgy, hogy az elemek között ne legyen szünet, azaz nem használt memória. Továbbá érdemes típusonként a regiszterekbe írni az adatokat úgy, hogy a legnagyobb helyet igénylő típussal kezdődjön a memória terület írása/olvasása.

4. táblázat. Típusok méretei bájtban

Típus	Memóriában foglalt terület
unsigned int	4 bájt
double	8 bájt
char	1 bájt
bool	1 bájt

A *4. táblázatban* szereplő adatok alapján a következő sorrendben kell eltárolni a memóriában a változókat.

***espTouch* esetében:**

- Fűtésprogramok, és szenzor offset: 0-967 bájtig.
- Port szám, óra, perc, aktív program index, szenzor elhelyezkedése, fűtéskör azonosító, modbus konfigurációhoz szükséges változók: 968-989- bájtig fog elhelyezkedni.
- *char* típusú változók (SSID, jelszó, eszköz neve): 990- 1056 bájtig
    - SSID, jelszó*:* 21 bájton lesz ábrázolva.
    - IP címnek 16 bájt szükséges.

      Eszköz neve legfeljebb 8 karakterből állhat, és a lezáró nulla (’\0’) miatt szükséges még egy, így 9 bájton lesz eltárolva.
- *bool* típusú változó a WIFI engedélyezéséhez pedig az utolsó helyen: 1057-s bájton lesz eltárolva.

Így összesen 1058 bájtra lesz szükségem az EEPROM emulációt ellátó partícióból, aminek mérete *flash*-eléskor beállítható. Az Arduino IDE-ben vannak előre definiált beállítások ezen partíciókhoz, hogy minden megfelelően működjön a program feltöltése után. Mindkét eszköz esetében a szabad partíció mérete 190 KB, ami azt jelenti, hogy az általam használt memória nagyság elenyészően kicsi, így elfér a memóriában.

Az *espCarryable* esetében a sorrend nem fog változni az EEPROM-ban elfoglalt helyek tekintetében, viszont a fűtésprogramot ez az eszköz nem fogja eltárolni. Így 968 bájttal kevesebb helyet kell foglalni az eszköz használata során, amely azt jelenti, hogy 90 bájt elegendő lesz az adatok tárolására.

Az EEPROM memória kezelése a programban

A programban ezen memória területhez több helyen is hozzá kell majd férnem, így egy *EEPROMHandler* könyvtárat hoztam létre. A könyvtár tartalmazza az *espTouch*, és *espCarryable* memóriában elhelyezni kívánt adatok kezdőcímét, amelyek előre definiált címek.

Az *Init task* és a *Serial task* fogja használni ezen könyvtárat. Úgy kell kialakítanom a könyvtárat, hogy az *Init task* típusosan kapja meg a beolvasott adatokat, míg a *serial task*-nak egységesen kell tudnia kezelni minden egyes függvényt. A *serial task* csak írni fogja az EEPROM memória területét, így visszatérési értékként egy *bool* típust választok az EEPROM-t író függvényeknek, ami megfogja mutatni, hogy sikeres volt-e az írás vagy sem. Ezen funkciók paraméterlistája egy változót fognak tartalmazni, amely *void** típusú lesz. Ez a típus le tud kezelni bármilyen beérkező adatot típustól függetlenül, illetve a függvények megírásakor vissza kell konvertálni a beérkező adatot annak eredeti típusára. Ez komplex programírást tesz lehetővé, amit *5.2.3. Eszköz konfigurálása* című alfejezetben fogok megmagyarázni.

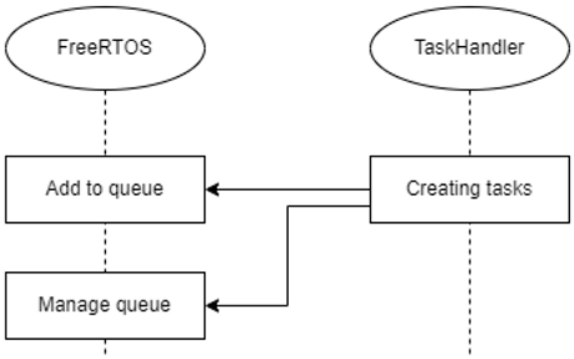
5.2. *espTouch* szoftvere

A *4.2. Hálózati modulok főszekvenciáinak tervezése* című alfejezetben megtervezett folyamat alapján implementálom az egyes feladatokat, és ezen egységek részletes működéséről lesz szó a következőkben.

A program bonyolultsága végett folyamatábrák helyett szekvenciadiagramokat készítettem, amelyek a program felépítését és működésének megértetését célozzák meg. Az ábrákon szereplő funkciónevek nem egyeznek a program funkcióinak neveivel, mivel az ábrákon szerepelő elnevezések több, nagyobb folyamatot/folyamatokat is magába foglal majd. Ezen ábrák a program megértését segítik elő, és a tényleges program külön mellékletként lesz csatolva a dolgozathoz, amely nem a *Mellékletek* című részben lesz megtalálható, hanem az *Összefoglaló* című részben szereplő linken.

5.2.1. Feladatkezelő

A *4.2. Hálózati modulok főszekvenciájának tervezése* című fejezetben leírt *system flow*-t fogja megvalósítani a feladatkezelő, azaz a *TaskHandler.*



41.ábra. Feladatkezelő szekevenciája

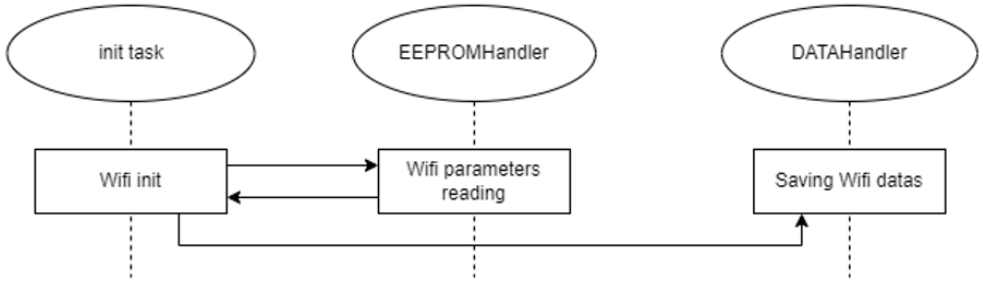
A *41. ábrán* látható a feladatkezelő működése. A feladatkezelőt a *TaskHandler* objektum valósítja meg. Feladata, hogy a *28. ábrán* látható szekvenciát megvalósítsa. Az ESP32 két maggal rendelkezik, így a feladatokat egyesével létre kell hozni, illetve a létrehozott feladatok egy queue-ba fognak kerülni, ahol FreeRTOS operációs rendszer lesz a felelős az egyes feladatok indításáért. Egy queue lesz rendelve az 1. processzormaghoz, hogy lekezelje az *init task*, *main task*, és *serial task* feladatokat. Lehetőséget biztosít a FreeRTOS arra, hogy egy futó alkalmazásból egy másik processzormagon a feladattól független alkalmazást indítsak. Így a *main task*-ból fog a *wifi task* indulni, illetve a *main task* fogja leállítani ezt a feladatot, ha a felhasználó konfigurálni szeretné az *espTouch* eszközét.

Az implementált feladatokat a *Creating tasks* függvény átadja prioritás helyesen az operációs rendszerben létrehozott queue struktúrának (*Add to queue*). Az átadás után megtörténik a feladatok futtatása. (*Manage queue*)

A FreeRTOS egy ingyenes és nyílt forráskódú RealTime operációs rendszer, amelyet a Real Time Engineers Ltd. fejlesztett ki. Tervezése úgy lett kialakítva, hogy nagyon kis beágyazott rendszereken is elférjen, és minimalista funkciókat valósít meg. Egyszerű feladat- és memóriakezelést valósít meg. Nem biztosítja a hálózati kommunikáció működését, illetve külső hardverekhez, meghajtókhoz, vagy fájlrendszerhez való hozzáférést sem. Jellemzői közé tartoznak azonban a következő tulajdonságok: preemptív feladatok, 23 mikrokontroller-architektúra támogatása a fejlesztők által, kis helyigény (4,3 kByte egy ARM7-en a fordítás után), C nyelven íródott és különböző C fordítóprogramokkal fordítható, mint például borland C++ programfordítóval. Ezen kívül korlátlan számú feladat egyidejű futtatását teszi lehetővé, és nem korlátozza ezen feladatok prioritásait, amennyiben a használt hardver ezt lehetővé teszi. Végül sorokat, bináris és számoló szemafor-okat és mutex-eket valósít meg.[16]

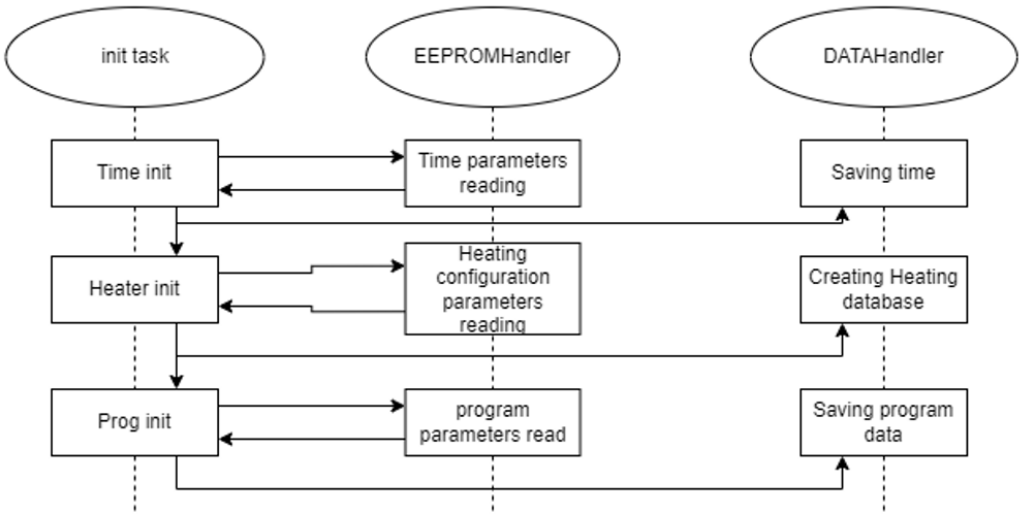
5.2.2. Inicializáló feladat bemutatása

Az inicializáló feladat (*Init task*) két fő funkcióra lett megalkotva, amelyek elősegítik az eszköz helyes és megbízható működését. A feladatban szereplő *upload data* beolvassa az EEPROM-ból az adatokat és menti ezen elemeket az adatbázisba (*DataHandler*). (*29.ábra*)



42.ábra. A WIFI adatok beolvasása és mentése

A *42. ábrán* látható folyamat bemutatja, hogy hogyan működik a WIFI-s (SSID, jelszó, port szám, engedélyező flag) adatok beolvasása. Az *Init task main* függvényéből hívom a *Wifi init* funkciót, amely az *EEPROMHandler* segítségével beolvassa az adatokat egyesével egy lokális változóba. Ezen lokális változókat az üres adatbázisba elmentem, és a *wifi task*-ban felhasználom őket, ha az engedélyező flag értéke *true*, azaz igaz. Ha igaz, az azt jelenti, hogy a *wifi task* csatlakozhat a hálózati routerhez és működtetheti az eszköz webszerverét.



43.ábra. Offline adatok beolvasása

A *43. ábrán* látható folyamat bemutatja, hogy hogyan történik a többi adat beolvasása. Az adatbeolvasásokat az *Init task main* függvényében hívom meg egymás után.

