Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Elektronikai Technológia Tanszék

Kuglics Lajos
Megyeri István

Elektronikai gyártólaboratórium környezetének vizsgálata egyedi okoseszközökkel és mobil integrációval

KONZULENS

Dr. Géczy Attila, Dr. Martinek Péter

BUDAPEST, 2022

Tartalomjegyzék

Összefoglalás

A dolgozatunk célja az általunk készített, a villamosmérnöki karon található elektronikai gyártólaboratóriumba tervezett okoslaboratórium hardveres és szoftveres megvalósításának bemutatása.

Munkánk során irodalmi összefoglalót követően bemutatjuk, hogy miért és milyen eszközöket választottunk az IoT adatgyűjtő rendszer tervezéséhez és elkészítéséhez, valamint hogyan integráltuk ezeket az eszközöket egy könnyen használható, kis méretű szenzor-elektronikába (node-ba). Megmutatjuk, hogyan jutnak el az mért adatok egy központi számítógépen keresztül a megjelenítésükért felelős eszközökre.

Ismertetjük a hardveres és szoftveres fejlesztés során felmerülő nehézségeket és az ezeket orvosoló megoldásokat. Megmutatjuk, hogyan implementáltuk az alkalmazással szemben támasztott követelményeknek megfelelő funkciókat. Az alkalmazást először emulált Android környezetben teszteltük, majd egy valós fizikai eszközön is kapcsolatot teremtettünk a szenzoradatok forrásával. Végül az alkalmazást élesben is leteszteltük az Elektronikai Technológia Tanszék laboratóriumában. Ezzel a jövőben a labor munkakörülményeinek vizsgálata a jövőben kibővíthető, sőt az eszközeink akár konkrét berendezést validáló mérőegységek fogadásaira is képesek lehetnek.

A munkánk eredményét később fel tudjuk használni a laboratórium légminőségének javítására, valamint hasznos adatokat gyűjthetünk vele az ott végzett különböző munkafolyamatok lehetséges egészségkárosító hatásairól, munkavédelmi aspektusairól.

Abstract

The goal of our thesis is to present an integrated smart air quality monitoring and visualisation solution we created in the laboratory of the Department of Electronics Technology at the Faculty of Electrical Engineering and Informatics.

After a short literature review, we are going to present the tools used to build an IoT system that is capable of collecting and visualising data. We will show the reader how we designed a small and easy-to-use sensor board, which can transfer its data wirelessly to accommodate portable usage. We will present how we processed the data generated by this node, and how we made it available for multiple visualisation clients to use.

We created an Android application that is capable of visualizing environmental data collected by a group of sensors from the laboratory. Although we prioritised functionality, some design elements were implemented to provide smoother user experience. Initial tests were performed on an emulated device in Android Studio. Later on, tests were performed in a laboratory environment with the use of an actual Android tablet. In the future, this will allow us to expand the laboratory's working environment and even to have the ability to receive measuring units validating specific equipment.

Thanks to the foundations we created during our work, we can use the data provided by this system to improve the air quality of the laboratory, and possibly identify any operational safety and health processes that might cause air pollution endangering the people working there.

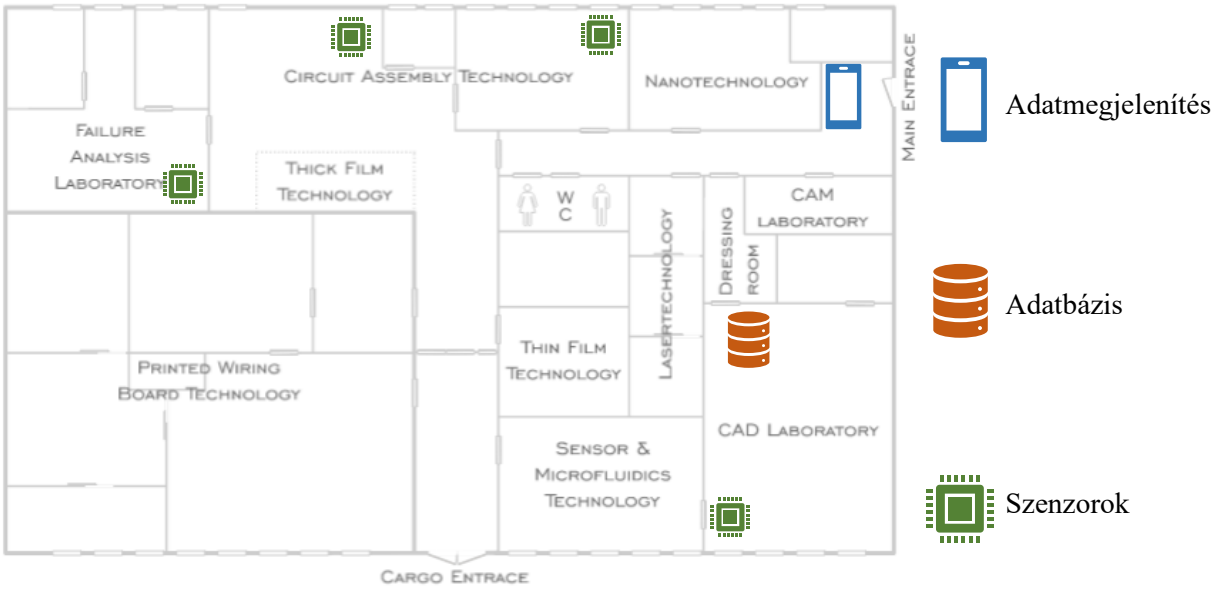
1 Bevezetés

1.1 Motiváció

Mivel a levegő a mindennapi élet része, ezért fontos a különböző paramétereinek ismerete, monitorozása. Egészségügyi szempontból a minősége az egyik legfontosabb változó: megfelelő levegő hiányában olyan panaszok merülhetnek fel, mint a levertség, fejfájás, a szem irritációja; huzamosabb idejű kitettség esetén akár komolyabb krónikus betegségeket is okozhat. A körülöttünk mindenhol jelen levő szennyező anyagokat sok esetben nem is érzékeljük, ezért különösen fontos környezetünk folyamatos ellenőrzése.

1.2 Feladat specifikálása

A feladatunk célja egy olyan rendszer létrehozása, amely képes az ambiens levegőminőséget monitorozni szenzorok segítségével, majd az összegyűjtött adatokat hosszú távon eltárolni, igény szerint okostelefonon vagy tableten megjeleníteni. Ezt a Segítségével naprakész információkat kaphatunk a laboratórium levegőjéről, amit akár munkavédelmi célból is felhasználhatunk. A szenzor-elektronika felé elvárás a különböző környezeti változók mérése, továbbítása, mindezt autonóm módon. Az adatgyűjtő rendszer fő feladata a folyamatos működés, a beérkező adatok hosszú távú tárolása, valamint ezen adatok elérhetővé tétele a különböző adatvizualizáló kliensek felé. Az alkalmazás a fő funkcióit tekintve képes kell legyen nézetek közötti navigációra, HTTP kommunikációra, illetve valós idejű adatokból grafikus megjelenítésre. A feladat elvégzése során foglalkoztunk hardverfejlesztéssel, beágyazott szoftverfejlesztéssel,



1. ábra: A Smartlab projekt terve

adatbázis és webszerver programozással, valamint használtuk a Kotlin nyelvet és az Android Studio fejlesztői környezetet. A projektben részt vevő eszközök elhelyezésének egy tervét az 1-es ábra szemlélteti.

2 Irodalomkutatás, eszközök feltárása

2.1 Levegőminőség

Egy ember naponta több ezer liter levegőt lélegez be. Eme levegő összetétele az életünk minőségét egyik leginkább meghatározó tényező. Egy átlagos (USA-ban lakó) ember az idejének közel 90%-át zárt térben tölti, így különösen fontos ezen terek levegőjének monitorozása. Az egészségünkre legnagyobb hatással bíró, levegőben megtalálható anyagok a szálló por, valamint az illékony szerves anyagok [1].

2.1.1 Finomszemcsés anyagok

A finomszemcsés anyagok (PM – particulate matter) a levegőben lebegő szennyező jellegű, egészségre káros részecskéket jelentik. Az ilyen részecskéket több kategória szerint is csoportosíthatjuk, például származás, fizikai, kémiai tulajdonságok; legpraktikusabb azonban a méret szerinti megkülönböztetés (PM10, PM2.5 stb.) [2].

2.1.2 Illékony szerves anyagok

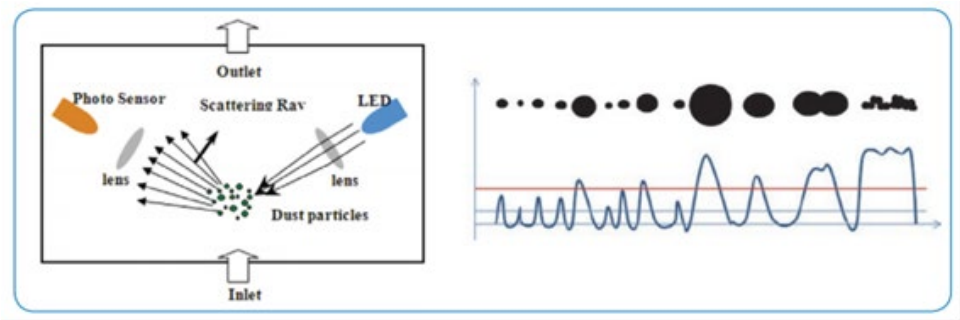
Az illékony szerves anyagok (VOC-k) legtöbbször emberi tevékenység eredményeként jutnak a levegőbe. Ezek a gázok folyékony vagy szilárd anyagok párolgásakor keletkeznek. Ilyen VOC-k megtalálhatóak az építőanyagokban, lakkokban, festékekben, valamint rengeteg háztartási vegyiáruban [3]. Különösen veszélyesek lehetnek az MDF lapból készült bútorok: a bennük található formaldehid párolgása miatt a mért szennyezőanyagok akár a határérték 20-szorosát is elérhetik. A VOC-k koncentrációjának növekedése szem- és torokirritációval, rossz közérzettel, fejfájással is járhat [4].

2.1.3 A levegőminőség mérése

A levegőben található szennyezőanyagok mérésére rengeteg szenzor kapható, amelyek funkciójuk, felbontásuk, valamint áruk alapján széles felhasználási területet fednek le. Ezek között pontos, precíz kvantitatív mérésre csak a drága, kalibrált szenzorok alkalmasak, azonban az olcsóbb szenzorok is tökéletesen alkalmazhatóak kvalitatív mérésekre, valamint a koncentrációk időbeli trendjeinek feltérképezésére. Kémiai anyagok esetében az összetételük a legfontosabb jellemzőjük, míg szálló por esetén a fizikai méretük hordozza a legtöbb információt.

A szálló por koncentrációját mérő szenzorok legtöbbször optikai úton működnek. Ezekben egy fényforrás és egy fényérzékelő található. Két fő típus létezik: az egyikben a

forrás és az érzékelő közé került porszemcsék által kitakart fény okozta intenzitáscsökkenést detektáljuk (direkt), a másikban a porszemcsékről szóródó fényt mérjük (light scattering); ilyen szenzor látható a 2. ábrán. A levegő mozgatásáért egyszerűbb szenzorok esetén egy fűtőellenállással előállított konvekció felel, drágább modellek esetén beépített ventilátor.



2. ábra: A szóródó fényt detektáló optikai detektor működése [5]

Illékony szerves anyagok mérésére elektrokémiai szenzorokat használhatunk. Ezekben az érzékelő elem kémiai kölcsönhatásba lép a mérendő anyaggal, aminek következtében mérhető elektromos változás következik be. Ide tartoznak az egyre szélesebb körben elterjedt MEMS alapú gázdetektorok is: ezekben egy felhevített érzékelő található (hotplate), amely illékony szerves anyagok hatására megváltoztatja az ellenállását [6].

2.2 Android

Az Android egy Linux alapú operációs rendszer. Legnagyobb előnye, hogy elterjedtségének köszönhetően az erre az operációs rendszerre fejlesztett alkalmazások rengeteg céleszközön futtathatóak. Az Android, mint fejlesztő-platform rengeteg lehetőséget biztosít a fejlesztő számára: az ingyenesen elérhető könyvtárak segítségével gyorsan és hatékonyan lehet programokat (applikációkat) készíteni [7].

2.2.1 Kotlin nyelv

A Kotlin egy erősen típusos, magas szintű programozási nyelv, amely JVM-re (Java Virtual Machine) vagy JavaScript kódra fordítható. A nyelv támogatja az objektum orientált, illetve a funkcionális programozást is. A Kotlin nyelv sikeressége mögött több alapelv áll, mint például a nyelv folyamatos modernizálása, valamint a felhasználói bázis visszacsatolásának folyamatos figyelembevétele. Azonban a legfontosabb tulajdonsága, hogy egy projekten belül a Kotlin és a JavaScript nyelvet is használhatjuk, ezért könnyű az áttérés a két programnyelv között [8].

A Kotlin nyelv mögött a JetBrains nevű cég áll. A cég profilja a fejlesztők munkáját elősegítő eszközök készítése, így a Kotlinnak is nagyon erős úgynevezett „*Tooling*” támogatása van. Ilyenek a fejlesztői környezetek, mint az *IntelliJ IDEA* amelyen az Android Studio is alapszik [9].

2.2.2 Java Virtuális Gép

A Java virtuális gép (JVM) egy olyan program, amelynek a feladata Java bájtkód futtatása. A virtuális gép legfőbb feladata, hogy a Java bájtkódra fordított programok bármilyen eszközön bármilyen operációs rendszer alatt lefussanak. Ezen felül a JVM felügyeli a programmemória kezelését is [10].

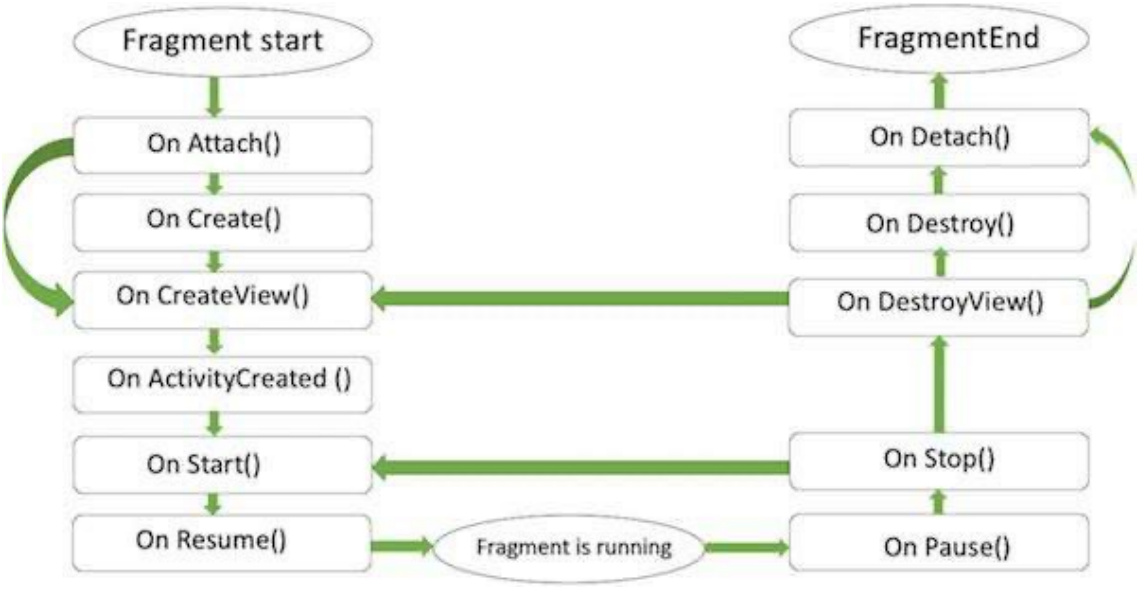
2.3 *Activity* és *Fragment*

Minden Android program alap építőköve az *Activity*. Ez a komponens felelős a képernyő inicializálásáért, valamint a felhasználó és a rendszer szolgáltatásainak összekötéséért. Az Android elterjedésének kezdeti időszakában a felhasználói interfész megjelenítése teljes egészében egy-egy *Activity*nek a feladata volt. Az akkoriban kiadott applikációk egyszerű felhasználói felülettel rendelkeztek, illetve az Androidos eszközök formavilága sem volt ennyire sokrétű, ezért ez a megoldás elfogadhatóan működött. A legtöbb esetben néhány egyedi felhasználói felület implementálása elegendő volt, hogy az eszközök jelentős hányadán megfelelően fussanak az applikációk.

Azonban ma a piaci sokrétűségnek köszönhetően ez már nem járható út. Minden okostelefonos applikáció esetében elvárás a dinamikus és interaktív felhasználói felület. Ahhoz, hogy ezt több képarányon, felbontáson is hatékonyan tudjuk biztosítani, a programnak reszponzívnak kell lennie; erre kínál megoldást a *Fragment* osztály.

Az Android *Fragment*ek lehetővé teszik, hogy felosszuk a felhasználói felületeinket. Ezekben a partíciókban funkcionalitás szerint rendezhetjük el a szükséges interfész elemeket. Egyetlen *Activity* több *Fragment*et is kezelhet: a *Fragment*ek kezelik az adott kijelzőhöz való igazodást, amíg az *Activity* egy magasabb perspektívából irányítja az interfész feladatokat [11].

Az *Activity*-k és *Fragment*-ek is saját életciklussal rendelkeznek: ezen ciklusok között minden váltásnál egy adott függvény hívódik meg; ezek a függvények a 3. ábrán láthatóak. Az applikációban történő navigálásnál a *Fragment* Navigation Graph definiálja a *Fragment*-ek közötti átmeneteket. A felhasználó által megtett utat a Back Stack tárolja LIFO adatszerkezetben [12][13].



3. ábra: Fragmentek életciklusai

2.4 Android navigációs alapelvek

Android applikáció fejlesztésekor célravezető pár alapelv betartása. Ilyen alapelv a fix, mindenhonnan elérhető főmenü: a navigációt úgy kell kialakítani, hogy bármilyen útvonalat bejárva a vissza gombok megnyomásával a teljes applikációból való kilépés előtt közvetlenül a kezdő felületre érkezzen vissza a felhasználó.

A visszalépés konzisztenciája: a vissza gomb a képernyő alján helyezkedik el, feladata pedig a már bejárt felületeken való visszafelé lépegetés, a kronológiai sorrend betartásával [14].

2.5 Android LiveData

A LiveData egy olyan osztály, amelynek változóihoz megfigyelőket, ún. *observer*eket csatolhatunk. Amikor a LiveData objektum valamely változója értéket vált, a hozzá csatolt *observer*ek erről értesülnek. Ez a struktúra biztosítja, hogy a felhasználói felületen mindig a legfrissebb adatokkal dolgozunk. Ez azért hasznos, mert így nem kell a megfigyelni kívánt adatok állapotait nyilván tartani – az *observer*ek segítségével könnyen építhetünk eseményvezérelt programfolyamatot [15].

2.6 *ViewModel*

Alapesetben az Android keretrendszer kezeli a felhasználói felület vezérlőit, az *Activity*ket és a *Fragment*eket. Amikor a keretrendszer úgy dönt, hogy leállít és újból inicializál egy ilyen elemet, akkor minden abban nem perzisztens módon tárolt adat

elveszik. Amennyiben ezeket az adatokat minden újbóli inicializálás során vissza akarnánk állítani, az rengeteg erőforrást venne igénybe.

A *ViewModel* osztály feladata a felhasználói felület számára releváns adatok kezelése és tárolása. Az így tárolt adatok képesek túlélni a konfigurációs változásokat, mint például a képernyő elforgatása, ami során az *Activity*ből vagy *Fragment*ből egy új példány keletkezik [16].

3 Szenzoráramkörök, alkalmazott szenzorok

3.1 Tervezés

A szenzoráramkör (node) tervezésénél több fontos szempont alapján kellett dolgoznunk. Nagyon fontos volt a szenzor fogyasztásának alacsonyan tartása: a laboratóriumban nem található mindenhol hálózati dugalj, így amennyiben szeretnénk egy adott gép (pl. forrasztó kemence) közelében méréseket végezni, azt akkumulátorról kell végeznünk. A hosszú időtartamú mérésekhez ezért elengedhetetlen volt a teljesítmény-felvétel minimalizálása.

A mindennapi használathoz elengedhetetlen volt a vezeték nélküli kapcsolat lehetősége: ez biztosítja, hogy a laboratórium bármely pontján tudunk méréseket végezni. A legkézenfekvőbb megoldás természetesen egy Bluetooth/WiFi képes mikrokontroller használata volt. Végül a WiFi mellett döntöttünk a nagyobb hatótáv és a könnyebb bővíthetőség miatt.

Mivel vezeték nélküli rendszerek esetén gyakori probléma a jelvesztés és az ezzel járó adatvesztés, ezért egy plusz adattárolóra is szükségünk van, ami hiba esetén helyettesíti a központi adatgyűjtő szervert. Erre a célra egy SD kártya foglalatot is terveztünk a rendszerbe: az ebben található SD kártyára minden mért adatot elmentünk, így a WiFi rendszer hibája esetén sem veszítünk el adatot, valamint a kártya tartalmát könnyen és gyorsan kiolvashatjuk egy számítógép segítségével.

Végül, de nem utolsó sorban figyelnünk kellett a node méretének lekorlátozására: amennyiben túl nagyra tervezzük, nem tudjuk problémamentesen elhelyezni a laboratórium több pontjára is, valamint nem tudunk vele mobilis méréseket végezni (például zsebben hordva).

3.2 Hardverfejlesztés

3.2.1 Szenzor - BME680

Mivel a projekt a laboratóriumra fókuszált, ezért a szenzor kiválasztásánál azok a jellemzők kaptak nagyobb hangsúlyt, amelyek egy zárt térben jobban hozzájárulnak a levegő minőségéhez. Ezért választottuk a Bosch BME680-as szenzorát, amely egy elterjedt, költséghatékony páratartalom, hőmérséklet, nyomás, és VOC szenzor. Kis mérete (3x3x1mm3) és alacsony fogyasztása miatt nagyszerű választás volt a projekthez.

A fejlesztés elősegítéséhez a Bosch egy Arduino könyvtárat is elérhetővé tett, így a szenzor “nyers” adatait egy kifinomultabb algoritmus segítségével dolgozhatjuk fel [17].

3.2.2 MCU – ESP8266

A szenzoradatok olvasásához, vezeték nélküli továbbításhoz, valamint az SD kártyával való kommunikációhoz egy erősebb vezérlő integrált áramkörre volt szükségünk. Ezért választottuk az Espressif Systems ESP8266-os mikrokontrollerét. Ez az MCU beépített 2.4GHz-es vezeték nélküli rádióval rendelkezik, 80Mhz-es órajelének, SPI és I2C interfészének köszönhetően jó választás a feladatra; természetesen alacsony ára sem elhanyagolható szempont. A BME680-hoz tartozó könyvtár algoritmusának sok programmemóriára és RAM-ra van szüksége, valamint lebegőpontos számításokat kell végeznie, ezért ez a 32 bites, árkategóriájában relatíve sok ROM-al és RAM-al rendelkező mikrokontroller optimális választás [18].

Nagy előnyt jelent még, hogy beépített analóg-digitális átalakítóval rendelkezik, így a tápellátás felügyeletét is el tudja látni külső ADC nélkül. Széleskörű használatának köszönhetően rengeteg online segédanyag és szoftveres támogatás található hozzá, így gyorsan és hatékonyan lehet rá szoftvert fejleszteni.

3.2.3 Tápellátás

Mivel a node-ot akkumulátoros megtáplálás köré terveztük, ezért egy megfelelő áramforrást is kellett választanunk. Hordozható, kis méretű eszközök esetén a lítium alapú akkumulátor használata a legelterjedtebb: kis mérete, alacsony önkisülése, valamint nagy energiasűrűsége miatt ideális választás. Ezeknek az akkumulátoroknak üzemszerű állapotban a cellafeszültsége ~2,7V és 4,2V között mozog. A maximum feszültség túl magas a rendszer aktív eszközei számára, ezért a feszültség stabilizálására egy feszültségszabályzót is be kellett építenünk.

A prototípus elkészítéséhez az egyszerűség kedvéért lineáris feszültségszabályzót alkalmaztunk, így fontos volt a kimeneti és bemeneti feszültség közötti különbség alacsonyan tartása. Egynél több cella használata esetén töltéskiegyenlítő áramkört is biztosítanunk kellene, így az előző szempontok alapján 1 cellás lítium-polimer akkumulátort választottunk. A cella töltéséért egy Microchip MCP73843 típusú töltésvezérlő felel, amely beépített felügyeleti áramkörökkel rendelkezik az akkumulátor védelme érdekében [19].

Az áramkör megtáplálásához figyelembe kell vennünk az akkumulátor lehetséges feszültségtartományát, a mikrokontroller, a szenzor, és az SD kártya tápfeszültségét is. Az akkumulátor lehető legtöbb töltésének felhasználása érdekében a Torex Semiconductor XC6222-es 3,3V-os ULDO feszültségszabályzót választottunk. Kiválasztásának fő szempontja az alacsony dropout[1] feszültség, valamint a kedvező ár [20].

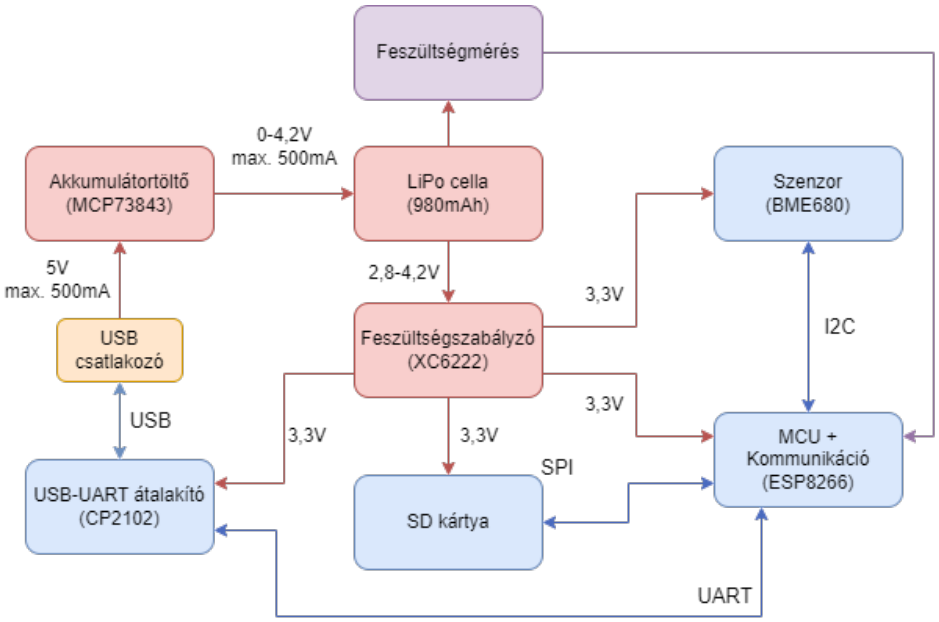
3.2.4 Kiegészítő áramkörök

A mikrokontroller programozásához, valamit az áramkör teszteléséhez szükségünk van az MCU soros portjához. Egy Silicon Labs CP2102N-es USB-UART átalakító, valamint megfelelő USB csatlakozó segítségével az áramkör könnyen és gyorsan felprogramozható [21]. A szoftverben elhelyezett debug-üzenetek segítségével egyszerűen nyomon tudjuk követni a program működését: a különböző változók értékeit számítógépes terminálon is megfigyelhetjük.

Az akkumulátor töltöttségének ellenőrzésére egy kapcsolható feszültségosztót is terveztünk, amelynek kimenetét a mikrokontroller analóg-digitális átalakítójába vezettünk. Segítségével tudjuk jelezni a felhasználó felé, amennyiben az akkumulátor feszültsége túl alacsony szintre csökken. Mivel kapcsolható, ezért nem jelent állandó terhelést az akkumulátor számára.

Az áramkör egyes részeinek kapcsolatát a 4-es ábra szemlélteti részletesen:

[1] Az a minimális feszültségkülönbség, amely adott kimeneti áramnál a szabályzó bemenete és kimenete között fenn kell, hogy álljon.

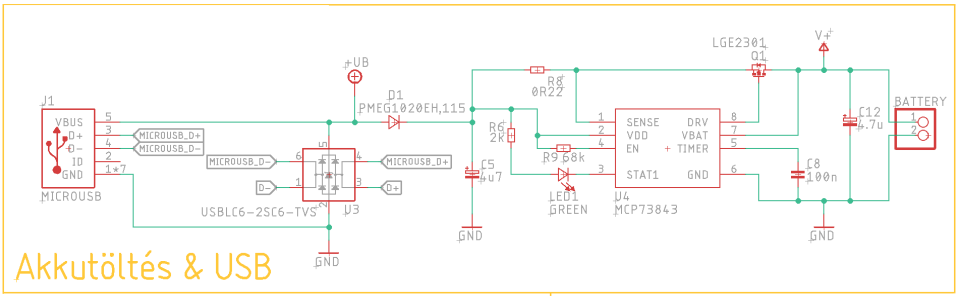


4. ábra: A szenzoráramkör blokkvázlata

3.3 Áramkörtervezés

Az áramkör tervezéséhez az Autodesk Eagle program 9.6-os verzióját használtuk, ami egy kombinált áramkör és NYÁK-tervező szoftver. A tervezés első lépése a kapcsolási rajz elkészítése volt.

3.3.1 USB + töltőáramkör



5. ábra: Az USB csatlakozó, valamint a töltőáramkör

Az áramkör tervezését a töltőáramkörrel kezdtük. Ahogy az 5-ös ábra mutatja, az USB csatlakozó vonalai először egy TVS diódalétrán mennek keresztül. Amennyiben egy statikusan feltöltődött test az áramkörrel érintkezik, az elektrosztatikus kisülést eredményezhet (ESD), amely az áramkör szempontjából akár végzetes is lehet. Ez ellen védenek a TVS diódák, amelyek a veszélyes feszültségimpulzusokat elnyelik.

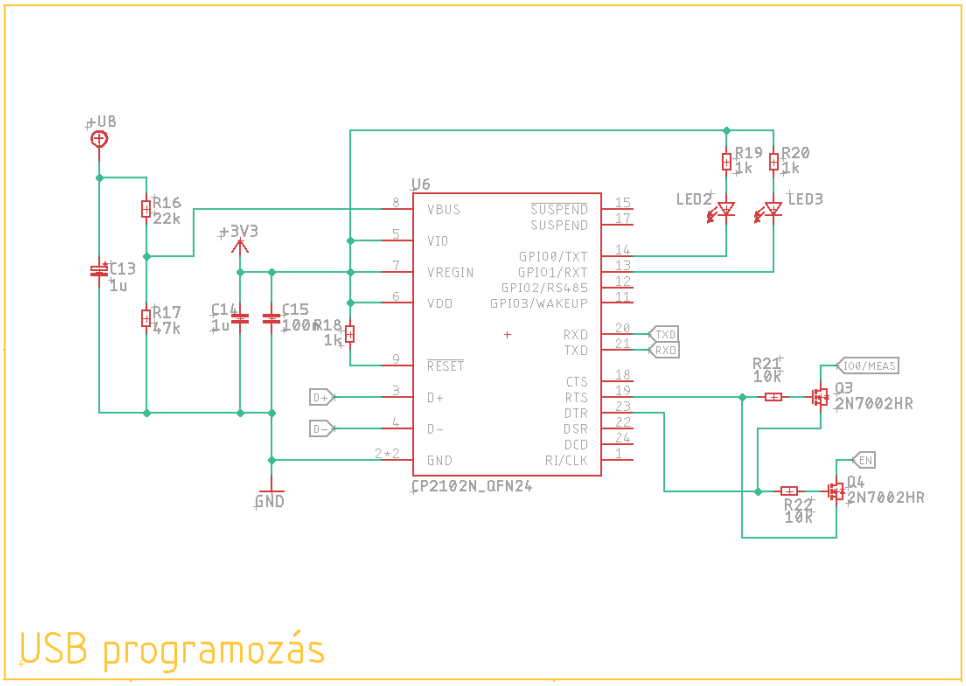
A TVS diódák után az adatvonalak az MCU felé, a tápvonal a töltőáramkörbe érkezik, előtte azonban egy Schottky-diódán megy keresztül. Ez a dióda megakadályozza, hogy akkumulátoros üzemmód esetén az akkumulátor feszültsége Q2 diódáján és R5-ön

keresztül visszajusson az UB tápvonalra, ezáltal azt a látszatot keltve, mintha az USB busz áram alatt lenne. Ez az USB-UART átalakító IC számára fontos.