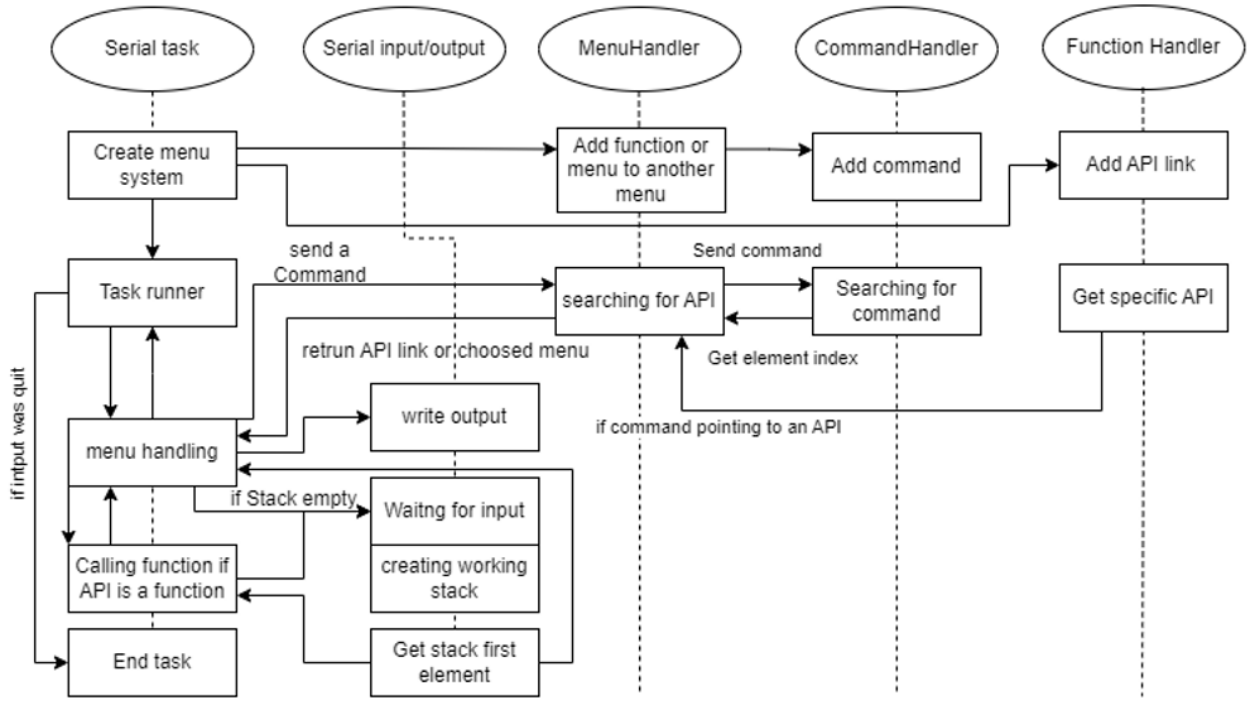
Time init függvény beolvassa és elmenti a memóriában tárolt időt.

A *Prog init* funkció elmenti az EEPROM-ban tárolt kívánt hőmérsékletet, utolsó aktív programot, és minden fűtésprogram értékét. Utóbbit fűtésprogramonként olvassa be egy átmeneti tömbbe, amit a beolvasás végén kiment az adatbázisba a *Saving program data* függvény segítségével. Mivel öt darab fűtésprogram van létrehozva, így az átmeneti tömböt ötször használom fel, és minden egyes alkalommal csak felülírom a változó elemeinek értékeit.

A *Heater init* függvény elmenti a modbus kommunikációhoz szükséges adatokat az adatbázisba, valamint beolvassa a memóriából a fűtéskörök számát, az *espTouch* helyének számát, szenzor offset-t, és a fűtésmódot is. A beolvasás után felépítem az adatbázis dinamikus részét, amely *Creating Heating database* függvénnyel lehetséges. A funkció létrehoz egy fűtéskör típusú (*HeatingCircleHandler*) dinamikus tömböt a beolvasott szám alapján. A *HeatingHandler*-ben eltároljuk a létrehozott tömböt, és a tömb megfelelő elemében eltárolom az *espTouch* adatait, köztük a mérési adatokat is. A megfelelő elem a beolvasott helynek száma lesz. Ennek a számnak kisebbnek kell lennie, mint a fűtéskör tömb elemszámának. Ha nagyobb lenne, akkor az adatok nem lennének eltárolva, és az *espTouch* nem lenne részese a fűtéskörök hőmérséklet szabályozásának.

5.2.3. Eszköz konfigurálása

A *4.2.4. Eszköz beállításáért felelős feladat* című alfejezetben leírtak alapján fogom implementálni a feladatot. A következőkben a *32. ábra Serial communication* elemről lesz szó.



44.ábra. *Serial task* működési szekvenciája

44. ábrán a *serial task* működési szekvenciája látható, amely több különböző segédprogrammal rendelkezik. Ezen objektumok és funkcióik egy keretprogramot biztosítanak, hogy a tényleges feladatot csak a *serial task*-ban kelljen létrehozni. A keretprogram, vagy middleware az *EEPROMHandler*-t köti össze a *serial task* és a *Serial input/output*-tal. Ez a middleware maga a menürendszer, amelyet *a MenuHandler, CommandHandler, és Function Handler* alkot.

A *Function Handler* egy olyan objektum, amelyben el lehet tárolni egy függvény pointert, aminek visszatérési értéke *bool*, és paraméterlistájára egy darab *void** típusú változó érkezhet. Ezzel az objektumtípussal elérhetőségi utat biztosítok egy specifikus függvényhez, amit egy parancsszóval szeretnék a program futása során elérni és futtatni.

CommandHandler típusú objektum képes eltárolni több karakterláncot (*String*-ket) egy tömbben. Ebben a tömbben egy kívülről érkező parancsszó segítségével lehet megtalálni a tömb adott elemét. Ha van találat, akkor a függvény hívásának helyére visszaadja a találat helyének számát, azaz a tömb elemének az indexét, ellenkező esetben egy hiba kódot, amely egy 0-nál kisebb szám.

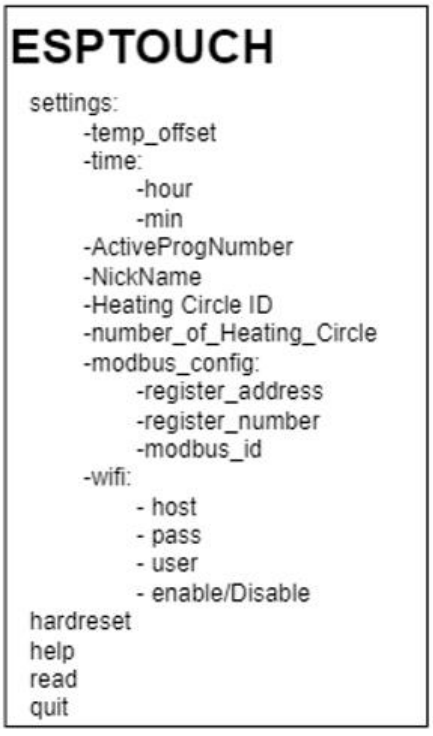
A *MenuHandler* objektum magát a menürendszert alkotja. Egy ilyen objektumnak lehet *Function Handler* tömbje, illetve *MenuHandler* tömbje. Ezekhez a tömbökhöz tartozik egy-egy *CommandHandler* változó, amely a menühöz hozzáadott elemek parancsszavait tartalmazza. Mivel tömbökről van szó, így a hozzáadott API-k és menük sorrendjének meg kell egyeznie a parancsszavak sorrendjével. Ezt úgy értem el, hogy amikor hozzáadok egy elemet az adott menühöz, akkor azzal egyidőben egy parancsszót is hozzáadok az adott *CommandHandler* változóhoz. (például: *bool*

*addSideMenus(String CallName, MenuHandler*menuElem);*)

Egy *MenuHandler* objektum a program működése során két *switch case* szerkezetet használ. Az egyik felel a parancsszavak megtalálásáért. Ez a szerkezet a találat alapján eldönti, hogy egy almenüt, egy funkciót (API-t), vagy egy olyan parancsszót talált, amely az aktuális menü elemhez tartozik, például: „help”, „cancel”, „quit”. A másik szekvencia kezeli az előbbit, és annak eredménye alapján visszatér a hívás helyére egy menüre, vagy egy API-ra mutató funkció pointerrel. Ha egyikkel sem, akkor vagy egy az aktuális elem funkciójával tér vissza, vagy azzal, hogy vége a *serial task*-nak, vagy pedig az ős menübe lépjen vissza a program.

A *Serial input/output* objektum kezeli a soros porton érkező adatokat. Minden beérkező adatcsomagot egy karakterláncba ment el. Az elmentés után szóközönként feldarabolja a láncot, és egy *String* tömbbe menti el azokat, ez lesz a *Stack*, amiből a *serial task* dolgozni fog. Ha létezik a *Stack*, akkor a *Get stack first element* függvénnyel a *Stack* első elemét ki lehet kérni, amely ezután ki fog törlődni, és a következő elem kerül az első helyre. Továbbá lehetőséget nyújt arra is, hogy a felhasználó az általa küldött adatokról, és a programban történtő változásokról értesüljön, például: sikeres SSID mentés után a kimeneten láthatóvá válik a mentett adat, és hogy sikeres volt-e a mentés („Successfully saved”).

A *44. ábrán* az eddig említett objektumok láthatóak és a *serial task*, illetve ezek összefüggő működése. Az *espTouch* esetében egy komplex menürendszert fogok alkotni.



45.ábra. *espTouch* menürendszer terve

A *45. ábrán* látható a terve az *espTouch* menürendszerének, amely 3 almenüt, és 18 API-t tartalmaz. Az *5.1. Adatbázis felépítése* című fejezet tartalmazza a képen látható elemek leírását, hogy mik az EEPROM-ban tárolt adatok, és miért van rájuk szükség.

A *Serial task Create menu system* függvényében létrehozom az egyes menüket, mint változó, és ezen elemekhez hozzárendelem az egyes API-kat, például létrehozok egy *Menuhandler* változót és hozzáadok egy API-t, amelynek a neve *„host”* lesz. Ezt a parancsszót az *Add Command* függvénnyel lehet hozzáadni a változóhoz, hogy az API elérhetővé váljon. Ekkor fel fogom használni az *Add function or menu to another menu* funkciót, amely hozzáadja a *Menuhandler* változóhoz a funkció parancsszavát, és a *Function Handler* linket, amin keresztül el lehet érni az API-t. Ha ez megtörtént, akkor az üres linknek megadom, hogy milyen API-t hívjon meg, ha a program futása során a *„host”* parancs érkezik. Létrehozhatok egy újabb *MenuHandler* változót, ami a főmenü lesz. A főmenünek lehetnek ugyanúgy API-jai és *MenuHandler* elemei is, mint egy átlagos *MenuHandler* változónak. Az elsőnek létrehozott menüt ugyanúgy hozzá tudom adni a főmenühöz vagy bármely más menühöz, mint a *„host”* paranccsal elérhető API-t. Ugyanúgy meghívásra kerül az *Add function or menu to another menu*, amely argumentumában átadom a menüt, mint pointer, és a parancsszót, amivel el lehet majd érni a program futása során az adott menüt.