A dióda után egy P típusú MOSFET segítségével szabályozzuk az áthaladó áramot, aminek a maximumát a soros R8 ellenálláson eső feszültség segítségével tudjuk beállítani. Ez a maximum töltőáram lítium alapú akkumulátorok esetén általában 1C, azaz 1A/Ah (azaz esetünkben 980mAh esetén 980mA). Mivel a használt USB szabványon keresztül maximum 500mA áramot tudunk felvenni, ezért ezt az áramot céloztuk meg. A töltő IC adatlapja alapján Rsense ≈ 0,11/$I_{töltő}$, amely esetünkben 0,22Ω-ra jön ki.

A töltés állapotát, esetleges hibáját LED1 jelzi. A C8 kondenzátor segítségével beállíthatjuk a töltéshez kapcsolódó biztonsági időzítők határait.

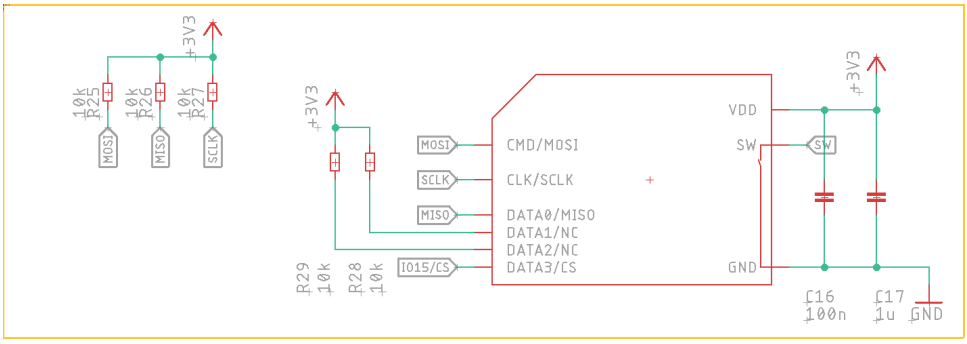
3.3.2 USB-UART átalakító



6. ábra: Az USB-UART híd áramkör

Mivel a modern számítógépeken nincs UART soros port, a mikrokontrollerben pedig nincs USB PHY, ezért a vezetékes kommunikáció megvalósításához egy segédáramkörre is szükségünk van. Esetünkben az 6-os ábrán látható USB-UART híd kapcsolás felelős a mikrokontroller programozásáért, valamint a debug üzenetek továbbításáért. Ez az IC egy feszültségosztón keresztül érzékeli az USB vonal állapotát. Kommunikáció közben két LED segítségével jelzi az adatátvitelt. Két N típusú MOSFET segítségével egy automatikus RESET áramkört is használunk, amelyek programozáskor a megfelelő BOOT módba rakják a mikrokontrollert.

3.3.3 SD kártya foglalat

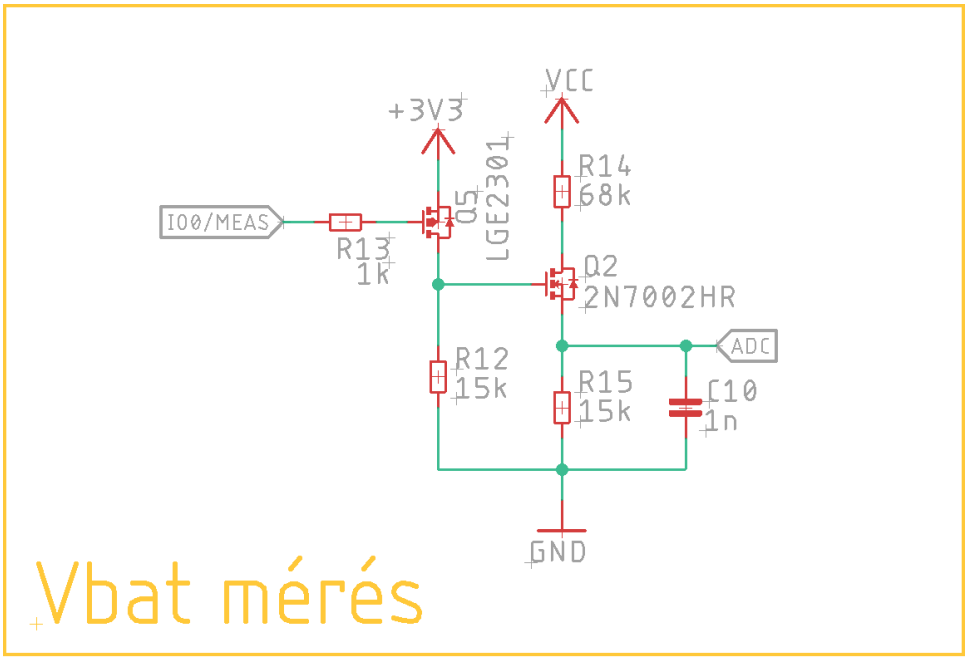


7. ábra: Az SD kártya foglalata

Ahogy a 7. ábrán látható, az SD kártya foglalat egy kifejezetten egyszerű áramköri elem. Csupán két dologra kellett figyelnünk:

1. Mivel az SD kártya egy aktív eszköz, ezért megfelelő tápszűrést (hidegítést) kell biztosítani számára.
2. Az adatvonalaknak nem szabad lebegnie, ezért felhúzó ellenállásokat kell alkalmaznunk.

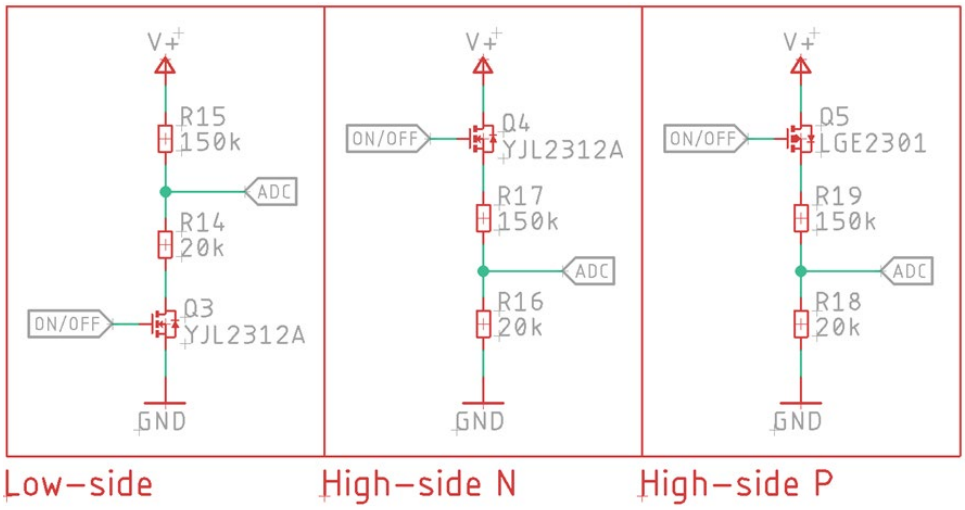
3.3.4 Feszültségosztó



8. ábra: A tervezett kapcsolható feszültségosztó

Ha szeretnénk tudni az akkumulátor töltöttségét, szükségünk van egy analóg bemenetre. Az MCU beépített szukcesszív approximációs analóg-digitális átalakítójának a maximális feszültségszintje 1V, tehát a mérőáramkör kimenete a körülbelül 4,2V-os feszültség esetén is ez alatt kell, hogy maradjon. Ehhez értelemszerűen valamilyen

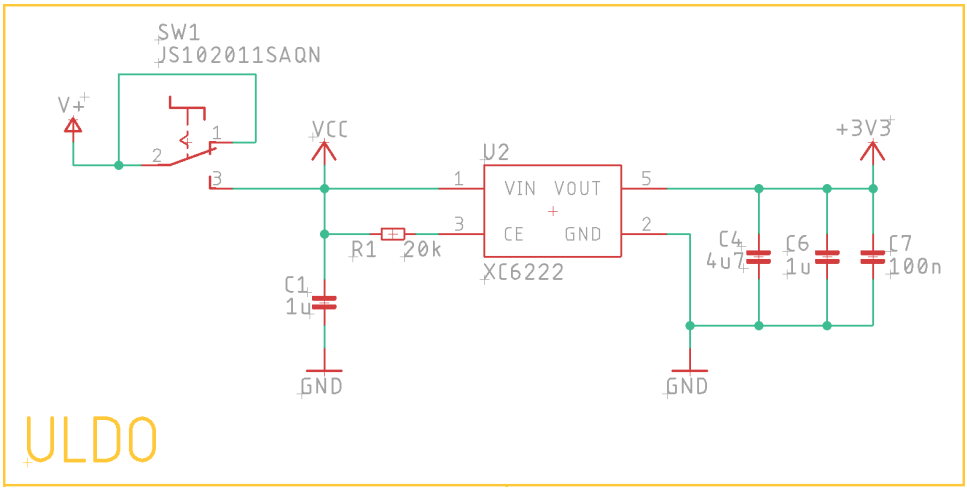
ellenállásosztót kellett terveznünk. Itt szerettük volna, ha az ellenállásosztó impedanciája nem terhelné folyamatosan az akkumulátort, ezért egy olyan kapcsolt osztót terveztünk, amely kikapcsolt állapotban nem fogyaszt áramot, valamint feszültsége nem terheli túl az ADC bemenetét.



9. ábra: Kapcsolható feszültségosztó topológiák

Low-side kapcsolás esetén kikapcsolt állapotban az ADC bemenetére akár 4,2V is kerülhet, amely meghaladja az adatlapban megadott 1V-os értéket. High-side kapcsolásnál N-FET esetén nem tudnánk teljesen bekapcsolni a tranzisztort, P-FET használata esetén pedig a kikapcsolt állapothoz magasan kellene tartanunk az ON/OFF lábat, ami a használt P típusú FET esetén akár 100nA-es szivárgási áramot is jelenthetne. Emiatt a 9. ábrán látható topológiák közül egyik sem alkalmas a feladatra. Végül a 8-as ábrán látható átmeneti megoldást választottunk. Itt még teljesen be tudjuk kapcsolni a FET-et, valamint kikapcsolt állapotban is csak a FET csatornájának szivárgási árama folyik, ami ezen a feszültségen a nA-es tartományban lesz.

3.3.5 Feszültségszabályzó, egyéb elemek

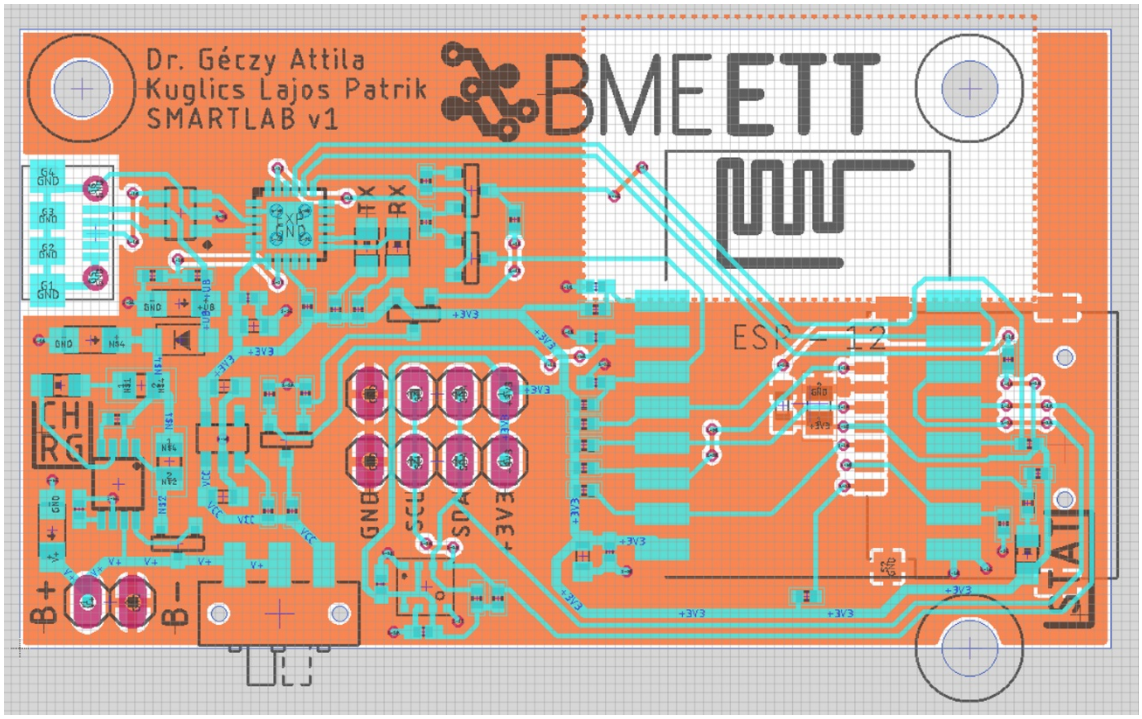


10. ábra: A feszültségszabályzó rajza

Az XC6222-es stabil működéséhez a gyártó 1µF-os kerámia kondenzátort ír elő a bemeneten. A kimenetre 2,5-5V-os modellek esetén legalább 2,2µF-ot ír elő. Nagyobb kondenzátorok használata esetén megnő a szivárgási áram: emiatt nem túl szerencsés túlméretezett kondenzátort tervezni a kimenetre, de stabilitás szempontjából mindenképp érdemes túllőni a gyártói ajánlásokon. Ezen mérlegelés után egy 4,7µF-os kimeneti kondenzátor mellett döntöttünk. A 10. ábra jobb oldalán látható 1µF-os és 100nF-os kondenzátor a mikrokontroller tápszűréséért felel.

Az áramkörben a fentieken túl, valamint a mikrokontrolleren kívül található még egy csúszókapcsoló is, valamint két csatlakozó, amelyeken keresztül a felhasznált BME680-as modulokat csatlakoztathatjuk.

3.4 NYÁK-tervezés



11. ábra: Az elkészült nyákterv

A kapcsolási rajz megtervezése után a nyákterv elkészítése következett. Ez egy összetett, iteratív folyamat. Legelső lépésben el kellett készítenünk az eddig nem használt alkatrészek lábkiosztását az adatlapjuk alapján. Ezután leellenőriztük a gyártásra kiszemelt cég előírásait a rétegfelépítésre és a különböző méretbeli határértékekre, mint például minimum huzalvastagság, minimum távolság a vezetők között, minimum furatátmérő stb. Ezeket az értékeket a nyáktervező szoftver Design Rule Check, azaz tervezési szabályellenőrzőjébe kellett bevinni; így mehettünk biztosra, hogy az elkészült nyáktervet a gyártó el is tudja majd készíteni.

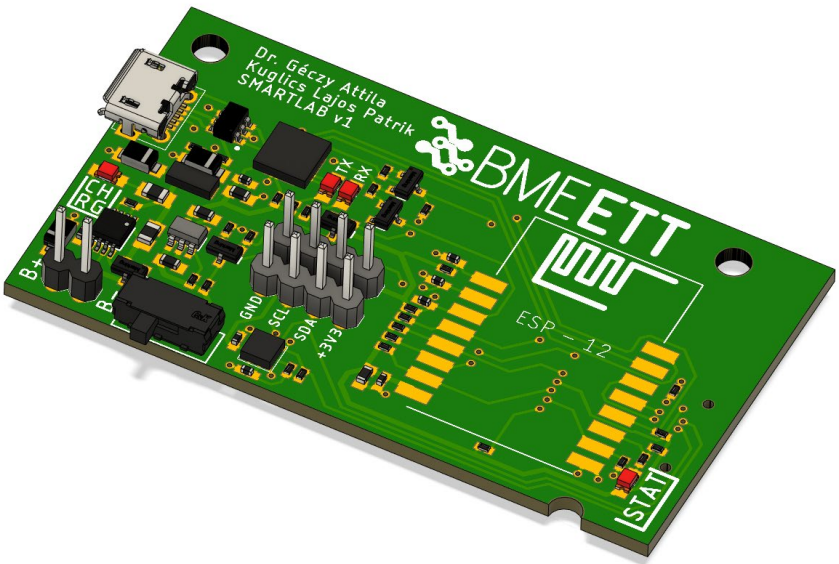
Ezután következett a tervezés neheze: az alkatrészeket a részegységek szerint csoportosítva elhelyeztük a nyákon. Először a csatlakozók, a csúszókapcsoló, valamint a mikrokontroller került a helyére, majd ezután következett a többi alkatrész csökkenő prioritás szerint. Ezután kezdtük csak meg a huzalozást. Itt a legoptimálisabb eredmény érdekében gyakran fel kellett bontani a huzalozást, átrendezni az alkatrészeket, majd az adott részt újrakezdeni. Természetesen a vezetékezésnek is megvan a maga sorrendje: legelőször a tápvonalakat huzaloztuk, majd ezután következtek a kommunikációs vonalak, végül pedig az alacsony sebességű ki- és bemenetek. Ezt azért ebben sorrendben kell elvégezni, mert nagyfrekvenciás áramkörök működése szempontjából a tápellátás a legkritikusabb: egy jól működő PDN (Power Delivery Network) nélkül akár véletlenszerű

újraindulásokat, nem definiált viselkedést is tapasztalhatunk. Ezután a nagysebességű kommunikációs vonalak következtek: mivel ők előbb kerültek a tervre, mint alacsony sebességű társaik, ezért prioritásuk volt a hely szempontjából; ezzel biztosítottuk, hogy a lehető legrövidebb úton jussanak célba a lehető legkevesebb átvezetés használatával.

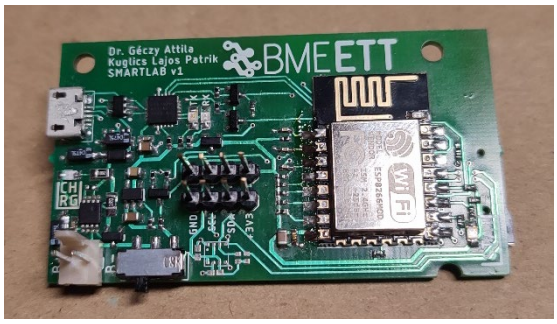
A tervet kettő rétegen valósítottuk meg. Ebből az alsó réteg a lehető legnagyobb mértékben földkitöltés. Ezzel biztosítjuk, hogy minden jelvezeték közelében van egy referencia sík, ami kevesebb EMC-zavar sugárzásával, kisebb EMC érzékenységgel és jobb jelintegritással jár. Nagyon fontos azonban, hogy a modul antennája alatt ne legyen semmilyen vezető és földkitöltés, hiszen azzal leárnyékolnánk az antennát. Ezt a kialakítást a 11. ábrán láthatjuk.

Miután a funkcionális részekkel végeztünk, a szitaréteg segítségével megjelöltük a töltésjelző LED-et, a csatlakozókat, a programozás és kommunikáció tesztpontjait, valamint a tervező nevét és a készítés dátumát. Ezután a nyáktervező programból 3D tervezőbe exportáltuk a tervet, így térben is ellenőrizni tudtuk az alkatrészek elhelyezését, a szitaréteg kinézetét. Ez a lépés kifejezetten fontos, hiszen a segítségével olyan hibákat is észrevehetünk, amelyeket 2D-s nézetben a különböző rétegek rengetegéből nagyon nehéz kiszűrni.

A 12-es és 13-mas ábrán látható a PCB 3D-s modellje, illetve az összeforrasztott áramkör: végső mérete 61x35mm^2 lett, amelyhez csak a két BME680-as modul (20x20x2,75 mm^3), valamint az akkumulátor csatlakozik.



12. ábra: Az elkészült node 3D-s modellje



13. ábra: Az elkészült szenzoráramkör

3.5 A beágyazott szoftver fejlesztése

Az ESP8266 programozásához az Arduino fejlesztőkörnyezetet használtuk. Ennek legfőbb előnye a nagy mennyiségű, gyorsan és ingyenesen beszerezhető könyvtárak, valamint a hozzájuk tartozó példakódok.

A szoftver megírása előtt egy blokkvázlatot terveztem, amely tartalmazza az egyes lépéseket, feltételeket. Ez a vázlat a 14. ábrán látható. A szoftver legfőbb összetevője a Bosch BSEC (Bosch Sensortec Environmental Cluster) könyvtár, amely bár zárt forráskódú, de ingyenesen elérhető [22]. Feladata a szenzor által mért adatok (hőmérséklet, páratartalom, nyomás, hotplate ellenállás) feldolgozása. Használatával elérhetővé válik az IAQ (Indoor Air Quality) index, amely egy 0-tól 500-ig terjedő skála: a magasabb számok rosszabb levegőminőséget jelentenek.

A szenzorral I2C interfészen keresztül kommunikálunk, az általa mért összes változót kiolvassuk belőle. Ezeket a változókat a BSEC könyvtár feldolgozza, majd a szenzoráramkör egyedi azonosítóját, az akkumulátor feszültségét, valamint a feldolgozott adatokat átküldjük az adatgyűjtő számítógépre, és elmentjük az SD kártyára is (amennyiben az csatlakoztatva van).

3.5.1 Mélyalvás

Az alacsony fogyasztás érdekében a mikrokontroller minden olvasás-küldés ciklus után alvó módba kapcsol. A program lefutása után a mélyalvás hosszát egy időzítővel állíthatjuk be. A BME680-ból üzemmódtól függően 3 vagy 300 másodpercenként kell adatot lekérni. Mivel a program futása változó hosszúságú, valamint két ciklus időtartama nagyban eltérhet (főleg a WiFi kapcsolat kialakulásának időigénye miatt), ezért az alvás hosszát a futásidő alapján dinamikusan állítjuk.

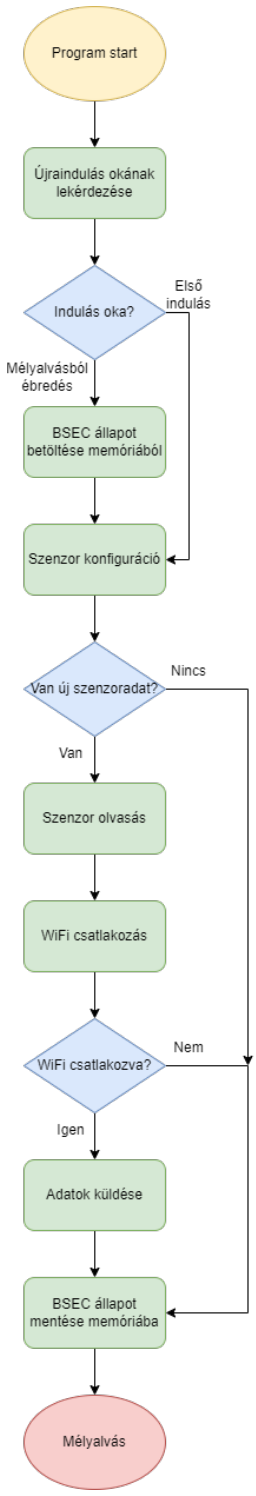
3.5.2 Vezeték nélküli kapcsolat

A lehető leggyorsabb WiFi csatlakozás érdekében az eszköz nem DHCP kliensként csatlakozik: a mikrokontroller statikus IP címmel rendelkezik, valamint a programban definiálva van a hálózati maszk és az átjáró is. Ezzel a körülbelül 3 másodperces csatlakozási időt lerövidítettük fél másodperc alá. Azért, hogy ezt még jobban lecsökkentsük, indulás után a WiFi hálózatokat nem szkenneljük végig: SSID helyett a hálózat MAC-címét keressük az előre beállított rádiófrekvenciás csatornán. Így a csatlakozási idő körülbelül 180-200ms körül alakul. Ehhez azonban több egyedi beállítást is kellett alkalmazni a WiFi elérési ponton. Mivel a WiFi hozzáférési pont MAC-címét le kell fixálni a kódban, ezért annak cseréje esetén a node-ok szoftverét is frissíteni kellene.

3.5.3 BSEC könyvtár

Mikrokontrollerek esetében az időzítés általában egy belső számláló feladata. A számláló a mikrokontroller működési frekvenciája alapján adott órajel-ciklusonként megnövel egy változót. Ennek a változónak az értéke stabil belső órajel esetén jól használható időzítési feladatokra. Arduino környezetben elérhető számunkra egy millis() függvény, amely a mikrokontroller indulása óta eltelt milliszekundumokat adja vissza.

A BSEC könyvtár használatának alapfeltétele az idő megtartása két mérés között. A szenzor kiolvasásakor alapesetben ezt az értéket használja az algoritmus, majd megadja a következő meghívás optimális idejét. Az adatfeldolgozó algoritmus minden olvasás után egy új állapotot generál magának, amelyben elmenti az utolsó olvasás idejét, valamint a következő meghívás idejét is. Amennyiben az utolsó olvasás ideje és a következő meghívás ideje közötti különbség túl nagy mértékben eltér az ideálistól (BME680 esetén ±50%), az algoritmus alaphelyzetbe állítja az IAQ kimenetét. Ezért nagyon fontos, hogy a helyes



14. ábra: A szenzoráramkör blokkvázlata

kimenethez a szenzort minél pontosabb időközönként hívjuk meg (például az 1000, 4000, 7000, 10000ms időpontokban).

Sajnos mélyalvás után az ESP8266 minden belső időzítője nullázódik. A megoldás a beépített RTC (Real-Time Clock) lehetne, azonban a mélyalvás időzítéséhez ezt a perifériát használjuk már. Ezért egy trükkös megoldást kellett alkalmaznunk:

1. A program első indulásakor kiolvassuk a szenzort, amely után kapunk egy időpontot a következő kiolvasási idővel.
2. A következő kiolvasási időpontot, valamint az algoritmus állapotát elmentjük EEPROM-ba (amelynek feladatát ESP8266 esetén valójában a flash memória látja el).
3. Az adatküldés és -mentés után mélyalvásba megyünk. A mélyalvás hosszát a következő időpont, az utolsó lefutási időpont, valamint a program futásának hossza alapján számoljuk ki. Például: a szenzort 100ms-nál olvastuk ki, a következő kiolvasás időpontja 3100ms-nál lesz. Az olvasás és az adatküldés 500ms alatt futott le. Ahhoz, hogy pontosan jókor fusson le a következő kiolvasás, 3100-100-500 milliszekundumot kell aludnunk (mínusz a kiolvasáshoz szükséges idő, ami körülbelül 70ms).
4. Ébredéskor visszaolvassuk memóriából az algoritmus állapotát, valamint a következő futási időpontot, majd ezt az értéket adjuk át a szenzoralgoritmusnak. Ezután a 2-es pontra lépünk.

4 Adatgyűjtő számítógép

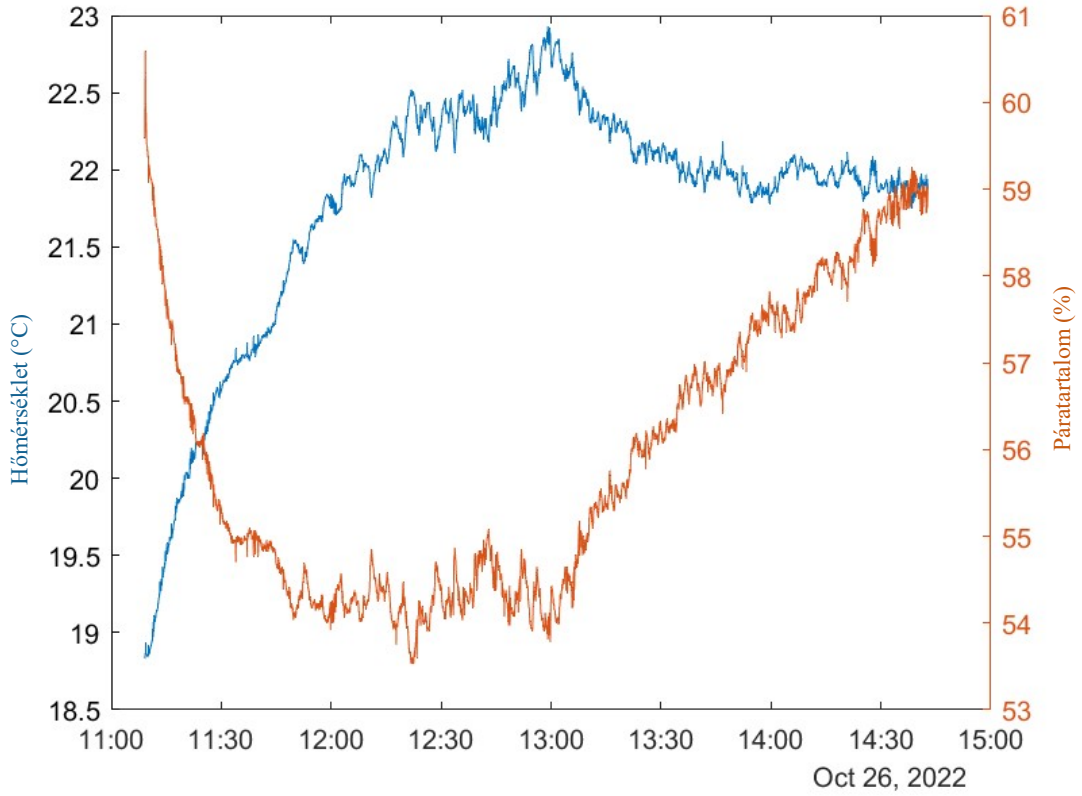
A mikrokontroller által küldött adatokat egy számítógép fogadja, dolgozza fel és tárolja el. Ezt a feladatot eredetileg egy Raspberry Pi SBC (Single-Board Computer) látta el, azonban ezek SD kártyáról működnek, így korlátozott hellyel és élettartammal rendelkeznek (az SD kártya elhasználódása miatt); ezen problémák miatt egy asztali számítógépre váltottunk.

4.1 Adatfogadás

Az adatok a mikrokontrollerből a hálózat útválasztóján keresztül érkeznek a számítógépbe. Az adatküldéshez az MQTT protokollt használtuk, amely alacsony sávszélesség-használatra, valamint sok-az-egyhez/sok-a-sokhoz kapcsolatok működtetésére specializálódott [23]. Ehhez a számítógépen egy MQTT kiszolgálót kell futtatnunk, amelyhez a mikrokontroller kapcsolódni tud.

Az MQTT kiszolgálón különböző “csatornák” vannak, amelyekre a kliensek adatokat tudnak küldeni, valamint onnan adatokat tudnak olvasni. A node itt egy csatornára publikál, a fogadó program pedig erre a csatornára iratkozik fel. A program Python nyelven íródott, képes az adatok fogadására, valamint azok feldolgozására, adatbázisba mentésére. Az adatbázisból így később könnyedén készíthetünk a 15. ábrához hasonló grafikonokat.

Adatok érkezésekor az első feladat az adatfolyam átkonvertálása bitfolyamból karakterekké. Ezután a beérkezett adatokat szétválasztjuk az előre meghatározott delimiter karakter alapján, amely esetünkben a pontosvessző karakter. Mivel a szenzorból a feszültséget a nyers ADC érték formájában küldjük át, ezért azt át kell számolnunk voltba. Az adatokat ezután elmentjük a helyi adatbázisba az aktuális dátum és idő kíséretében, így pontosan tudjuk melyik mérések mikor készültek.



15. ábra: A szenzoráramkör által mért értékek

4.2 Webszerver

Az adatbázis és az adatvizualizáló kliensek között a számítógépen futó webszerver teremt kapcsolatot. Mivel az adatbázisok nyílt hálózatra való megnyitása rengeteg adatvédelmi problémát okozhat, ezért az okostelefonon futó applikáció nem tudja direktben lekérni az adatokat az adatbázisból. Ehhez egy köztes réteget kellett implementálnunk egy webszerver formájában.

4.2.1 Django projekt

A Django egy Python alapú keretrendszer, amelynek segítségével gyorsan és hatékonyan tudunk webszervereket programozni. Egyik legnagyobb előnye, hogy minden benne készült projekt rendelkezik egy adminisztrációs felülettel, ami nagyban

megkönnyíti a fejlesztést. A Django projektek részegységeit applikációknak hívjuk. Ezek különálló elemek, amelyek modellekből (adatszerkezetek) és weboldalakból állnak. A mi esetünkben egy ilyen applikáció elég a teljes funkcionalitáshoz [24].