A kész menürendszert a *Task runner* fogja működtetni, amely a *menu handling* visszatérési értéke alapján fogja eldönteni, hogy a *Serial task*-t be kell-e fejezni, vagy a *root* menü elemet kell-e betölteni. A *menu handling* függvény fogja kezelni a Serial input/output-t. Ha a *Stack* üres, akkor a felhasználótól egy bemeneti adatcsomagot vár. Ha nem üres, akkor a program *Stack*-ből ki veszi az első elemet, és a *MenuHandler* változó belső funkciójával (*searching for API*) megkeresi a következő *MenuHandler* változót, vagy funkció linket (API-t). Keresés során a belső *switch case* megkeresi a *CommandHandler* változókban a beérkező parancsszót. Ha találat van, akkor a keresés befejeződött, és a külső *switch case* el fogja dönteni a visszatérési értékből a saját kimenetét. Ha API-t talált a program, akkor a *Get specific API* függvénnyel lekérdezi a szekvencia a funkció linket és visszaadja azt a *menu handling*-nek.

Egy parancsszó többféle is lehet:

- *MenuHandler* változó belső parancsszavai:
  - „quit”, „help”, „cancel”.
- *MenuHandler* menüihez tartozó parancsszavak.
- *MenuHandler* API-ikhoz tartozó parancsszavak.

Ha „quit” parancs érkezett a *searching for API* függvénybe, akkor a *Serial task* véget ér és az *espTouch* újraindítja magát.

Ha „help” parancsszót észlel a program, akkor az aktuális *MenuHandler* változó parancsszavait

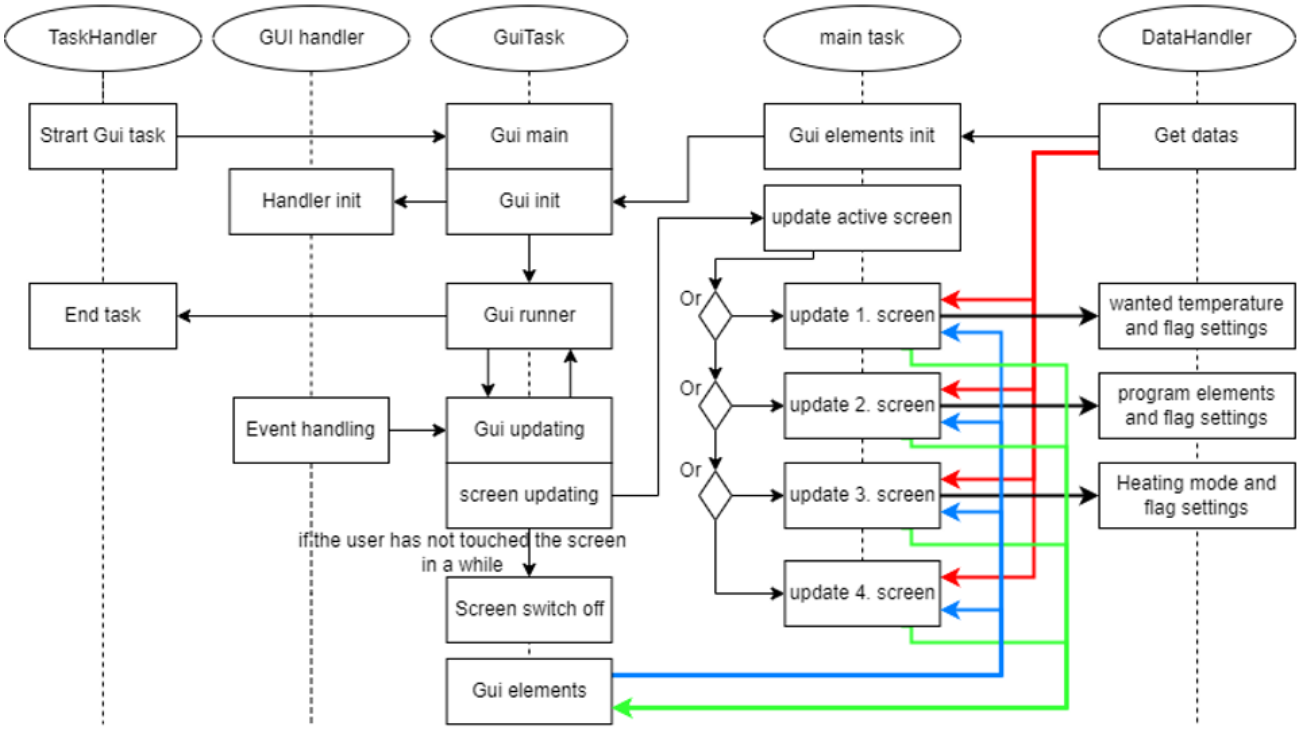
fogja kilistázni a kimenetre. A *menu handling* funkció a *serial input/output* segítségével (*write output*) tud a kimenetre adatcsomagot küldeni, hogy a felhasználó lássa parancsszavainak következményeit.

Ha „cancel” parancsszó érkezik, akkor *root* menüt fogja betölteni a program.

Ha nincs találat, akkor az adott menüpontban marad a program, és várja az input-t. Ellenkező esetben tovább lép a program a megfelelő funkció felé. Ha a találat egy API hívásnak felel meg, akkor a függvény lefutása után a *root* menü fog betöltődni függetlenül attól, hogy a lefutott függvény visszatérési értéke igaz vagy hamis volt.

5.2.4. Adatbázis és a GUI összehangolása

Az adatbázis és GUI (grafikus felhasználói felület) összehangolásának implementálása a *4.2.2. Főfeladat* című alfejezetben megtervezettek alapján készítettem el. Ebben a fejezetben be fogom mutatni, hogyan működik az *espTouch* felhasználói oldala.



46.ábra. *Gui task* és a *main task* szekvenciája

A *46. ábra* bemutatja a program működését, amely a felhasználói felületet fogja frissíteni a létrejövő interakciók alapján. Az interakciók a felhasználó beavatkozásai alapján jönnek létre. Ezek a beavatkozások végrehajthatók a grafikus felhasználó interfészen, vagy az *espCarryable*-n keresztül.

A megtervezett design *4.3. espTouch periféria felhasználói interakcióinak tervezése* című fejezetben olvasható. A design a *SquareLine Studio*-ban lett megtervezve, és ez a program generálta le a design kódját, amely elemeit össze kell kapcsolni az adatbázis adataival. A generált kód *lvgl (GUI*

handler) könyvtár szerkezetet használ azért, hogy a design megfelelő külalakot kapjon, illetve az interaktív elemeket eseményvezérelten lehessen kezelni.

Megfelelő adatokat az egyes funkcióknak a *DataHandler* fogja szolgáltatni, illetve ezen keresztül lehet az adatbázis adatait változtatni, ha felhasználói interakció történik a rendszerben.

A *GuiTask* feladata futtatni az *lvgl* által kínált eseményvezérlő programot, amely a design elemeit kezelni fogja (*Event handling*). Ezek az elemek *label*-k, gombok, csúsztatható felületek (*slider, ARC*), enumerációt végrehajtó elemek (*roller*-k) és lenyíló fül, ami szintén enumerációt hajt végre. Ezen elemekhez több eseményt is lehet társítani. A gombok működtetésekor oldalváltást, vagy *label* szövegének a változását idézem elő. Eseményt jelent az is, ha egy *slider*, *ARC*, vagy *roller* változik. Ezek a változások a rezisztív-interfészen keresztül történnek érintés útján. Az érintés iránya és folyamata alapján az *lvgl* meghatározza, hogy milyen beavatkozás történt. A meghatározott beavatkozás alapján frissíti a kijelzőn ábrázoltakat, és az ábrázolt elemek belső változóinak értékeit. A *main task* ezeket a változásokat figyeli, és ezek alapján frissíti az adatbázis elemeit. Az *espCarryable* is hatással van az adatbázisra. Ha a rendszerben nem a GUI általi beavatkozás történik, hanem belső változás általi, azaz program általi, vagy az *espCarryable* általi, akkor a *main task* az adatbázisban található új adatokkal fogja a GUI-t frissíteni. (*Gui updating*)

A *GuiTask* felépíti a GUI-t, és hozzárendeli az *lvgl* könyvtárhoz az általam implementált érintés érzékelést (*Handler init)*. A *Gui init* funkció minden oldal változójának értékének kezdő értékét fogja beállítani az adatbázisban található változók értékeinek segítségével (*Gui elements init*).

A *srceen updating* funkció felel a *main task* futásáért. Ebből a funkcióból érhető el a *screen switch off*, aminek feladat az, hogy ha a felhasználó nem érinti meg a kijelzőt 1 percig, akkor a kijelző kikapcsol addig, amíg azt újra meg nem érintik.

A *main task* egy middleware lesz az adatbázis és a design futásáért felelős *GuiTask* objektum között. Feladata, hogy mindig a kijelzőn megjelenő ablakot frissítse a felhasználói beavatkozások alapján, vagy a rendszerben fellépő változások hatására.

A *46. ábrán* látható a *main task* főbb feladatai. A desing-hoz tartozik egy *label*, amely minden ablakváltáskor az ablak indexére fogja állítani a *label* szövegét. (például: Főoldalról a *Programs* oldalra váltok, akkor a label ”0” -ról ”1” -re fog váltani) A *screen updating* függvény ezt a *String* értéket numerikus értékké konvertálja és átadja a *main task update active screen* függvénynek. A szám alapján egy *switch case* szerkezet eldönti, hogy melyik ablak frissítő funkciót kell meghívni. A program az *update 1. screen*, *update 2. screen, update 3. screen,* és *update 4. screen* közül választhat.

Az *update 1. screen* a *33. ábrán* látható főoldalt, az *update 2. screen* a *Programs* aloldalt, az *update 3. screen* az *Options* aloldalt, és végül az *update 4. screen* a *Connected Device* aloldalt fogja frissíteni. Ha az oldalakon felhasználói interakció jön létre, akkor az adatbázist a *DataHandler*-ben

található függvényekkel lehet frissíteni, amelyek a *klienseknek* (*espCarryable* vagy böngésző alkalmazás) küldött üzeneteket beállító flag-eket is állítják, hogy az új adatokat küldje az *espTouch* a *kliensek* felé. Ezek a függvények sorra a *wanted temperature and flag settings*, *a program elements and flag settings*, és a *Heating mode and flag settings*.

5.2.5. Kliensek kezelése

Az *espTouch*, mint szerverelem a rendszerben kezeli a csatlakozott *klienseket*, amely lehet *espCarryable*, és webböngésző is. Ebben a fejezetben a *4.2.3. Kommunikációért felelős feladat* című fejezetben olvasható *wifi task*-nak a *wifi server handler* moduljának az implementálását fogom részletezni.

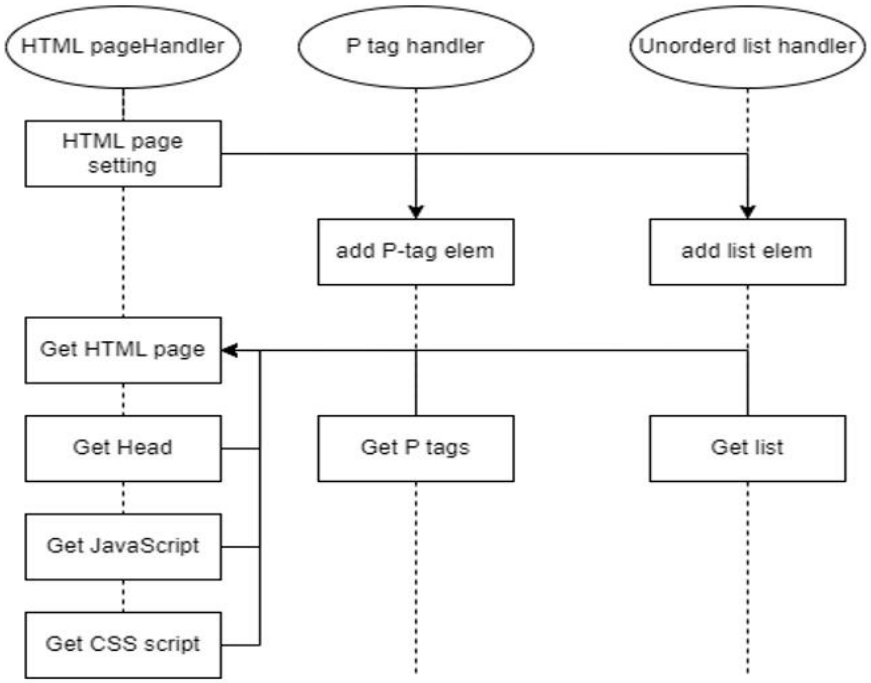
Bármely webszerver működtetéséhez szükség van egy fizikai elemre, ezek lesznek az *espTouch*, és *espCarryable*-k. Továbbá a *szervereknek* egy összetett programot kell futtatnia, hogy a *kliens* webböngészőben megfelelő külalakban jelenjen meg. A megfelelő weboldal design alapját HTML nyelven írt program fogja képezni, és a küllemét a CSS nyelven írt program script fogja adni. Egy weboldal dinamikus működtetéséhez JavaScript-t használok. A HTML programozási nyelv biztosítja a *span* tag-t, amelyhez egy *id* tartozik. Az *id*-nak a program írása során adok értéket, illetve a megírt programom fogja legenerálni a HTML oldalt, és a generálás közben fog minden egyes *id* értéket kapni. Az *id*-k értéke egy-egy karakterlánc.

Amikor egy *kliens* kapcsolódik a *szerverhez*, akkor egy HTTP GET kéréssel a *kliens* lekérdezi a *szerver* által üzemelt weboldalt. Dinamikus weboldalak esetén a cél az, hogy a weboldal változói automatikusan frissüljenek anélkül, hogy újabb HTTP GET kérést küldene a *kliens* a *szervernek*. Erre ad megoldás a Websocket kommunikáció, amit *a 3.2.2. WIFI kommunikáció* című fejezetben írtam le. Websocket kommunikáció során JSON üzenet típusokat használok, amely két karakterláncból áll. Az első karakterlánc a változó neve, míg a második a névhez rendelt érték lesz. Egy üzenetben több adatpárt is lehet küldeni a *kliensnek*.

A JSON üzenetek, és a *span* tag *id*-ának értékei összefüggenek. Az *id* értéke csak hivatkozási pontként szerepel a HTML kódban. A weboldalon ténylegesen megjelenő szöveg a hivatkozáshoz rendelt újabb érték lesz. A HTML oldal változóit, és a JSON üzeneteket a JavaScript program kód fogja összekötni. A kód HTML oldal hivatkozásait összehasonlítja az üzenetben kapott nevekkel, ha egyezést talál, akkor az üzenet neveihez tartozó értéket fogja megadni a HTML *span id* hivatkozásnak. Azaz a JSON üzenet értéke bemásolásra kerül a HTML oldal *id* változóba, amely a weboldalon HTTP GET kérés nélkül meg fog jelenni.

Az ESP32 mikroprocesszort C/C++ nyelven programozom, így a webszervert, és weboldalt is ezen a nyelven fogom megírni. A weboldal működéséhez összesen 4 nyelvre van szükségem, a

JavaScript-re, HTML-re, CSS-re, és C++-ra. Utóbbi nyelven lesz megírva az üzenetküldés és a HTTP GET kérésre küldendő válasz. A válasz tartalmazni fogja HTML programot és a hozzá tartozó script-eket. A script-ek előre megírt programok, amelyeket CSS-ben és JavaScript-ben fogok megírni. Mivel C++ az alapvető nyelv, és a többi csak webböngésző által futatott kód, így lehetőségem van arra, hogy karakterláncként tárolja a script-eket, és a HTML programot.

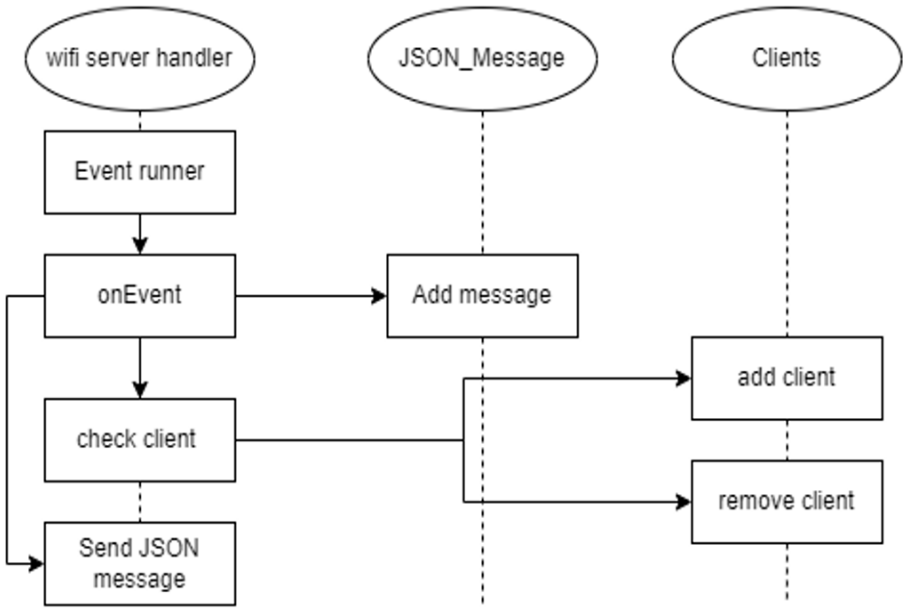


47.ábra. HTML programkód generálása

A HTML programot a script-ekkel ellentétben generálom, ennek a szekvenciája látható a *47. ábrán*. A programban összetett tag-ek fognak szerepelni, amik alapját a *P tag*-ek és az *Unordered list* tag-jei fogják alkotni. A *wifi server handler*-n keresztül a *HTML page setting* függvényben az általam előre meghatározott elemekkel fogja felépíteni a programot. A *HTML pageHandler*-nek van *egy P tag handler* tömbje és egy *Unordered list handler* tömbje. A *HTML page setting* függvénnyel tag-eket lehet hozzáadni a tömbökhöz, az *add P-tag elem*, és az *add list elem* függvények segítségével. A tag-ek létrehozásakor meg kell határozni, hogy milyen *span id*-ja legyen, milyen statikus szövege, és milyen CSS osztályba tartozzon. Utóbbi fog gondoskodni arról, hogy a weboldal betöltése során a szövegek/ábrák a megfelelő helyen és formában jelenjenek meg.

Amikor egy *kliens* csatlakozott a *szerverhez* és HTTP GET kérést küld a *szervernek*, akkor a *szerver* válaszolni fog. Esetemben a válasz küldése előtt egy *ServerHandler* objektum a Get HTML page elnevezés funkcióval le fogja generálni a weboldalt. A weboldal generálása során a program meghívja a *Get Head* függvényt, ahol a CSS script linkje található, amit a *Get CSS script* funkción keresztül ér el. Ezek után a HTML *body* részét kell legenerálni, ami a tag-kből és a JavaScript link-

jéből fog összeállni. Ezeket a *Get P tags, Get list*, és *Get JavaScript* függvények segítségével állítom össze. Az *espTouch* generált kódját a *4. mellékletben* lehet megtalálni.



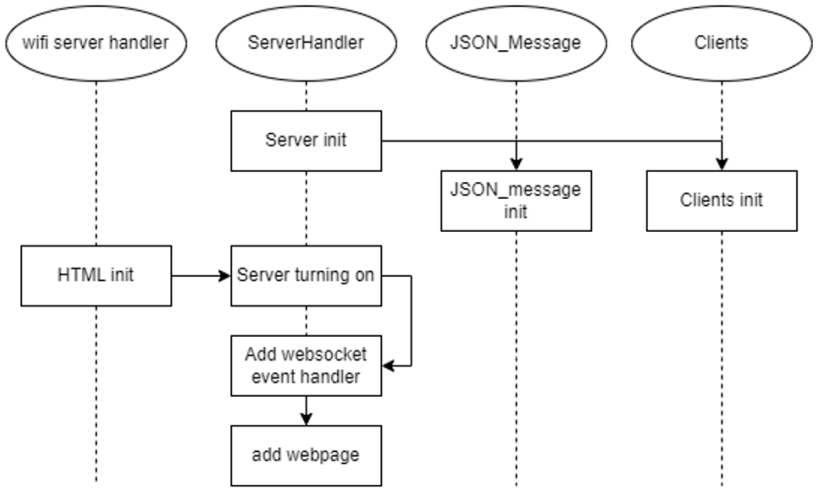
48.ábra. *Kliensek* kezelése

Amikor egy *kliens* csatlakozik, fontos, hogy megkapja a weboldal adatait, és a weboldal változóinak értékeit, illetve azoknak legfrissebb értékeit. A *szerver* és *kliens* között aszinkron Websocket kommunikáció zajlik, amit eseményvezérelten kezelek a programomban. Az *Event runner* függvény lesz az az esemény, ami le fogja kezelni az új *klienseket*, és a beérkező webSocket üzeneteket. Az *onEvent* belső felépítése *switch case* szerkezeten alapszik, amely az előbb leírtak alapján fogja eldönteni, hogy *a check client*, vagy az *Add message* függvény fog meghívódni.

Amikor az *add message* függvény fog futni, akkor a beérkező üzenet el lesz tárolva. Minden egyes *kliensnek* a legfrissebb beérkező üzenete lesz eltárolva a JSON_Message objektumban.

A *check client* függvény pedig el fogja dönteni, hogy új *kliens* érkezett, vagy egy *kliens* lecsatlakozott-e a *szerverről*. Ha új *kliens* érkezik, akkor az *add client* függvény hívódik meg, illetve a másik esetben a *remove client*. Ha HTML oldal értékeinek lekérdezésével kapcsolatos parancs érkezik, akkor a *Send JSON message* funkció fogja elküldeni a *kliens* felé a változók értékeit.