Az applikáción belül definiálnunk kell az adatstruktúránkat. Ehhez a 16-os ábrán látható modell "mezőket" kell használnunk, amelyek meghatározzák az adatbázisban használt típusukat is (például lebegőpontos szám, pozitív egész szám, karaktertömb stb.). Miután ezt elvégeztük, a Django menedzser-programja ezen mezők alapján létrehoz egy táblát az adatbázisban, amelybe a fogadó Python szkript el tudja helyezni az adatokat.

```
class bmedata(models.Model):
    clientid = models.CharField("Node ID", max_length= 20)
    voltage = models.FloatField("Akksifeszültség [V]")
    rawtemp = models.FloatField("Hőmérséklet (nyers) [°C]")
    pressure = models.FloatField("Nyomás [hPa]")
    rawhumidity = models.FloatField("Páratartalom (nyers) [%]")
    gasresistance = models.FloatField("Ellenállás [Ohm]")
    iaq = models.FloatField("Indoor Air Quality Index (dinamikus)")
    iaq_accuracy = models.FloatField("IAQ pontosság")
    temperature = models.FloatField("Hőmérséklet [°C]")
    humidity = models.FloatField("Páratartalom [%]")
    static_iaq = models.FloatField("Indoor Air Quality Index (statikus)")
    co2_eq = models.FloatField("ekvivalens CO2 [ppm]")
    bvoc_eq = models.FloatField("ekvivalens bVOC [ppm]")
    timestamp = models.DateTimeField('Időpont', auto_now = True)
```

16. ábra: Az adattípusok megadása a Django applikációban

A projekten belül egy központi fájlban definiálhatjuk a webszerverhez tartozó webcímeket. Itt megadhatunk:

1. fix, statikus címeket
2. dinamikusan változó címeket, amelyek képesek paramétereket fogadni
3. reguláris kifejezéseket

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('json/<slug:clientid>/<int:rows>', views.json2),
    path('json/<int:rows>', views.json)
]
```

17. ábra: A webszerveren található lapok címei

A 17-es ábrán látható egy statikus cím ('admin/', amely az adminisztrációs felületre vezet), valamint két dinamikus cím. Ezeket a beérkező változók típusai szerint tudjuk megkülönböztetni. Az első dinamikus cím a 'json/{szöveg}/{szám}' formátumot követi, a második pedig a 'json/{szám}' formátumot.

Amennyiben a webszerverre egy olyan kérést intézünk, amely megfelel valamelyik sablonnak, akkor az abban a sablonban definiált “nézet” fogja megválaszolni a kérést. Például az ‘szerver.hu/szöveg/1234’ kérésre a ‘views.json2’ nézet fog válaszolni. A dinamikus címek legnagyobb előnye, hogy a nézetek felhasználhatják a megkapott paramétereket: a fenti példát folytatva a 18-as ábrán látható ‘views.json2’ nézet a válaszüzenet összeállításakor a ‘clientid’ változóból a “szöveg” értéket, a ‘rows’ változóból az 1234 értéket ki tudja olvasni.

Ezek a nézetek nem csak statikus HTML oldalakat tudnak visszaküldeni, hanem dinamikusan felépített válaszokat is össze tudnak állítani adatbázisok, képek, egyéb erőforrások segítségével. Így egyetlen nézettel tudunk több kérést is definiálni: a kliens által kért oldal címe megadja, hogy melyik szenzor adatait kéri, és abból hány sort szeretne; ezen változók alapján egyedi SQL kérést tudunk intézni az adatbázis felé. Ez különösen hasznos lehet a mi esetünkben, hiszen egy Android kijelzőjén a megjeleníthető adatok száma nagyban függ a képernyő méretétől: egy előre beállított, túlméretezett darabszám helyett mindig csak annyi adatot kell lekérnünk, amennyire szükségünk van. Ezen felül minden, a rendszerhez hozzáadott szenzoráramkör adata elérhető a már megírt függvények segítségével, így gyorsan és egyszerűen tudjuk bővíteni a szenzorok számát.

```
def json2(request, **kwargs):
    rows = kwargs['rows']
    clientid = kwargs['clientid']
    query = f"SELECT * from (SELECT * FROM main_bmedata WHERE clientid = \"{clientid}\" order by id desc limit {rows}) nodedata"
    data = bmedata.objects.raw(query)
    data1 = serializers.serialize('json',data)
    return HttpResponse(data1, content_type="application/json")
```

18. ábra: A dinamikus lekérés nézete. Jól látható, hogy a beérkező paraméterek alapján formázzuk az SQL lekérdezést.

4.2.2 JSON

A webszerverek és Android applikációk közötti kommunikációra az ipari sztenderd a JSON adatformátum. A JSON egy könnyen írható, olvasható és átlátható szöveg alapú adatcsere-formátum. Egyszerűen generálható és elemezhető kódot takar, amely a JavaScript programozási nyelven alapszik. Ez a formátum programozási nyelvtől független; ezen tulajdonsága miatt hamar népszerűvé vált, mint adattovábbítási eszköz. A JSON-t tömbökből és/vagy objektumokból építjük fel [25].

Egy JSON objektum név-érték párosok rendezetlen halmaza, amelyet kapcsos zárójelek határolnak. A név és az érték között kettőspont található, a név-érték párosokat

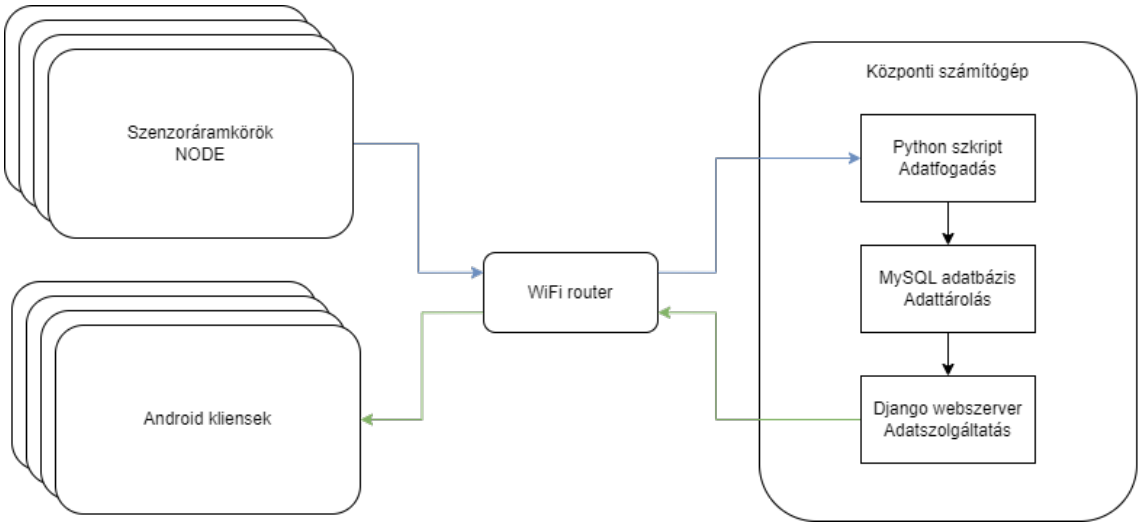
pedig vesszővel tagoljuk. A JSON tömb értékek rendezett halmaza. „[” karakter indítja, a tömb végét pedig a „]” karakter jelzi. Az értékeket vesszővel választjuk el.

A 19. ábrán egy példát láthatunk egy JSON tömbre, amelyben 2 objektum található, mindkettőben 2-2 név-érték párossal:

```
[ { “ID” : “1”, “NAME”: “Beijing” } , { “ID” : “2”, “NAME”: “Tianjin” }...]
```

19. ábra: A JSON adatformátum

Szerencsére a Django keretrendszer alapból tartalmaz egy olyan könyvtárat, amely képes SQL lekérdezések eredményeit JSON formátummá alakítani. Ehhez csupán egyetlen függvényt kell meghívnunk, amelynek bemenete az SQL lekérdezés eredménye, kimenete JSON formátumú szöveg. Ezek alapján a szenzoradatok útvonala a 20. ábrán látható.



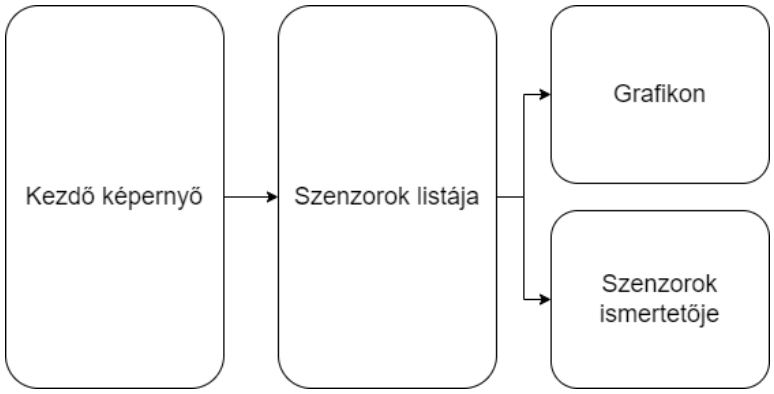
20. ábra: A projekt egyes részeinek kapcsolata

5 Android

Az alkalmazás egy Android operációs rendszert futtató tabletre készül. A tabletek nagy képátmérővel rendelkező eszközök, így egyszerre a grafikus felület több elemét is meg tudjuk jeleníteni a felhasználó részére. Ez különösen hasznos, hiszen a felhasználó egy pillantással ellenőrizni tudja az összes mért változót. A beépített vezeték nélküli internet-elérés, valamint alacsony súlyának és vékony felépítésének köszönhetően könnyen hordozható, bárhol elhelyezhető.

5.1 Navigáció megtervezése

A navigációs elveknek megfelelően az alkalmazás kezdő felülete szolgál majd a ki- illetve belépési pontként. A 21-es ábrán látható módon a kezdő felületről a szenzorok lista nézetére navigálhatunk, ami rendezett módon jeleníti meg a szenzorokat és adataikat. Innen nyílik lehetőség tovább navigálni a grafikonos, valamint a szenzorok ismertetőjét tartalmazó nézetre.

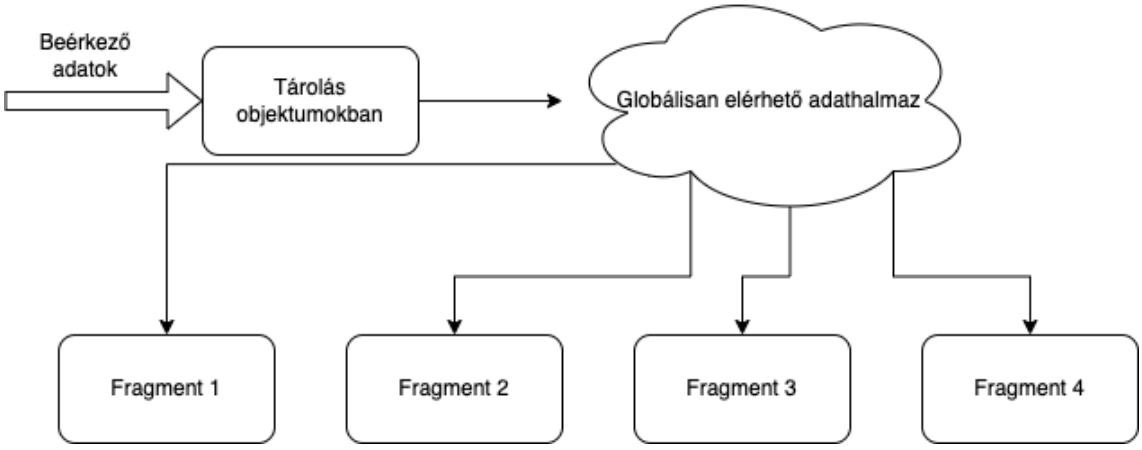


21. ábra: Navigáció megtervezése

5.2 Adatok áramlásának megtervezése

A JSON formátumban érkezett adathalmazt adatstruktúrák tárolására alkalmas osztályok segítségével objektumokba rendeztük. Egy olyan adattárolási módot kellett találnunk, amely képes a megjelenített felületek (*Activity* vagy *Fragment*) életciklusait

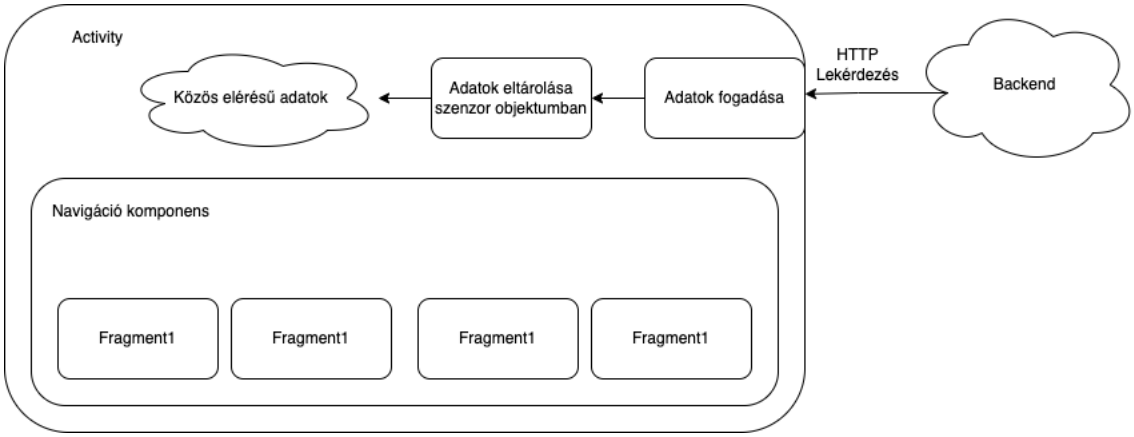
túlélni. Egy, az applikációban globálisan elérhető tárolási formára volt szükségünk: erre a *ViewModel* osztályt használtuk. Használatát a 22-es ábra szemlélteti.



22. ábra: Az adatáramlás terve

5.3 Moduláris felépítés

Szeretnénk, ha a lekérdezés ütemezését és lebonyolítását, illetőleg a teljes alkalmazást egyetlen *Activity* fogná össze. Ez megfelel a modern alkalmazás-fejlesztés elveinek, valamint megelőzi a kód-duplikálást is. Az *Activity* több *Fragment*et is tartalmaz a megtervezett navigáció alapján; ez a felépítés a 23-mas ábrán látható. Ezek a *Fragment*ek a globálisan elérhető adatokhoz való hozzáférést, illetve az így megkapott adathalmaz megfelelő megjelenítését implementálják.



23. ábra: Moduláris felépítés megtervezése

Hogy megelőzzük az adatokat tároló objektumok többszörös inicializálását a felületek közötti váltásoknál, terveztünk egy osztályt, amelyből egy adott szenzorra jellemző objektumot tudtunk példányosítani. Minden ilyen objektum egy-egy szenzort reprezentál és tárolja annak adatait. A mért adatokon felül tartalmazza a szenzor egyéb

jellemzőit, mint például: neve, mért változóinak mértékegységei és határai, fizikai helyzete, valamint rövid leírása. Ezen felül a szenzorhoz egy ikon is társítható. A további optimalizálás végett egy ilyen osztály a szenzorról több adatot is tartalmaz, mint tagváltozót. Ez a 24-es ábrán látható.

24. ábra: Szenzor osztály felépítésének terve

5.4 Fő funkciók

Az alkalmazás szempontjából a legfontosabb három funkcionalitás a kommunikáció megvalósítása, a kommunikáció ütemezése, illetve az adatok megjelenítése.

5.4.1 Kommunikáció megvalósítása

Az adatok megjelenítéséhez az Android eszköznek kapcsolatban kell lennie a fent megismert rendszerrel. Ennek a megvalósításához HTTP parancsokkal kommunikálunk a webszerverrel, ami JSON formátumban adja vissza a választ. Android oldalról a kapcsolatot a Retrofit2 könyvtár alkalmazásával oldottuk meg [26]. A kommunikáció megvalósításához alapul vettük Megyeri István szakdolgozatának releváns részét [27].

Elkészítettük a GET lekéréshez az interfészt, amelyben definiáltuk a paramétereit, mint a lekérdező függvény neve, vagy az adatok eléréséhez szükséges IP-cím. A felhasznált lekérés a http://{webszerveripcím}/JSON/smartlab_node_x/y, ahol az x adja meg, hogy melyik node adatait kérjük le, az y pedig a lekérdezett adatok darabszámát

definiálja. A visszakapott adatokat objektumokban tároljuk el, amelyhez az osztályt a JSONToKotlinClass nevű Android Studio bővítményből származtattuk [28].

5.4.2 Kommunikáció ütemezése

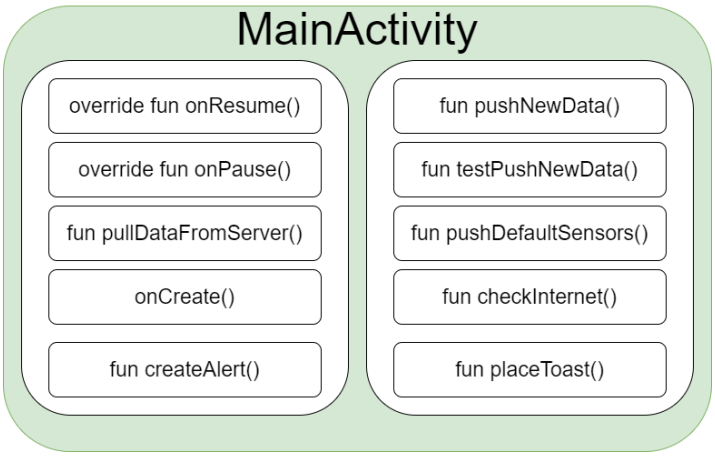
Hogy mindig a legfrissebb adatok álljanak rendelkezésre, a webszerverrel való kommunikációt bizonyos időközönként végre kell hajtani. Ehhez a fő *Activity*ben egy változtatható időzítő funkciót implementáltunk: a késleltetés paramétert milliszekundum mértékegységben lehet megadni. A lekérdezést try-catch blokkban valósítottuk meg, így a sikertelen lefutás esetén is tovább fut az applikáció. A ciklikus feladat lefutásának ellenőrzéséhez egy rendszerüzenetet helyeztünk el. Az „*onPause*" és az „*onResume*" függvényeket is felüldefiniáltuk, így más applikációba történő váltás esetén a ciklikus aktivitás szünetel; visszalépéskor folytatódnak az aktivitások.

5.4.3 Grafikonos megjelenítés

Az adatok megjelenítésének legkézenfekvőbb módja a grafikonos ábrázolás. Segítségével jól átlátható információt kaphatunk az aktuális levegőminőségről, valamint annak időbeli változásairól. A grafikonok implementálásához egy külső könyvtárat használtunk: ez az *MPCharts*. Segítségével részletesen kidolgozott grafikonokat lehet implementálni Kotlin vagy Java nyelven [29].

5.5 A Main *Activity* implementálása

A fő funkciók jelentős részét a Main *Activity* végzi el. Ide tartozik a ciklikus lekérdezés, az adatfeldolgozás, valamint az adatok eltárolása. A rendelkezésre álló adatok megjelenítése a felhasználó számára már a *Fragmentek* feladata.



25. ábra: A Main Activity függvényei

A Main *Activity* alkotja az applikáció gerincét, a 25-ös ábrán látható függvényei a fentebb említett fő feladatokat látják el. A következőkben bemutatjuk az egyes függvények feladatát:

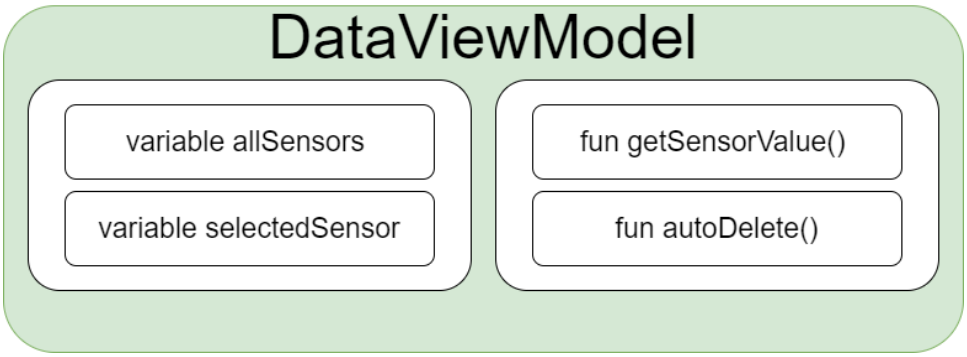
- Override fun onResume(): Ebben a függvényben kap helyet a ciklikus lekérdezést megvalósító időzítő. Azért volt szükség ennek a felüldefiniálására, mert csakis akkor szeretnénk lekérdezéseket végrehajtani amikor az *Activity* éppen látszódik a képernyőn.
- Override fun onPause(): Ebben a felülírt függvényben állítjuk le a ciklikus lekérdezést. Ez a függvény akkor fut le, amikor az alkalmazásunk a háttérbe kerül; ilyenkor nincs szükség az adatok frissítésére.
- Fun pullDataFromServer(): Ezt a függvényt az időzítő hívja meg ciklikusan. Ebben a függvényben implementáltuk a Retrofit2 könyvtár segítségével a http GET lekérést, illetve itt történik meg a JSON adatok objektumokba történő eltárolása.
- Fun pushNewData(): A pullDataFromServer() függvény hívja meg, amennyiben sikeresen megérkeztek az adatok a szerverről. A függvény feladata, hogy a megkapott objektumból a megfelelő tagváltozókat a megfelelő szenzorok adatait tartalmazó tömbbe elhelyezze. Ezután értesíti a *ViewModel*-t az adatok változásáról.
- OnCreate(): Ez a függvény fut le minden alkalommal amikor az *Activity*-ből egy új példány jön létre. Ez bekövetkezhet az alkalmazás újraindításakor, vagy a készülékünk elforgatásakor. Elhelyeztünk a függvényben egy internetkapcsolat-ellenőrzést: ennek hiányában nem is érdemes elindítani az alkalmazást. Ezt valósítja meg a “checkInternet” függvény, amelynek a createAlert() alfüggvénye felelős a hibaüzenet megjelenítéséért. Itt példányosítjuk a *ViewModel* objektumot, amelyet az alkalmazás további működése során végig használni fogunk.
- Fun pushDefaultSensors(): A függvény az onCreate()-ben fut le. Feladata a szenzor objektumok példányosítása a *ViewModel* objektum számára.
- Fun testPushNewData(): Egy, a debuggoláshoz használt függvény. Feladata, hogy a szenzoroknak példa adatokat szolgáltasson akkor is, amikor a szerver nem elérhető.

5.5.1 A navigáció létrehozása

Az applikációban történő navigáció a *Fragment*ek között az alábbi struktúra alapján történik. Az összesített és a grafikon nézet közötti váltást úgy oldottuk meg, hogy egyetlen csúsztatással váltani tudjunk a felületek között. Ezt egy külső könyvtár, a SlidingPaneLayout segítségével valósítottuk meg. Ebben az esetben a két *Fragment* szimultán létezik, viszont egyszerre mindig csak az egyik látható a képernyőn.

5.6 *ViewModel* és LiveData

Az adatok eltárolását a címben említett elemek segítségével szeretném megvalósítani. Ennek megfelelően létrehoztam a saját *ViewModel* osztályomat Data*ViewModel* név alatt, amelynek felépítése a 26-os ábrán látható.



26. ábra: A DataViewModel osztály felépítése

A Data*ViewModel* osztályomban létrehoztam egy „allsensors” nevű üres listát, amely egy szenzor objektumokat tartalmazó LiveData tömb. Ennek a tömbnek az adatait frissítjük lekérdezéskor. Létrehoztam egy másik LiveData minőségű “selectedSensor” változót: a típusa Integer és egyszerre egyetlen elemet tartalmazhat. Ez a változó mindig egy szenzor nevére mutat. Segítségével egy adott szenzor kiválasztásánál az alkalmazás minden *Fragment*je tudja, hogy melyik a megfigyelni kívánt szenzor. A Data*ViewModel* osztályban létrehoztam még két függvényt is. A getSensorValue() egy debug függvény, amely paraméterként egy Integer értéket vár, ez alapján pedig egy adott szenzor legutolsó kapott eredményét visszaadja. Az autoDelete() függvény a futásidőben eltárolt adatok mennyiségét kontrollálja: amennyiben az adatok mennyisége meghalad egy értéket, akkor a legkorábbi eredményekből töröl, biztosítva, hogy a memória ne teljen meg [30][31].

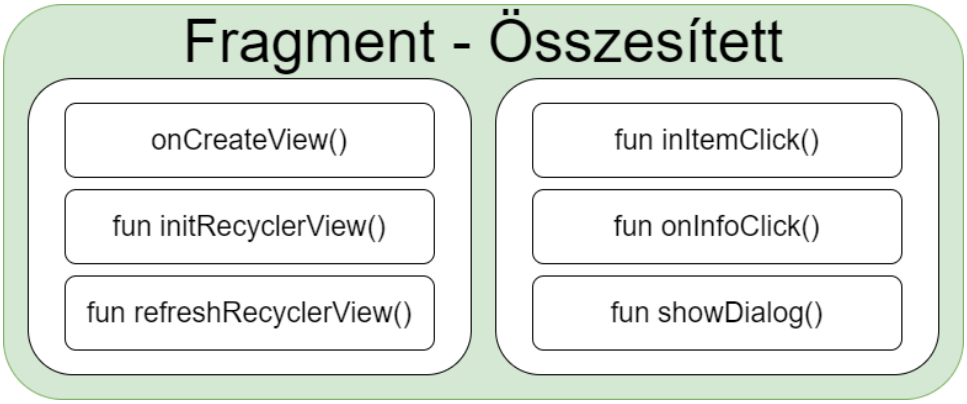
5.6.1 *Fragment*ek elkészítése

Az adatok megjelenítése *Fragment*ek segítségével történik. Ezeknek el kell érniük a *ViewModel* objektumot. Az elérés biztosításához *observer*eket fogunk használni, amit

a *ViewModel* objektum tagváltozóira csatolunk. Amikor az *observer*hez tartozó változó megváltozik, a felhasználói felületen annak megfelelően módosítható a megjelenítendő adathalmaz [32].

5.6.1.1 *Fragment*1 - Összesített nézet

A *Fragment* felépítését a 27-es ábra szemlélteti. Fontos építő eleme a RecyclerView, amelynek elemei egy-egy szenzor objektumot reprezentálnak a felhasználó számára.



27. ábra: Az összesített nézetet megvalósító Fragment függvényei