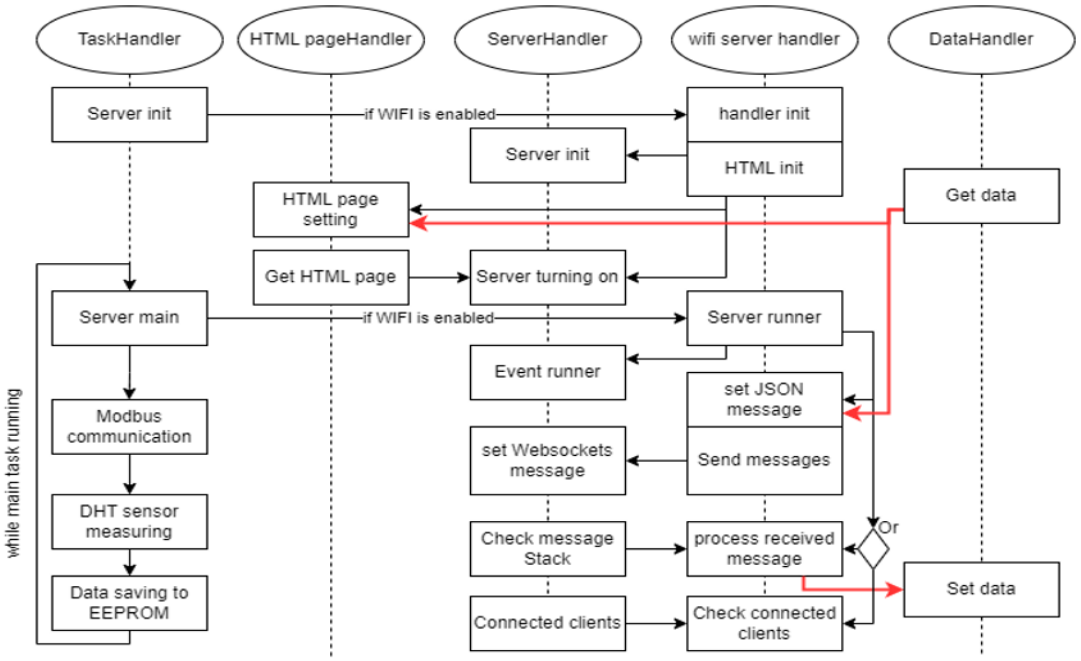
Ha az *add client* függvényt fogja a program meghívni, akkor a függvény eldönti, hogy egy *espCarryable* csatlakozott-e, vagy egy webböngésző. A döntés egyszerű, mivel az *espCarryable* is üzemel egy weboldalt, ami az eszköz IP címén keresztül érhető el, így elegendő egy HTTP GET kérést küldeni az új *kliens* felé. Ha a válasz „*200 OK*”, akkor az *espTouch* tudni fogja, hogy ez egy *espCarryable*. Ha a rendszerelem részét képezi a csatlakozott eszköz, akkor az IP címét hozzáadja a weboldal *Unordered list handler* objektumához. (*48.ábra*)



49.ábra. *Szerver* működése

A *49. ábrán* látható a *szerver* működési szekvenciája. A *Server init* funkció létrehozza a *Clients*, és *JSON_Message* objektumokat, hogy a csatlakozó *kliensek* adatai, és beérkező üzenetek eltárolásra kerüljenek. Ezen objektumok fogják biztosítani a megfelelő működést az *espTouch* és *kliensek* között.

Amikor a HTML oldal elemei beállításra kerülnek, akkor a *Server turning on* funkció segítségével a weboldal legenerálódik, és a *szerver* eseményvezérelt eleméhez hozzá rendeljük a *48. ábrán* definiált eseményláncot. Ha HTTP GET kérés érkezik, akkor a server az eseménykezelőn keresztül egy üzenetet tud küldeni a *kliensnek*, amely a script-eket és a HTML oldalt fogja tartalmazni. Ezen elemeket az *Add websocket event handler* függvény fogja a vezérlőbe beállítani. Mivel a HTML oldal lekérdezése nem Websocket alapú, így az oldalt és a további elemeit külön-külön kell az eseményvezérlőhöz hozzáadni, amit az *add webpage* függvény biztosít.



50.ábra. *Kliensek* kezelése az *espTouch* eszközön

Az eddig leírtak alapján szeretném bemutatni végleges formáját az *espTouch* webszerver működésének, amely az *50. ábrán* látható.

A *TaskHandler* inicializálja (*Server init*) a weboldalt, és a *klienseket* kezelő programrészeket, ha az eszköz konfigurációjában ez megengedett. Az eszköz konfigurálásáról a *5.2.3. Eszköz konfigurálásának implementálása* című alfejezetben lehet olvasni. A *handleinit* függvény fogja meghívni a *HTML init* funkciót, amelyen keresztül a generálandó weboldal lesz beállítva. A beállítást *a HTML page setting* függvényen keresztül lehet elérni. A weboldal működéséről a fejezet elején volt szó, és a generálást biztosító program szekvenciája a *47.ábrán* látható. Amikor a HTML weboldal elemei a megfelelő helyen elmentésre kerültek, akkor a *Server turning on* funkciója lesz meghívva, amely segítségével az *espTouch* kapcsolódik a routerhez, elindítja a *szervert*, és a *szerver* eseményvezérlőnek átadja a megfelelő funkciókat. Utóbbi a *49. ábrához* köthető.

Ha a csatlakozás a routerhez engedélyezett és sikeres, és a *szerver* indítás is, akkor a *Server main* függvény fogja trigger-elni a *Serve runner* függvényt, amely kezelni fogja a *kliensek* kéréseit, és állandóan frissíti a weboldalhoz tartozó változókat, ha azok megváltoznak.

A *set JSON message* funkcióval a program beállítja a *kliensnek* szánt üzenetet, amely weboldal elemeinek az értékeit fogja frissíteni. A funkció az adatváltozásról a *Get data* függvényen keresztül tud az adatbázisból tudomást szerezni, azaz az adatbázis elemeihez tartozó flag-kből. Ha valamely adat változik, akkor a JSON üzenet listára fog bele kerülni az adat, és a hozzá tartozó macro-ként definiált változó neve. Ezek után az adatbázis flag-jei *bool* típusú hamis értéket fognak felvenni, hogy a következő szekvenciában ne küldje el újra ugyanazon adatokat az eseménykezelő (*Event runner*). Ha sikeresen elkészült az üzenet, akkor a *serverHandle*-ben a *setWebsockets message* függvény fogja beállítani az eseménykezelőnek a küldendő üzenetet. Ezt a függvényt a *Send message* funkcióval érjük el a *wifi server handler*-ből.

Ha az *Event runner* beérkező Websocket üzenetet mentett el, akkor le fog futni a *process received message* által meghívott *Check message Stack* funkció. Az itt eltárolt JSON üzeneteket a programom dekódolja, és el fogja menteni a megfelelő helyre az adatbázisba a *Set data* függvényen keresztül. Egy üzenet feldolgozása után az adott üzenet törlődik a rendszerből, de a *JSON_Message* objektum továbbra is elérhető lesz, ha újabb üzeneteket kell az *Event runner*-nek elmentenie.

Ha az *Event runner* egy *kliens* lekapcsolódásának eseményét végzi el, akkor a *Clients* objektumban egy flag jelezni fogja az adott *kliens* elemében, hogy az eszköz már nem elérhető. A *Check connected* függvény fogja ellenőrizni a *Clients* objektum elemeit. A *Connected clients* függvény mindig egy *kliens* flag-jét adja vissza, hogy az elemhez tartozó eszköz csatlakozva van-e vagy sem. Ha a flag hamis, akkor az eszköz lecsatlakozott, és a *Check connected clients* függvény kitörli az elemet az adatbázisból, a *Clients* objektumból, és az *Unordered list* objektumból is. Az

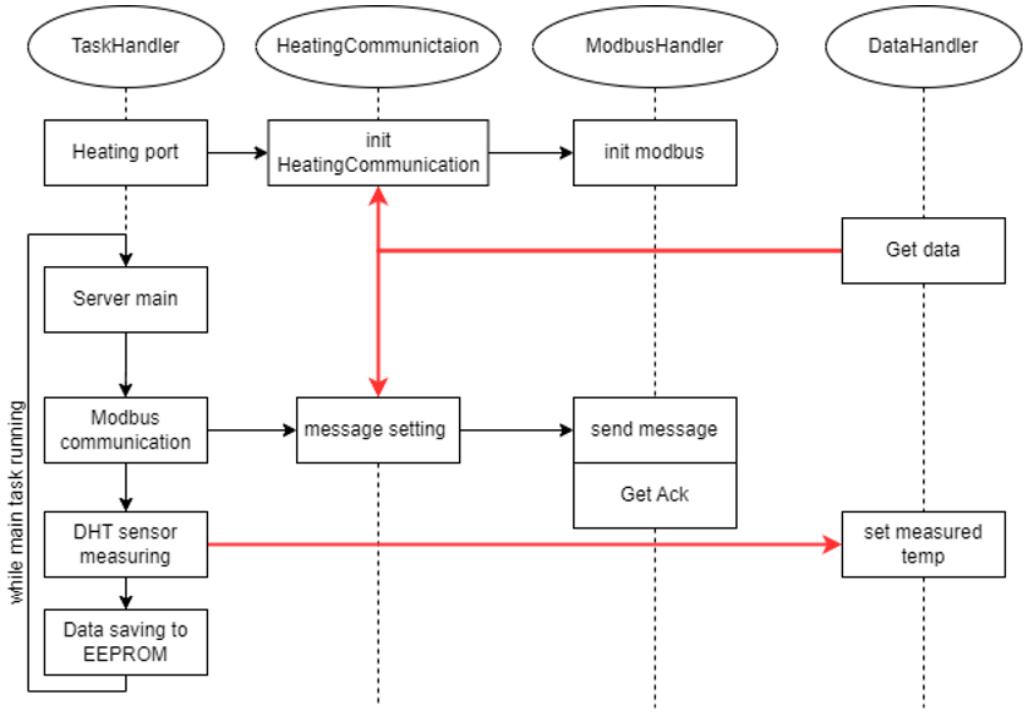
adatbázisban kitörlendő elem a *Connected Device* oldalon megjelenő design elem, amely a *Heathandler* objektumának *HeatingCircleHandler* objektumtömbjében helyezkedik el.

A feladat elvégzése sok időbe telik a processzormagnak, ezáltal a *process receive message* és a *check connected clients* függvényei váltakozva fognak meghívódni, hogy csökkentsem a feladat futásidejét.

Ha a *Server main* feladat véget ért, akkor a többi kisebb feladat fog lefutni, amit a következő alfejezetben fogok ismertetni.

5.2.6. Egyéb feladatok

A *wifi task*-nak több feladata is van, amikről a *4.2.3. Kommunikációért felelős feladat* című alfejezetben lehet olvasni. Az előző alfejezetben fejtettem ki a *kliensek* kezelését, de a *wifi task*-nak további alfeladatai vannak, amelyek a szenzor kommunikációért, modbus kommunikációért, és az EEPROM-ba történő adatok mentéséért felelnek. A következőkben ezeket az alfeladatokat fogom taglalni.



51.ábra. *Heating controller communication handler* szekvenciája

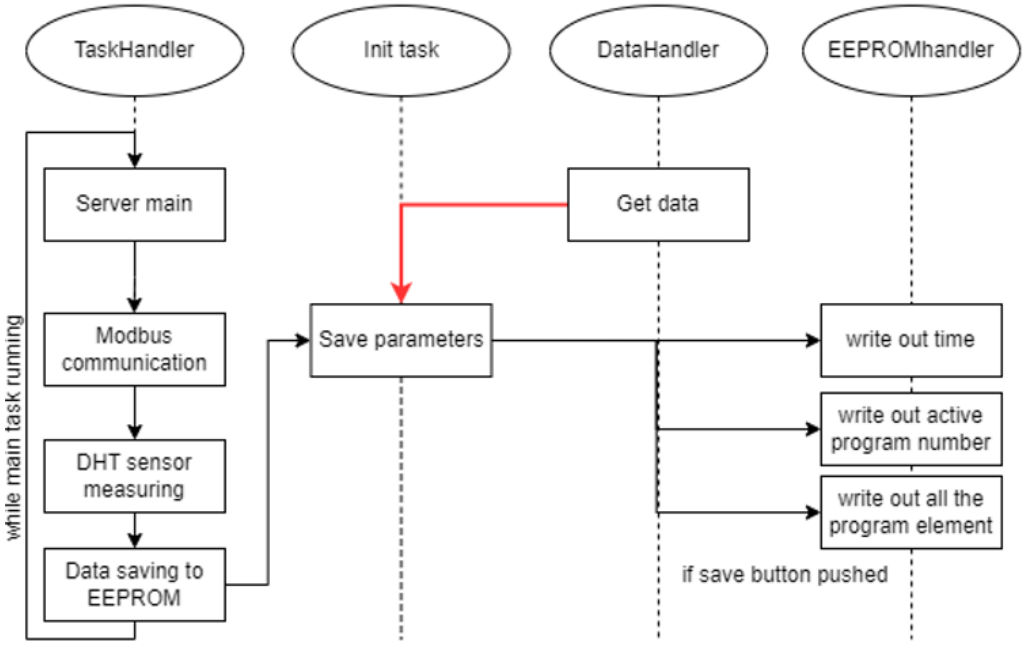
A *wifi task*-ban a *Heating controller communication handler* kulcsfontosságú, mivel az *espTouch* ezen az alfeladaton keresztül fog a *Controller for heating system*-mel kommunikálni a fűtés elindításáért adott fűtéskörön, illetve a fűtés módjáról. Az *51. ábrán* a *wifi task*-ból kiinduló alfeladat szekvenciája látható.