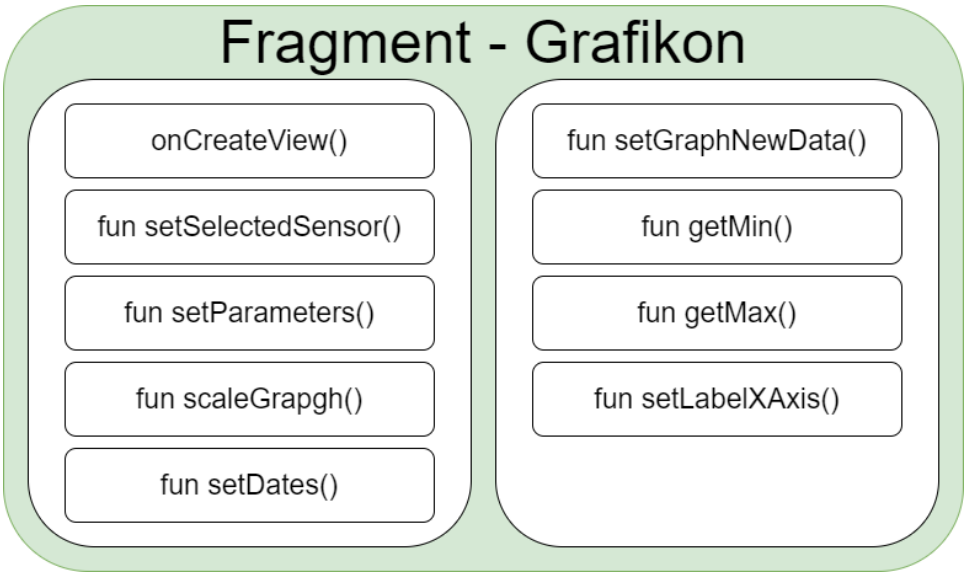
A *Fragment* funkcionalitását a következő függvények valósítják meg:

- OnCreateView(): A *Fragment* létrejöttekor ez a függvény fog elsőnek lefutni. Itt inicializáltuk a szükséges layout elemeket, illetve ebben a függvényben valósítottuk meg az *observer*t, amely a *ViewModel* objektum "allSensors" attribútumának változásaira reagál. Az *observer* függvényén belül történik a nézet elemek frissítése.
- Fun initRecyclerview(): A függvény feladata a RecyclerView példányosítása és paramétereinek feltöltése a szenzor objektumokból származó adatokból, amelyeket a *ViewModel* objektumból ér el.
- Fun refreshRecyclerView(): A függvény az *observer*en belül kap helyet. Amikor megérkeznek az új adatok, akkor a RecyclerView paramétereit megfelelően módosítja és frissíti a felhasználói felületet.
- Fun inItemClick(): A RecyclerView egy tagfüggvénye, amely az általa létrehozott elemekre való kattintás eseményre triggerel. A kattintás során az adott szenzor nevét eltárolja a *ViewModel* objektum "selectedSensor" változójában. Ez később lesz fontos a grafikonos megjelenítés implementálásánál.

- Fun onInfoClick(): A RecyclerView másik tagfüggvénye, amely minden szenzorhoz egy információs gombot rendel. Ennek megnyomásakor egy, a szenzor adatait mutató információs ablak jelenik meg.
- Fun showDialog(): A tervezéskor egy információs *Fragment* is helyet kapott, azonban a funkcionalitás integritását szem előtt tartva, az információs *Fragment* implementálható egy felugró ablak formájában. A függvény a szenzorok információs ablakának megjelenítéséért felelős. A megfelelő adatokat a *ViewModel* objektumból kapja meg.

5.6.1.2 *Fragment*2 – Grafikonos megjelenítés

A tervezés során választott másik megjelenítési forma az adatok grafikonban való megjelenítése. Ehhez a 28-as ábrán látható függvényeket használtuk fel.



28. ábra: A grafikon nézetet megvalósító Fragment függvényei

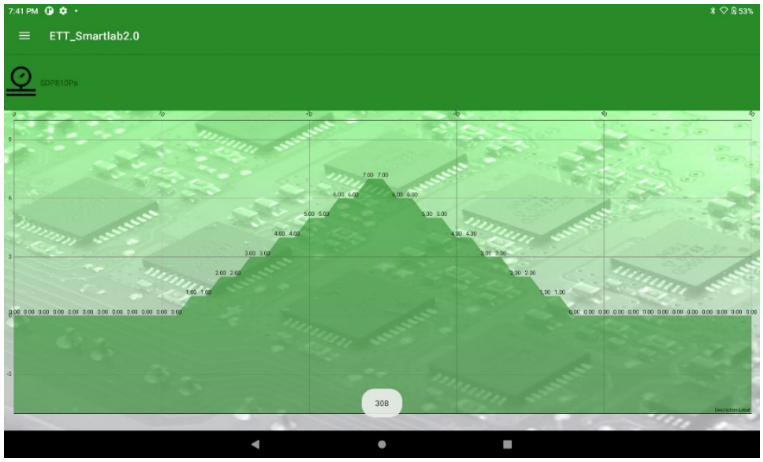
- OnCreateView(): Feladata a grafikon inicializálása és paramétereinek beállítása, továbbá két *observer*t is tartalmaz:
  - *Observer* a legfrissebb adatok eléréséhez a *ViewModel* objektum tagváltozóira csatolva. Minden alkalommal lefut amikor az adathalmazban változás történik.
  - *Observer* a *ViewModel* "selectedSensor" változójára csatolva, amit az összesített nézet *Fragment*en kattintással tud a felhasználó beállítani. Minden alkalommal lefut, amikor kattintás esemény történik, (tehát amikor a felhasználó szenzort vált).

- Fun setSelectedSensor(): A *ViewModel* objektumból lekérdezi a kiválasztott szenzor nevére mutató Integer értéket. Ezzel állítjuk be, hogy melyik szenzort jelenítsük meg a grafikonos nézetben.
- Fun setParameters(): Ez a függvény a grafikon kinézetének paraméterezését végzi el; ilyen például a megjelenítendő értékek száma, a grafikon vonalvezetésének vastagsága vagy a grafikon alatti terület kitöltésének színe.
- Fun setGraphNewData(): Akkor fut le, amikor új elem érkezik a grafikonba. A *Fragment* rendelkezik egy tömbbel, amelyben a megjelenítendő x-y értékpárokat tároljuk. A függvény feladata, hogy a legrégebbi adatokat kitörölje a tömb elejéről, a legújabb adatokat pedig elhelyezze abban.
- Fun getMin() és getMax(): A grafikon x-y érték pár adatokat tartalmazó tömbbel dolgozik. Végig iterál az elemeken, majd visszatér a legkisebb és legnagyobb értéket tartalmazó elemmel. Ennek a grafikon skálázásánál van fontos szerepe.
- Fun scaleGraph(): Feladata a grafikon skálázása a getMin() és getMax() függvények segítségével.
- Fun setDates(): Az x-tengely számára a megjelenítendő dátum értékeket tömbben tároljuk. A függvény feladata, hogy FIFO elv alapján kezelje a tömb elemeit.
- Fun setLabelXAxis(): Az felhasznált kódkönyvtár, az MPCharts egy függvényének a felüldefiniálása. Lehetővé teszi, hogy saját magunk által formázott elemeket jelenítsünk meg az x-tengely mentén.

6 Alkalmazás tesztelése

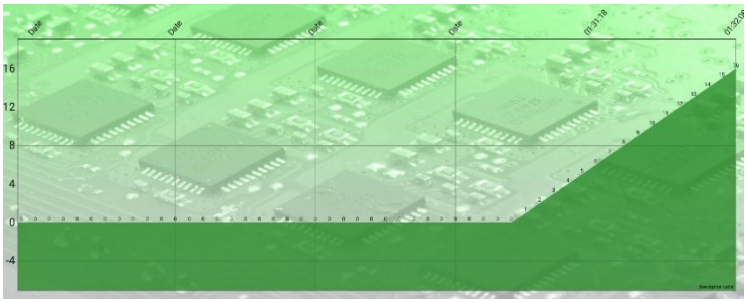
6.1 Offline tesztelés

Az alkalmazás tesztelését két részre osztottuk, hogy az esetleges hibák oka könnyebben kideríthető legyen. Először internetes kommunikáció nélkül, a Main *Activity*-ben előre megírt teszt függvény segítségével ciklikusan próba adatokat adtunk át a Data*ViewModel* objektumnak. Minden szenzorhoz egyedi adatot rendeltünk, így az is ellenőrizhető, hogy a váltások valóban megtörténnek. A tesztfüggvény minden egyes lefutásakor egy rendszerüzenetet küld, amely kiírja képernyőre, hogy hányadik sikeres lekérdezés zajlott le éppen. Ez látható a 29-es ábrán.



29. ábra: A szenzorok közötti váltás próba-adatokkal való tesztelése

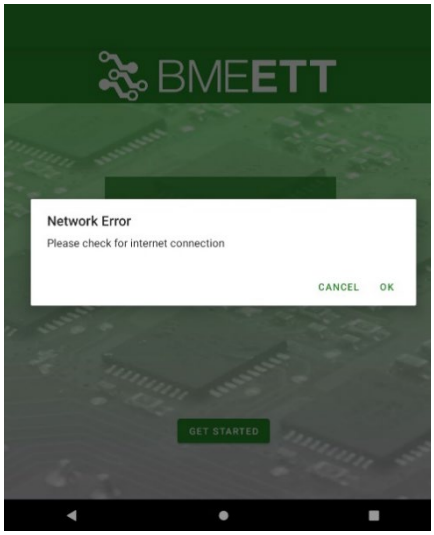
Az adatokat szolgáltató tesztfüggvényt úgy módosítottuk, hogy minden ciklus után eggyel nagyobb értéket adjon, ezzel tesztelve a grafikon automatikus méretezését. Ez a 30. ábrán látható.



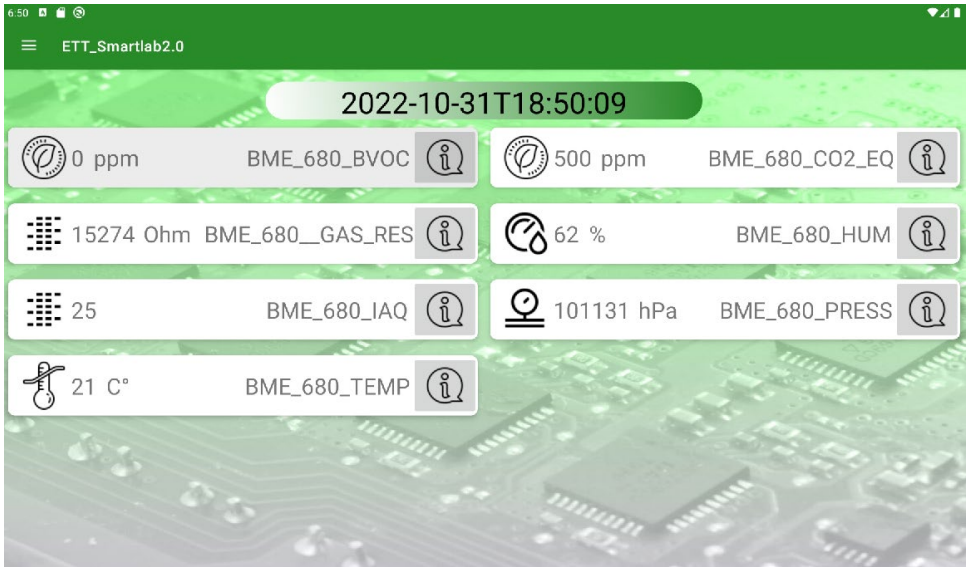
30. ábra: A grafikon tengelyeinek automatikus igazítása

6.2 Online tesztelés

Az első indításnál leteszteltük, hogy az applikáció figyelmeztet-e WiFi kapcsolat hiányára. Ehhez kikapcsoltuk a WiFi elérést a tableten. Indításkor a 31-es ábrán látható üzenet fogadott:

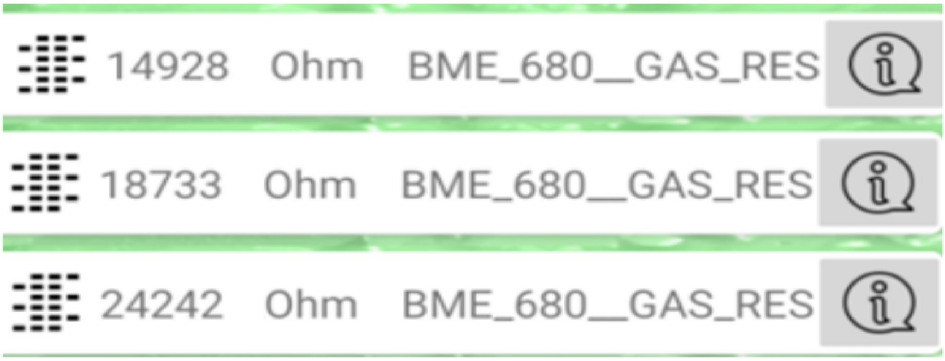


31. ábra: Hibaüzenet internet elérés hiányában



32. ábra: Képernyőkép a szenzorok összesített nézetéről

Amikor bekapcsoltuk a szenzoráramköröket, és elkezdtük figyelni az adatokat, a BME680 szenzoráramkör által mért értékek látványosan megváltoztak. A 32-es ábrán látható értékek már valós mérési eredmények. Az alábbi, 33-mas és 34-es ábrán látható képernyőkép-részletek 30 másodperces időközöket mutatnak.



33. ábra: Képernyőkép a szenzorok összesített nézetéről

34. ábra: Képernyőkép a szenzor adatairól grafikonban

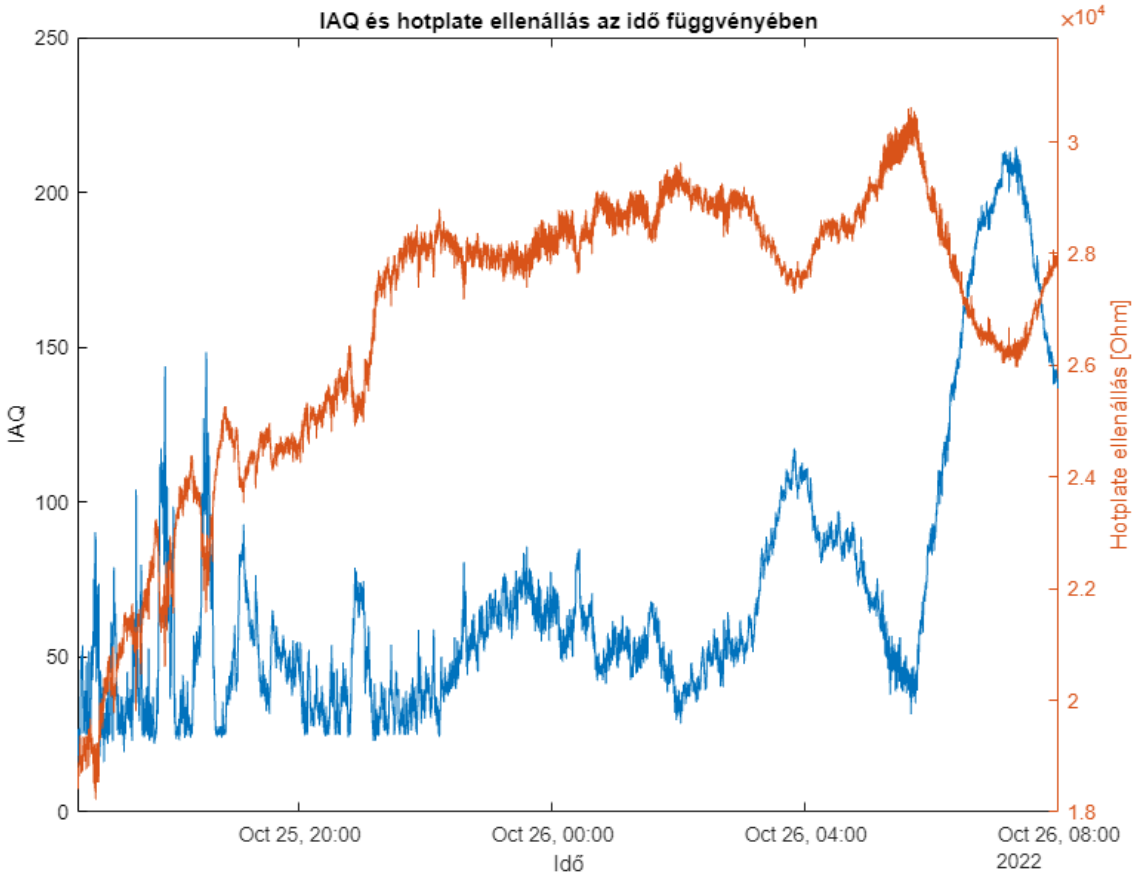
Amennyiben futásidőben megszakad az internetkapcsolat, új adatok hiányában az applikáció küld egy üzenetet a sikertelen kommunikációról. Az internetkapcsolat visszaállása után gond nélkül érkeznek tovább az adatok.

35. ábra: Képernyőkép az internet elérés nélküli állapotról

Az adatokat mérő node kikapcsolása után is hagytuk futni az alkalmazást. Ekkor azt tapasztaltuk, hogy folyamatosan ugyan azok az adatok érkeztek újra és újra, amely a 35-ös ábrán látható.

7 Teszteredmények

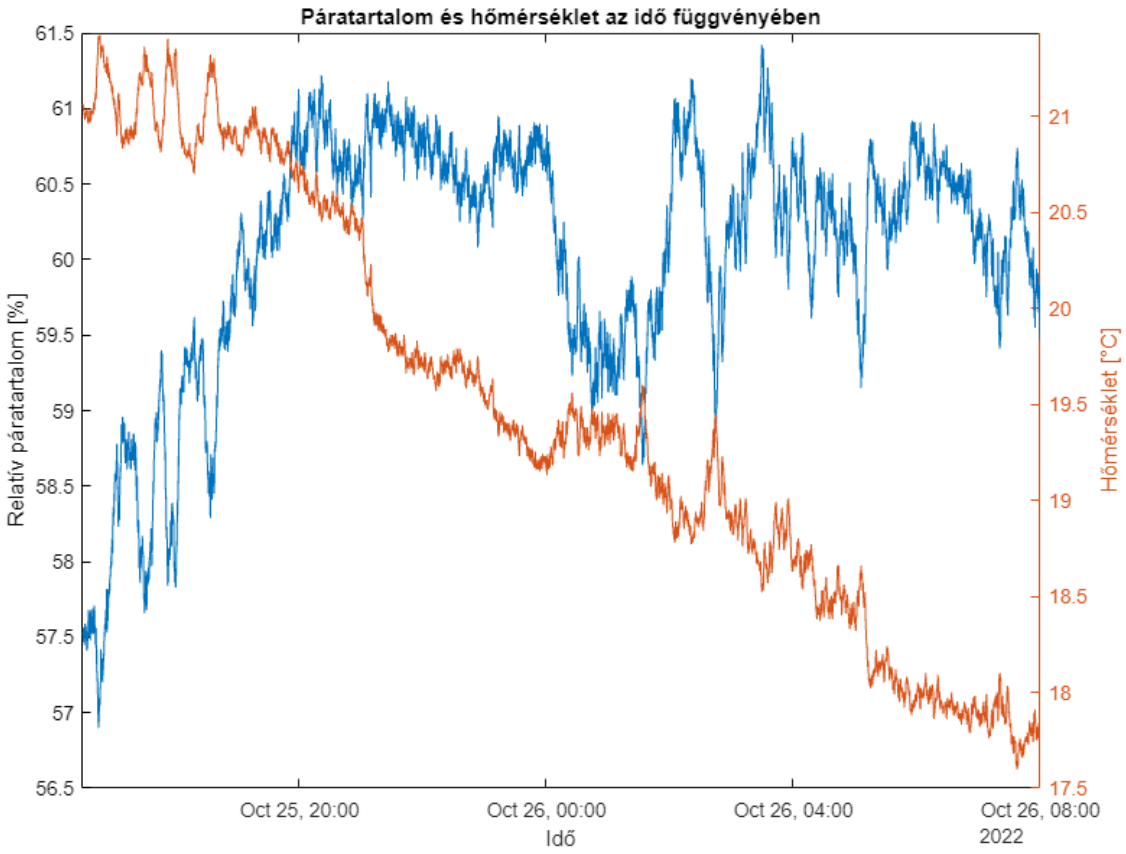
Alább pár grafikon látható, amelyeket a laboratóriumban mért adatokból állítottunk össze.



36. ábra: IAQ és hotplate ellenállás összefüggése

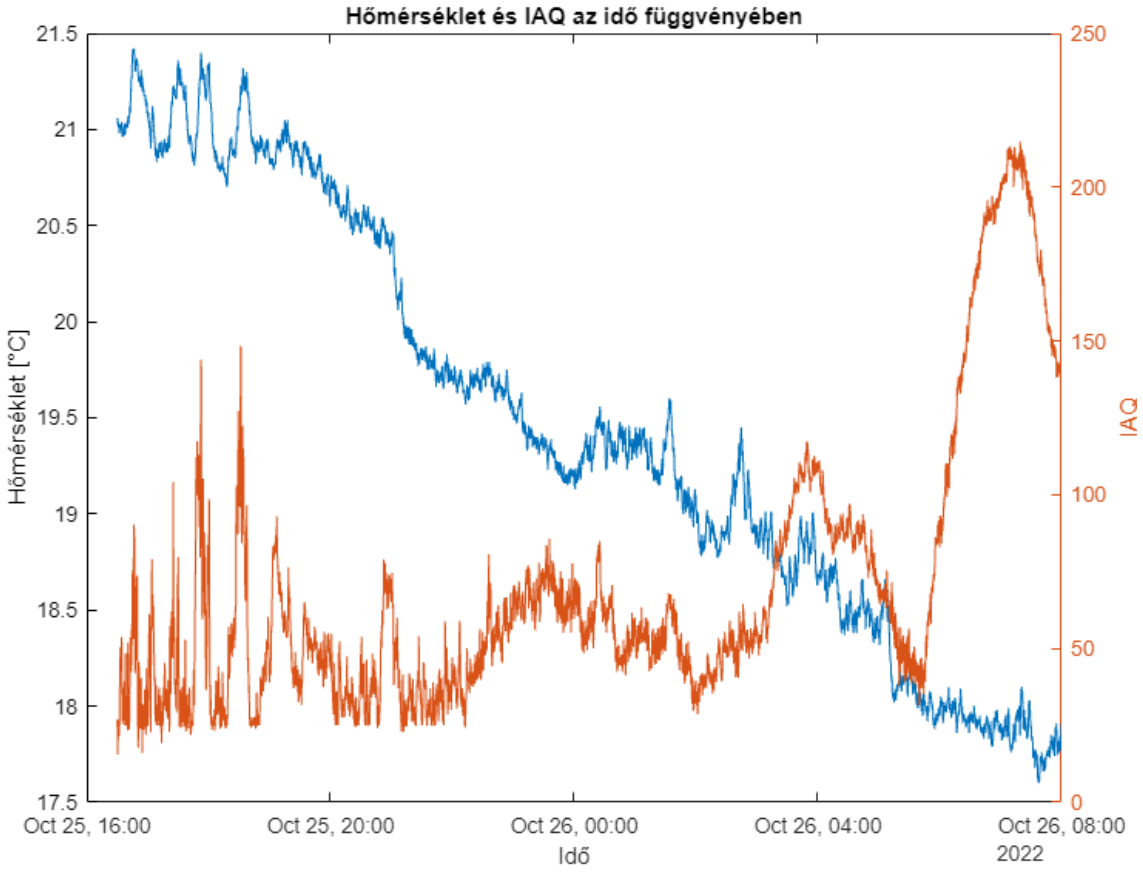
A 36-os ábrán az IAQ és a szenzor mérőellenállásának a kapcsolata látható egy grafikonon. Mint látható, az ellenállás csökkenése (VOC-k jelenléte miatt) esetén az IAQ érték jelentősen megnő.

A 37-es és 39-es ábrán kettő, azonos szobában elhelyezett szenzoráramkör adatait tudjuk összehasonlítani. Látható, hogy azonos trendeket követnek az egyes hőmérséklet és páratartalom adatok, azonban nem egyeznek meg teljes mértékben. Ezt magyarázhatja az egyes szenzorok közötti különbség, a különböző mintavételi időköz (az 1-es node esetén 3 másodperc, a 2-es esetén 5 perc), valamint a különböző fizikai elhelyezés a szobán belül (az 1-es node egy forrasztóállomás mellett lett elhelyezve, a 2-es node a laborszoba bejáratához). Hosszabb működés után majd lehetőségünk nyílhat a szenzorok részletes összehasonlítására, esetleges állandó hibáik kiküszöbölésére is.



37. ábra: Az 1-es node páratartalom és hőmérséklet adatai

Érdemes egy picit a mérések gyakorlati hasznát és a valós eseteket összehasonítani. A mérések elején (az első négy órában) körülbelül este 7 óráig működött a forrasztóberendezés, aminek köszönhetően a forrasztóberendezéstől körülbelül 1 m távolságra elhelyezett szenzorok képesek voltak a ciklusokat ~20.5 – 21.5 °C között érzékelni. A páratartalom ez idő alatt jelentősen nőtt, 20:00-24:00 között a zárt laborban nem történt ilyen szempontból változás, viszont az éjszakai dinamikus változásait valószínű a laboratórium ventillációs rendszere okozta. A hőmérséklet mérésével 2022 őszének-telének egyik nagy problémája lett monitorozva. A hőmérséklet 18 °C alá esett ahogy a kollégák esti távozása után reggel 8-ra lehűlt a laborhelyiség.

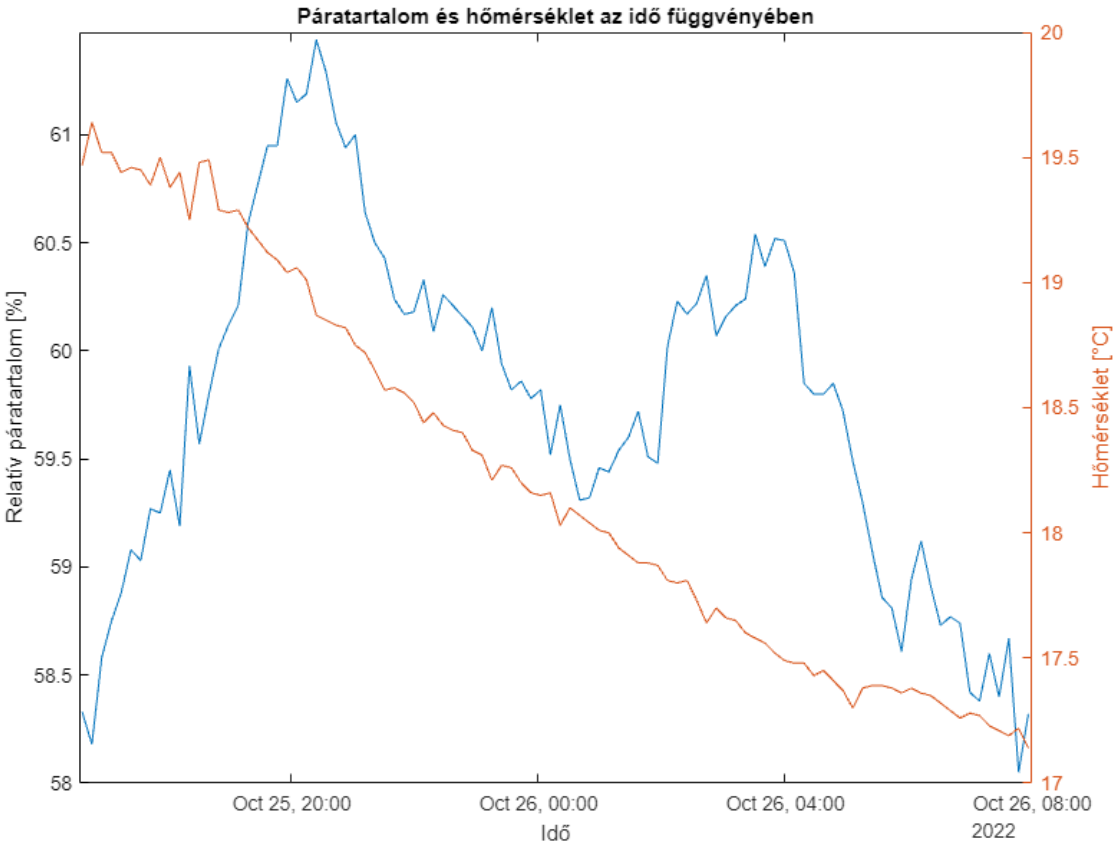


38. ábra: Az 1-es node hőmérséklet és IAQ görbéje

A 38-as ábra hőmérséklet- és IAQ görbéit figyelembe véve az látszik, hogy az 1-es node IAQ értékei a 16:00-18:00 közötti forrasztási ciklusnál egyértelmű tüskéket mutatnak, amelyek a forrasztókemence ciklusok során távozó anyagok (nem csak illékony szerves vegyületek, lásd 36. ábra) miatt lehetnek jelen, melyek a többi node-on nem voltak megfigyelhetőek. A 36. ábrán az illékony vegyületek távozása a hot-plate ellenállás értékeivel mutatható ki – érdekes módon ezek nincsenek teljesen összhangban az IAQ értékekkel. Ezek az értékek szintén összecsengenek az eredményeink munkavédelmi jelentőségével, vizsgálatuk további feladatokat képez a félév során.

Az IAQ görbe utolsó, hajnalban mért tüskéjére egyelőre nem találtunk magyarázatot, de ez szintén összefüggésben lehet az épület/laborkomplexum éjszakai automatikus szellőztetésével.

A következő ábrán látható simább jelalak a fentebb is írt mintavételi idő különbség miatt adódott a node 2-n.



39. ábra: A 2-es node által mért páratartalom és hőmérséklet grafikon

8 Kitekintés

Munkánk eredményeként egy olyan rendszert sikerült elkészítenünk, amely nagy mennyiségű adat összegyűjtésére, eltárolására, valamint kimutatására is alkalmas. Ezen adatok segítségével potenciálisan javíthatunk a laboratórium levegőminőségén, valamint növelhetjük az ott dolgozók komfortérzetét. A szenzorok számának, valamint típusának növelésével megbízhatóbbá tehetnénk a rendszert, így munkavédelmi szempontokat is elláthatna.

Az Androidos applikáció fejlesztésével, valamint a laboratóriumban dolgozó kollégák bevonásával akár egy „riasztórendszert” is létrehozhatnánk, amely időben figyelmeztetné az embereket a levegő minőségének romlására; ezáltal előbb és hatékonyabban meg lehet szüntetni a szennyező anyagok koncentrálódását.

A rendszerrel a laboratóriumi hőmérsékletek is vizsgálhatóak, amelyek következtében a 2022 téli szezon nehéz körülményei feltárhatóak és adott esetben a kollégák számára jelzést biztosíthatunk majd az egészségre már ártalmas munkakörülményeket illletően.