A *Heating port* funkció fogja a modbus protokollt konfigurálni az *init HeatingCommunication*

és az *init modbus* funkciók segítségével, hogy az *espTouch* RS-485 busz hálózaton keresztül tudjon a fűtésvezérlő egységgel kommunikálni (*Controller for heating system*). A konfigurációs adatokat az adatbázisból fogja lekérdezni a *Get data* funkció segítségével. Ezen adatokat a felhasználó előre be tudja állítani a *serial task* segítségével.

A *wifi task* ismétlődő ciklusában a *Modbus communication* funkció meghívja a *HeatingCommunication* objektum *message setting* függvényét, amely beállítja a küldendő üzenetet a *Slave* felé. Az üzenet tartalmazza a fűtéskör indexét, állapotát, és a fűtés típusának állapotát is. Az üzenetet 8 bites hasznos adatcsomagban küldöm a *Slave* felé, így az indexet 4 bittel ábrázolom, míg az állapotot és típust 1-1 bittel. A fennmaradó két bitet szóközként használom az adatok között, hogy a *Controller for heating system* egyszerű módon tudja a beérkező üzenet tartalmát feldolgozni. Üzenetküldésre a *send message* függvényt használom, míg a nyugta fogadására a *Get Ack*-t. Az *espTouch* csak is akkor küld üzenetet a *Slave*-nek, ha valamelyik adat megváltozik.

A *wifi task* alfeladatai közé sorolható továbbá az *espTouch* DHT hőmérséklet és páratatalom mérő szenzorával végzett szenzor kommunikáció a One-Wire buszon keresztül. A szenzor által mért hő és páratartalom az adatbázisban kerül elmentésre a *set measured temp* függvényen keresztül. A mentés helye az a fűtéskör elem lesz, ahol az *espTouch* adatai el vannak tárolva. Ezt az adatot a *serial task*-n keresztül a felhasználó előre beállíthatja.



52.ábra. A save alfeladat szekvenciája

Az *52. ábrán* a *wifi task Data saving to EEPROM* alfeladatának szekvenciája látható. A feladat az időt és az aktív program számát minden egyes iterációban elfogja menteni, ha azok megváltoztak.

A *4.3.2. A Programs* aloldal című alfejezetben látható desing tartalmaz egy *Save* gombot, amely

lenyomásával a *save* flag kerül igaz állapotba. Ha a *save* flag igaz lesz a *Save parameters* függvényben, akkor az összes fűtéstervhez tartozó elem is kiírásra kerül az EEPROM-ba. Ha a fűtésterv megváltozik és a *Save* gomb nem kerül lenyomásra, akkor a változás nem lesz elmentve. Ha nem lett elmentve a változás és az eszköz külső okok miatt újraindításra kényszerül, akkor a beállítások elvesznek, és az utoljára mentett fűtéstervek fognak az *espTouch*-n betöltődni.

5.3. *espCarryable* szoftvere

Az *espCarryable* kiegészítőelemként jelenik meg a hálózatban. Feladata, hogy mérje a hőmérsékletet, és azt közölje az *espTouch*-csal.

Az eszköz egy fűtéskörhöz tartozhat, amit a konfiguráció (*serial task*) során lehet beállítani. A *serial task* implementálást az *5.2.3. Eszköz konfigurálása* című alfejezetben ismertettem. A feladat kialakításánál fontos tényező volt az, hogy a megírt kód univerzális legyen, és egy inicializáló függvényen keresztül lehessen a használni kívánt menürendszert specifikálni.

Az *EEPROMHandler* függvényei is hasonló univerzitást élveznek. A felhasznált adatblokk nagysága a memóriában teljesen más, mint az *espTouch*-é, amelyet az *5.1.3. EEPROM szerepe az adatbázis felépítésében* című alfejezetben leírtam. A két eszköz adatai a memóriában hasonló sorban helyezkednek el, egymáshoz szorosan illeszkedve. Az univerzitást macro-k biztosítják, amelyek az adatok kezdőcímét adják meg a macro hívásának helyénél. Továbbá két macro-t használok arra is, hogy elkülönítsem az *espTouch*, és *espCarryable* macro-jait is. Ezzel az elkülönítéssel egy macro nevet kétszer is definiálhatok más-más értékekkel. Az *EEPROMHandler*-ben mindig csak is az aktív macro értéke fog megjelenni a hívás helyén. Az *5. mellékletben* láthatók az *EEPROMHandler* definiált macro-i és a módszer, amellyel aktiválom és deaktiválom a két eszköz macro-it.

Az *EEPROMHandler* tulajdonságai a macro-kon kívül teljes mértékben megegyeznek, amely azt jelenti, hogy az EEPROM-ban ugyanazokkal a függvényekkel tudok írni, és onnét tud a program adatot kiolvasni. Továbbá az inicializáló, és az adatmentésért felelős feladat szekvenciája (*init task*) is megegyezik az *espTouch*-éval. Különbség a menteni kívánt adatokban lesz, amelyeket az *5.1.3. EEPROM szerepe az adatbázis felépítésében* című alfejezetben írtam le.

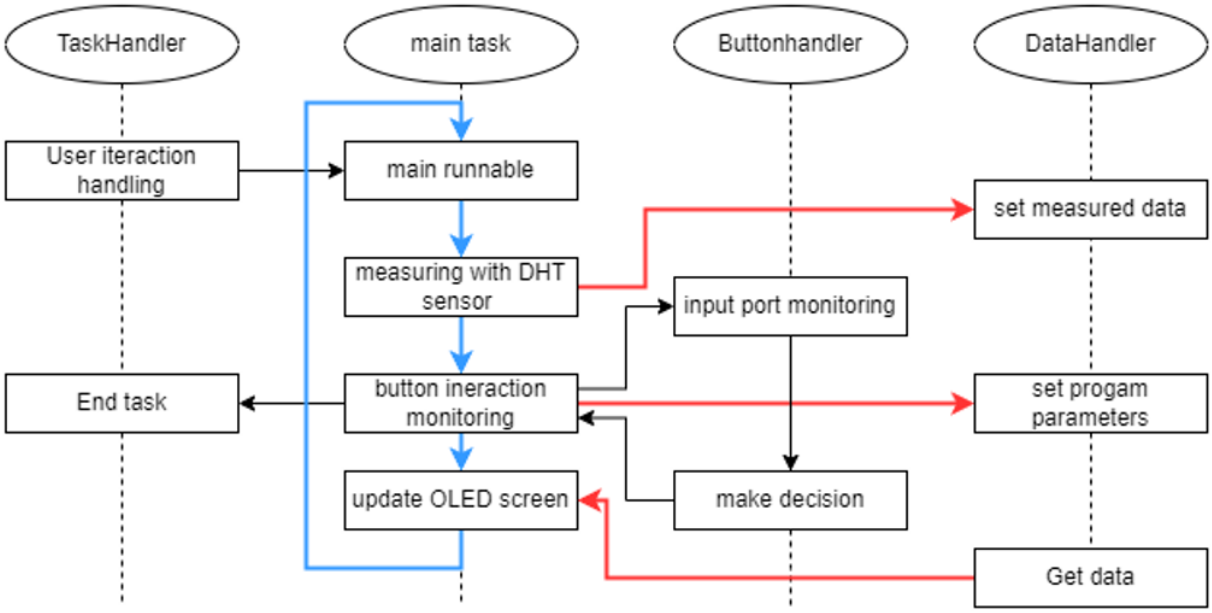
Az *espCarryable* a *wifi task*-ján keresztül fog az *espTouch*-csal kapcsolatba lépni és adatot közölni, illetve fogadni. Az *espCarryable* hasonló módon működik ezen a téren is, mint az *espTouch*. Az eszköz működtet egy webszervert, amelyhez tartozik egy weboldal és Websocket kommunikáció. Az *espCarryable* esetében a *szerverre* csak webböngészőből lehet csatlakozni monitorozás céljából. A weboldal az *espTouch* weboldalán keresztül lesz elérhető, amely programját az *espCarryable* fogja generálni.

A két létrehozott modul *szerver-kliens* kapcsolatot épít ki egymás között, aminek a kommunikációs alapját a Websocket protokoll fogja megvalósítani. Az *espCarryable* HTTP GET kéréssel csatlakozik a *szerverhez* (*espTouch*), amely elindítja a Websocket kommunikációt a két eszköz között. A kapcsolat kialakításához szükség van az *espTouch* elérési adataira, amely a portszám, és a routertől kapott IP cím lesz.

A router automatikusan oszt ki egy-egy IP címet a csatlakozó eszközöknek, így gondoskodni kell arról, hogy az *espCarryable* megtalálja az *espTouch* csatlakozási pontját. Mai routerekben alapvető funkció az, hogy a felhasználó a routert konfigurálni tudja, illetve MAC cím alapján statikus IP címet állítson be az eszközeinek. Az *espCarryable* konfigurációja során a *szerver* IP címét lehet megadni, amely az *espTouch szerver* csatlakozási pontjául szolgál. Ebből következik, hogy a rendszer felépítésekor az *espTouch*-hoz kell rendelni egy statikus IP címet, továbbá miden egyes *espCarryable* készülék konfigurációja során ezt az IP címet kell beállítani az EEPROM-ba.

5.3.1. Főfeladat részletes ismertetése

Az *espCarryable*-höz egy OLED kijelző lett csatlakoztatva, és a nyomógombok segítségével tud a felhasználó a rendszerben változásokat előidézni. Ezekről a *4.4. espCarryable felhasználói interakcióinak tervezése* című fejezetben írtam.



53.ábra. *espCarryable main task* szekvenciája

A főfeladat szekvenciája az *espCarryable*-nek az *53.ábrán* látható. Az eszköz feladatkezelőjének a *TaskHandler User interaction* függvénye fogja trigger-elni a *main task*-nak a *main runnable* funkcióját. A *main runnable* fogja futtatni egymás után a *measuring with DHT sensor*-

t, a *button interaction monintoring*-t, és az *update OLED screen* függvényeket.

A *measuring with DHT sensor* függvényeken keresztül fog az *espCarryable* a DHT szenzorral kommunikálni One-Wire buszon keresztül. A mért hőmérséklet és páratartalom a *set measured data* funkcióval az adatbázisba kerül elmentésre.

Az *update OLED screen* függvényben az OLED kijelző design-ában megjelenő változók értékeit fogom frissíteni az adatbázisban megtalálható új értékekkel. Ezeket az elemeket az adatbázisból (DataHandler) a *Get data* funkció segítségével lehet lekérdezni. Az új adatokat I^2C buszon keresztül küldöm el a kijelző modulnak, amely a beérkező adatokat feldolgozza, és frissíti a kijelzett szövegeket.

A *button interaction monitoring* szekvencia a fizikai gombok interakcióját monitorozza az *input port monitoring* segítségével, valamint visszakap egy flag-t *a make decision* funkciótól, hogy milyen beavatkozás történt. Utóbbi két függvény a *Buttonhandler* objektumhoz tartozik, amely egy szoftver-hardver interfészt valósít meg. A *Buttonhandler*-ben az ESP32 bemeneti lábainak a feszültség potenciáljának nagyságát figyeli, illetve csak az előre definiáltakat, amiket a *4.4. espCarryable felhasználói interakcióinak tervezése* című fejezetben leírtam. A bemeneti portokon közel 0V és 5V nagyságú feszültség jelenhet csak meg. Ha 5V jelenik meg a bemeneten, akkor a porthoz rendelt flag értéke logikai igaz lesz.

A flag-k a következők lehetnek:

- Kívánt hőmérséklet növelése.
    - *U1* lenyomásával kerül igaz állapotba.
- Kívánt hőmérséklet csökkentése.
    - *U2* lenyomásával kerül igaz állapotba.
- Aktív program növelése.
    - *U1*, *U3* gombok kombinációja teszi igaz állapotba a flag-et.
- Aktív program csökkentése.
    - *U2*, *U3* gombok kombinációja teszi igaz állapotba a flag-et.
- *serial task* indítás, illetve minden más feladat befejezése.
    - *U1*, *U2* gombok kombinációja teszi igaz állapotba a flag-et.

A program futása során egyszerre csak egy flag állapota lehet igaz. A *make decision* függvény fog dönteni a flag-ek alapján, hogy az adatbázisban milyen adatot kell elmenteni. Ha a *serial task*-t kell elindítani, akkor visszalép a program a *TaskHandler*-be, és véget érnek a feladatok, és elindul a *serial task.* Ha a többi gombkombináció valamelyike teljesül, akkor vagy az aktív programszám, vagy a kívánt hőmérséklet fog változni. Az aktív program számának lépésszáma 1, míg a kívánt

hőmérséklet lépésszáma 0.2°C pozitív és negatív irányban is. A változás az adatbázisban a *set program* függvénnyel lesz eltárolva. A funkció megkapja a változás nagyságát, és hozzáadja azt az adatbázis eredeti értékéhez.

6. Összefoglaló

Irodalomjegyzék

[1] Piyu Dhaker: *Introduction to SPI Interface*
Analog Dialogue 52-09, September 2018

[2] DHT11 hő és páratartalom mérő modul:
https://www.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translated-Version-1143054.pdf
2024. augusztus 12.

[3] Mr. Bhrijesh N. Patel (, Mr. Mrugesh M. Prajapati): OLED: A Modern Display Technology
International Journal of Scientific and Research Publications, Volume 4, June 2014

[4] Jonathan Valdez (, Jared Becker): Understanding the I2C Bus
Application Report Texas Instruments, SLVA704 June 2015

[5] Dan Awtrey: Transmitting Data and Power over a One-Wire Bus
Sensors: The journal of Applied Sensing Technology, Dallas Semiconductor, February 1997

[6] Thomas Kugelstadt: The RS-485 Design Guide
Texas Instruments, Application Report – SLLA272D, február 2008

[7] John Griffith: RS-485 Basics: When Termination Is Necessary, and How to Do It Properly
Texas Instruments, Technical Article – SSZTB23, július 2016

[8] Anthony Bishop: CRYDOM company: Solid-State-Relay Handbook with applications

[9] Szilárd test relé katalógusa:
https://hu.mouser.com/datasheet/2/657/lr_series_ac_pcb_mount_ssr_datasheet-2933688.pdf
2024. augusztus 12

[10] Suresh C. Satapathy (, Vikrant Bhateja):Smart Computing and Informatics
Smart Innovation, Systems and Technologies volume 77, Proceedings of the First International Conference on SCI 2016, Volume 1
2024. október 25

[11] Silicon Labs: Single-Chip to UART bridge (CP2102)
https://www.alldatasheet.com/view.jsp?Searchword=Cp2102%20datasheet&gad_source=1&gclid=Cj0KCQiAire5BhCNARIsAM53K1iE5ZLfoGGdDQQulM2ZRnaksywbuEVzZ-3DgcYuGquAWAt72OPwe1kaApUkEALw_wcB
2024. november 08

[12] George Thomas: Introduction to the:Modbus Protocol
the EXTENSION, A Technical Supplement to Control Network, Contemporary Control Systems, Inc. Volume 9 Issue , July – August 2008
2024. november 09

[13] Dejan Skvorc (, Matija Horvat, Sinisa Srbljic): Performance evaluation of Websocket protocol for implementation of full-duplex web streams
Conference Paper, ResearchGate, DOI:1.11.9/MIPRO.2014.6859715, May 2014
2024. november 10

[14] International Standard: Information technology - Telecommunications and information exchange between Systems - Twisted pair multipoint interconnections
Second Edition, 1993 december 15., ISO/IEC 8482

[15] Javier Calpe, (Italo Medina, Alberto Carbajo, María J. Martínez): AD7879 Controller Enables Gesture Recognition on Resistive Touch Screens
Analog Dialogue 45-06, June 2011

[16] Nicolas Melot: Study of an operating system: FreeRTOS
http://wiki.csie.ncku.edu.tw/embedded/FreeRTOS_Melot.pdf
2024.11.24

Ábrajegyzék

[A1] https://github.com/witnessmenow/ESP32-Cheap-Yellow-Display/blob/main/OriginalDocumentation/4-Driver_IC_Data_Sheet/ESP32-WROOM-32.PDF
(utolsó látogatás időpontja: 2024. július 18.)

[A2] https://www.analog.com/media/en/analog-dialogue/volume-45/number-2/articles/gesture-recognition-on-resistive-touch-screens.pdf
(utolsó látogatás időpontja: 2024. július 18.)

[A3] https://www.analog.com/media/en/analog-dialogue/volume-35/number-1/articles/the-pda-challenge-met.pdf
(utolsó látogatás időpontja: 2024. július 18.)

[A4] Piyu Dhaker: *Introduction to SPI Interface*
Analog Dialogue 52-09, September 2018

[A5] https://www.tme.eu/Document/7a4fd48d400b8c4c8309ef1e2b13cdd4/MR003-005-1.pdf
(utolsó látogatás időpontja: 2024. július 18.)

[A6] https://www.allaboutcircuits.com/technical-articles/low-pin-count-serial-communication-introduction-to-the-1-wire-bus/
(utolsó látogatás időpontja: 2024. augusztus12.)

[A7] https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=404ce87063f98616c42e08b2c969f8d52cfa6a1e
(utolsó látogatás időpontja: 2024. július 18.)

[A8] https://espressif-docs.readthedocs-hosted.com/projects/arduino-esp32/en/latest/api/i2c.html
(utolsó látogatás időpontja: 2024. július 18.)

[A9] Application Report The RS-485 Design Guide by Texas Instruments
(utolsó látogatás időpontja: 2024. augusztus 20.)

[A10] Technical Article RS-485 Basics: When Termination Is Necessary, and How to Do It Properly by Texas Instruments
(utolsó látogatás időpontja: 2024. augusztus 20.)

[A11] https://olegkutkov.me/2019/11/08/rs-485-practice-and-theory/
(utolsó látogatás időpontja: 2024. augusztus 21.)

[A12] https://www.allaboutcircuits.com/technical-articles/basics-of-ssr-solid-state-relay-the-switching-device/
(utolsó látogatás időpontja: 2024. július 25.)

[A13] https://www.sensata.com/products/relays/lr-series-ac-output-pcb-mount-ssr-lr600240d25r
(utolsó látogatás időpontja: 2024. augusztus 5.)

[A14] https://www.ti.com/lit/an/slva704/slva704.pdf?ts=1729748308484&ref_url=https%253A%252F%252Fwww.google.com%252F
(utolsó látogatás időpontja: 2024. október 24.)

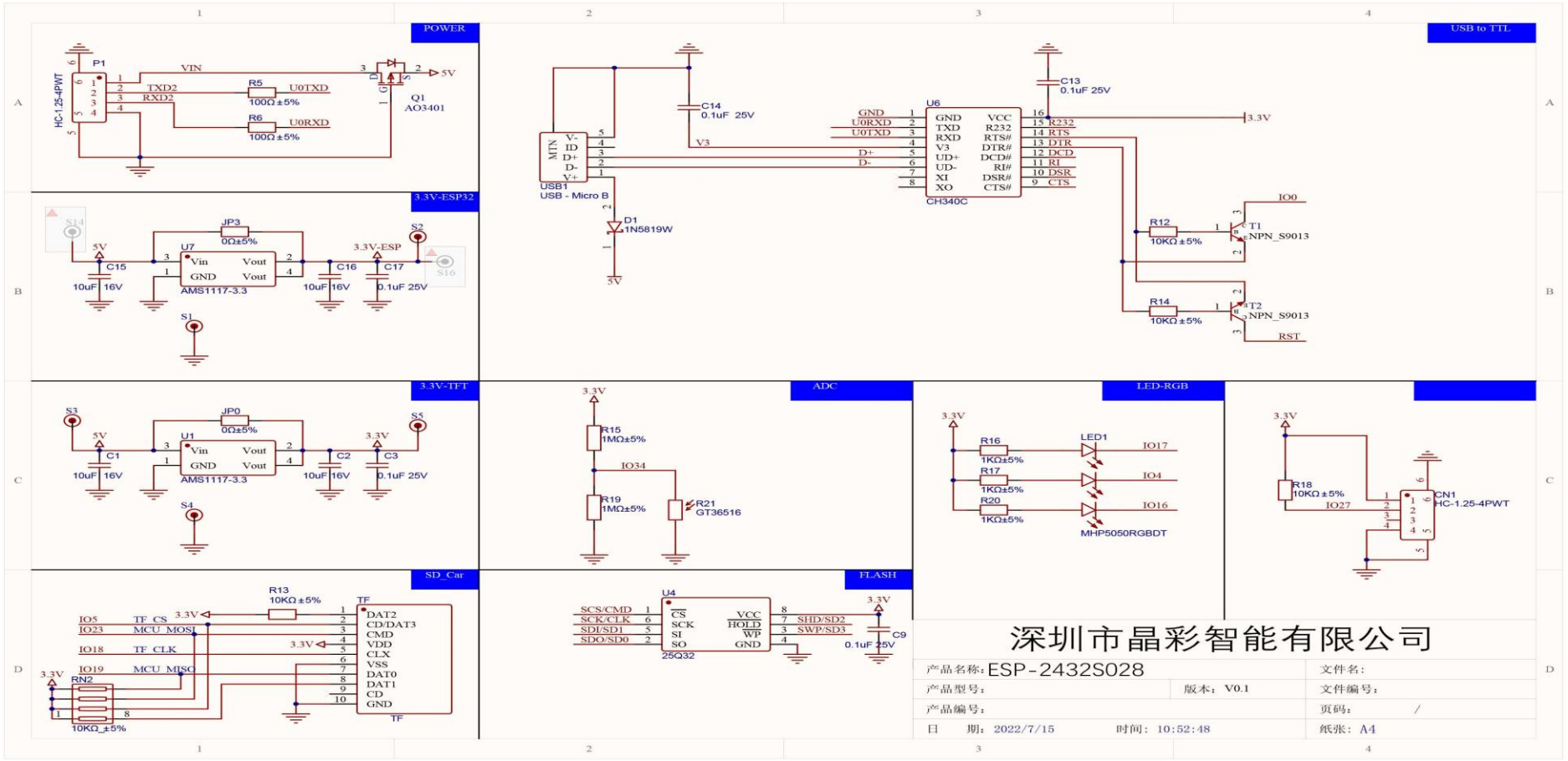
[A15] Suresh C. Satapathy (, Vikrant Bhateja): Smart Computing and Informatics
Smart Innovation, Systems and Technologies volume 77, Proceedings of the First International Conference on SCI 2016, Volume 1
2024. október 25

[A16] George Thomas: Introduction to the:Modbus Protocol
the EXTENSION, A Technical Supplement to Control Network, Contemporary Control Systems, Inc. Volume 9 Issue , July – August 2008
2024. november 09

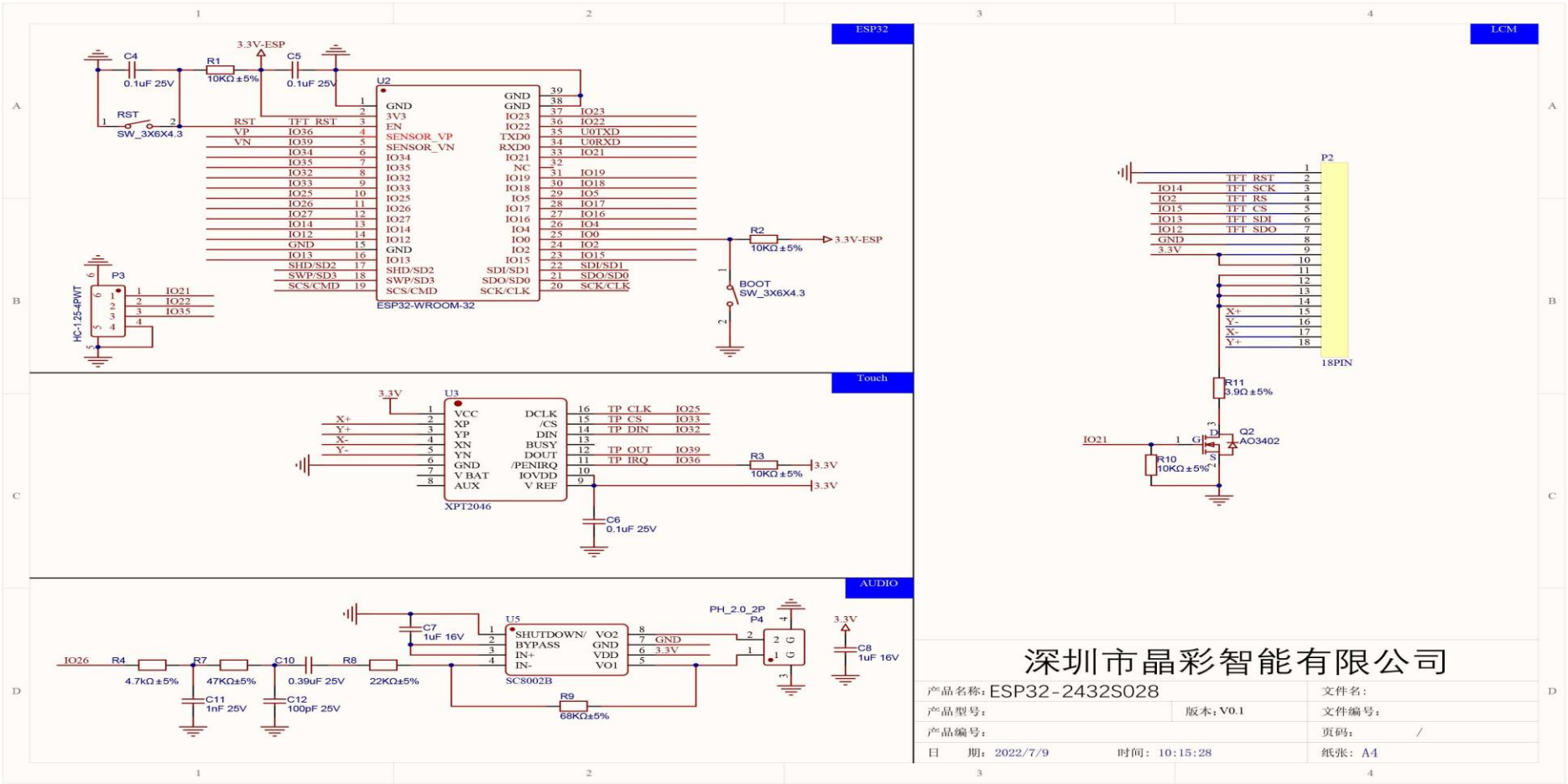
[A17] Dejan Skvorc (, Matija Horvat, Sinisa Srbljic): Performance evaluation of Websocket protocol for implementation of full-duplex web streams
Conference Paper, ResearchGate, DOI:1.11.9/MIPRO.2014.6859715, May 2014
2024. november 10

Mellékletek

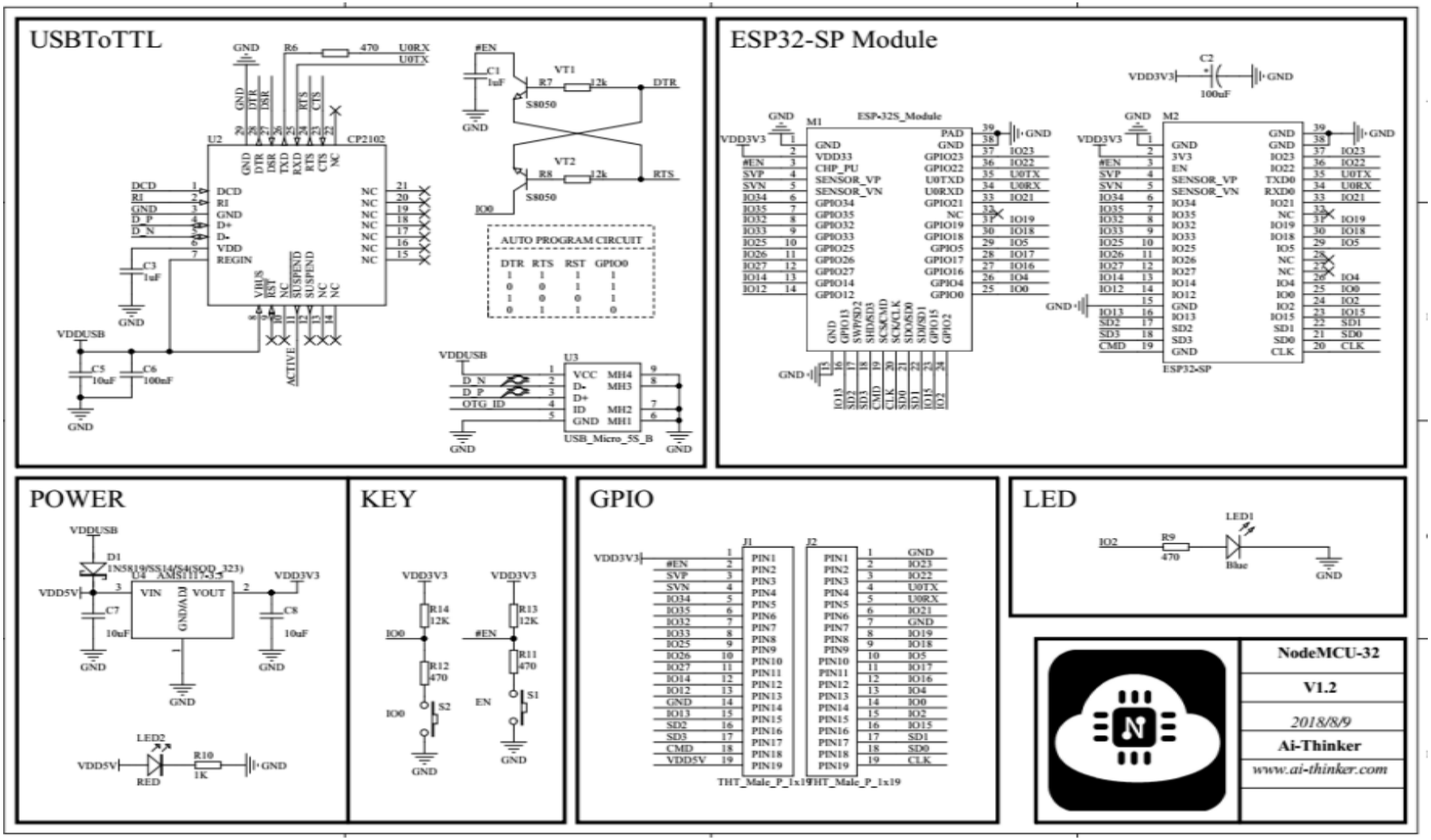
1. melléklet



2. melléklet


ESP32
LCM
Touch
AUDIO
U2
ESP32-WROOM-32
U3
XPT2046
U5
SC8002B
P2
18PIN
P3
HC-1.25-4PWT
P4
PH_2.0_2P
Q2
AO3402
BOOT
SW_3X6X4.3
RST
SW_3X6X4.3
深圳市晶彩智能有限公司
产品名称: ESP32-2432S028
产品型号:
产品编号:
版本: V0.1
日 期: 2022/7/9
时间: 10:15:28
文件名:
文件编号:
页码: /
纸张: A4

3. melléklet


USBToTTL
ESP32-SP Module
POWER
KEY
GPIO
LED
AUTO PROGRAM CIRCUIT
NodeMCU-32
V1.2
2018/8/9
Ai-Thinker
www.ai-thinker.com

4. melléklet

```html
<html>
    <head>
        <title>ESPTouch</title>
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet" type="text/css" href="styles.css">
    </head>
    <body>
        <div class="content">
            <div class="card-grid">
                <div class="card">
                    <p class="card-title">Active program</p>
                    <p class="reading"><span id="active_prog"></span></p>
                </div>
                <div class="card">
                    <p class="card-title">Wanted temperature</p>
                    <p class="reading"><span id="wtmp"></span> &deg;C</p>
                </div>
                <div class="card">
                    <p class="card-title">Temperature</p>
                    <p class="reading"><span id="temp_"></span> &deg;C</p>
                </div>
                <div class="card">
                    <p class="card-title">Humadity</p>
                    <p class="reading"><span id="hmd_"></span> %</p>
                </div>
                <div class="card">
                    <p class="card-title">Heating status on 0. Circle: </p>
                    <p class="reading"><span id="heating_0"></span></p>
                </div>
                <div class="card">
                    <p class="card-title">Heating status on 1. Circle: </p>
                    <p class="reading"><span id="heating_1"></span></p>
                </div>
                <div class="card">
                    <p class="card-title">Heating status on 2. Circle: </p>
                    <p class="reading"><span id="heating_2"></span></p>
                </div>
                <div class="card">
                    <p class="card-title">Connected devices</p>
                    <p class="reading">ESPCarryable links:</p>
                    <ul>
                        <li><a href=http://192.168.0.107>192.168.0.107</a></li>
                    </ul>
                </div>
            </div>
        </div>
        <script src="script.js"></script>
    </body>
</html>
```