A munka előzményét tavaly egy IEEE konferencián publikáltuk [33].

9 Köszönetnyilvánítás

Szeretnénk megköszönni Dr. Géczy Attilának, konzulensünknek, hogy mindvégig hasznos tanácsokkal és ötletekkel segítette munkánkat, valamint lelkes hozzáállásával ösztönzően hatott ránk.

Szeretnénk megköszönni Dr. Ekler Péternek az Android szoftverfejlesztés című tárgy során tanúsított lelkes, ösztönző oktatói hozzáállását, ezzel biztosítva motivációmat és érdeklődésemet a terület iránt.

10 Irodalomjegyzék

[1] https://www.epa.gov/report-environment/indoor-air-quality (elérés:2022.10.30)

[2] https://www.epa.gov/pm-pollution/particulate-matter-pm-basics (elérés:2022.10.30)

[3] https://www.epa.gov/indoor-air-quality-iaq/what-are-volatile-organic-compounds-vocs (elérés:2022.10.30)

[4] https://www.theguardian.com/money/2019/feb/09/mdf-furniture-toxic-fumes-formaldehyde (elérés:2022.10.30)

[5] https://www.mouser.com/catalog/specsheets/Amphenol_12202018_AAS-916-139A-Telaire-SM-UART04L-AppNote-102018-web.pdf (elérés:2022.10.30)

[6] C.K.Laird, I.Verhappen (2010); Instrumentation Reference Book (Fourth Edition), Chapter 25 - Chemical Analysis: Gas Analysis; Pages 401-428

[7] J.F.DiMarzio; Beginning Android Programming with Android Studio (2017) ; ISBN: 978-1-118-70559-9

[8] https://kotlinlang.org/ (elérés:2022.10.30)

[9] Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Karának Kotlin alapú szoftverfejlesztés egyetemi tantárgya 2021 tavaszi félév – belső oktatási anyagok

[10] https://www.infoworld.com/article/3272244/what-is-the-jvm-introducing-the-java-virtual-machine.html (elérés:2022.10.30)

[11] Jim Wilson Creating Dinamic UIs with Android Fragments Second Edition (2016) – ISBN 978-1-78588-959-2

[12] J.F.DiMarzio; Beginning Android Programming with Android Studio (2017); ISBN 978-1-118-70559-9

[13] https://www.tutorialspoint.com/android/android_Fragments.htm (elérés:2022.10.30)

[14] https://developer.android.com/guide/navigation/navigation-principles (elérés:2022.10.30)

[15] https://developer.android.com/topic/libraries/architecture/livedata (elérés:2022.10.30)

[16] https://developer.android.com/topic/libraries/architecture/ViewModel (elérés:2022.10.30)

[17] Bosch – BME680: https://www.bosch-sensortec.com/products/environmentalsensors/gas-sensors/bme680/ (elérés:2022.10.30)

[18] https://www.espressif.com/en/products/socs/esp8266 (elérés:2022.10.30)

[19] https://www.microchip.com/en-us/product/MCP73843(elérés:2022.10.30)

[20] https://www.torex-europe.com/products/voltage-regulators/high-speed/xc6222/ (elérés:2022.10.30)

[21] https://www.silabs.com/interface/usb-bridges/usbxpress/device.cp2102n-gqfn24?tab=specs (elérés:2022.10.30)

[22] https://www.bosch-sensortec.com/software-tools/software/bsec/ (elérés:2022.10.30)

[23] https://mqtt.org (elérés:2022.10.30)

[24] https://www.djangoproject.com/ (elérés:2022.10.30)

[25] https://www.json.org/json-hu.html (elérés:2022.10.30)

[26] https://square.github.io/retrofit/: https://github.com/square/retrofit (elérés:2022.10.30)

[27] https://diplomaterv.vik.bme.hu/hu/Theses/BME-ETT-SmartLab-Applikacio-Fejlesztese (elérés:2022.10.30)

[28] https://github.com/wuseal/JsonToKotlinClass (elérés:2022.10.30)

[29] Philipp Jahoda; https://github.com/PhilJay/MPAndroidChart (elérés:2021.05.13)

[30] https://www.youtube.com/watch?v=N7J27pBTtuI (elérés:2022.10.30)

[31] https://www.youtube.com/watch?v=H_ItzJp5yVE (elérés:2022.10.30)

[32] https://developer.android.com/topic/libraries/architecture/ViewModel (elérés:2022.10.30)

[33] A. Géczy, L. Kuglics, I. Megyeri, R. Gelbmann and G. Harsányi, "Sensor-based IoT monitoring of Electronics Manufacturing in University Lab Environment," 2021 IEEE 27th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, 2021, pp. 230-233, doi: 10.1109/SIITME53254.2021.9663576.*

Sensor-based IoT monitoring of Electronics Manufacturing in University Lab Environment

Attila Géczy,
Department of Electronics Technology
Budapest University of Technology and Economics, Faculty of Electrical Engineering and Informatics)
Budapest, Hungary
gattila@ett.bme.hu

Lajos Kuglics
Department of Electronics Technology
Budapest University of Technology and Economics, Faculty of Electrical Engineering and Informatics)
Budapest, Hungary

István Megyeri,
Department of Electronics Technology
Budapest University of Technology and Economics, Faculty of Electrical Engineering and Informatics)
Budapest, Hungary

Róbert Gelbmann
Department of Electronics Technology
Budapest University of Technology and Economics, Faculty of Electrical Engineering and Informatics)
Budapest, Hungary

Gábor Harsányi
Department of Electronics Technology
Budapest University of Technology and Economics, Faculty of Electrical Engineering and Informatics)
Budapest, Hungary

***Abstract*—We present a sensor-based IoT monitoring of an electronics manufacturing laboratory in a university environment. The project focuses on the possibilities of retrofitting the current electronics manufacturing equipment (e.g., surface mount technology apparatus) with sensors to gather extensive in-situ information about the critical parameters of the machines and their ambient surroundings. The project is based on two different sensor nodes functioning as fixed and mobile measurement devices around the given points of the laboratory located at our university. The demonstration presents a cost-effective solution of retrofitting machines with IoT-based sensor nodes, enabling new data acquisition functionalities. Also, the demonstration shows an example of a system from where the students can learn about IoT and Industry 4.0 requirements in their educational environment and from remote access. The project also highlights the importance of operational safety and health (OSH) in such workplaces. The paper presents the block diagrams and the functionalities of the sensor nodes. Also, it presents the frontend and backend setup. The user interface layers are also presented, one, which is available through web access, and the other is a proprietary app for clear visualization of the recorded data. The two approaches share the same SQL database, which is stored on the server.**

Keywords— IoT, Industry 4.0, electronics manufacturing, SMT, surface mounting, Raspberry Pi, applied sensors

I. INTRODUCTION (*HEADING 1*)

Our department was a pioneer in establishing a modern electronics manufacturing laboratory for educational purposes. This was also a motivating factor in presenting a virtualized version of our laboratory for e-learning [1] back in 2000. The laboratory was moved to a new location recently; now, we prepared an IoT-based smart-device extension for further expanding the education possibilities of our lab, not only for students but also for industrial partners too, to present the local and remote monitoring possibilities of an IoT based system.

The concept was presented in the literature at other universities or teaching institutes as well. In the following part we selected some of the most recent results in the given field, to show the relevancy of our work.

Khriji et al [2] showed a solution for wireless sensor network (WSN) for a smart-lab purpose. The IoT devices are integrated to a centralized system, with a proprietary application programming interface (API), to present an easy to use method to collect and represent the sensor data. The test device results are stored in MySQL database, and the visualization is presented with JSON technology. Pachecho et al. [3] presented the easy use of Arduino, in order to perform adoption of IoT technologies for easy remote learning. The cheap solution is relevant in many economic sectors, such as transportation, energy, agriculture, home automation and the electronics industry as well. Maiti et al. [4] showed that an IoT network can be used for a complex university course, where sensors and lab equipment monitoring is used for smart learning in agricultural application. Lin presented an experimental classroom with intelligent laboratory management system based on the IoT technology. The work presented the sensor and hardware basis, also the application which is running on mobile platform. Lin showed that such methods can improve the teaching and reduce the burden of management and logistics of a laboratory class [5]. Tokarz et al [6] also showed an IoT network infrastructure for remote learning (or in other words, distant laboratory classes), where the physical hardware and student interaction is enhanced with IoT approach. The method is also connected to the recent SARS-COV-2 outbreak, which needed novel solutions for distant learning. In a most recent paper, Samonte et al. [7] presented a mobile application based IoT smart laboratory, where the environment variables and laboratory settings are used to analyze data or generate visual reports. The biosafety, the air quality, temperature, humidity and power outages are monitored in medical or experimental facilities. The study shows the application of microcontrollers, sensors, and related breakout boards. Safety is also a concern in the paper of Kanál and Kovácsházy [8], where BLE (Bluetooth Low Energy) sensor boards, a BLE-WiFi gateway (formed by a single board card computer) are utilized to investigate parameters for students overall well-being at universities and dormitories.

The CO_2 level, temperature, humidity, noise, and illumination are highlighted in the given work.

Further examples can also be found in the literature, as the topic is getting more attention, due to the remote learning aspects.

We focused on the idea to present an universal node, which can be used at the laboratory ambiance, and also around given apparatus. Next to the fixed node, we present a mobile node, which can be worn by the user to investigate personal ambiance in the working area, to emphasize the importance of operational safety and health (OSH) too. The nodes are connected to a central card computer (Raspberry Pi), which works as a server in the network. The data can be visualized by a proprietary Android app and from web browser as well.

The concept is focused to be applied in an electronics manufacturing environment at university level. Our work can also help to promote remote education in the IoT mindset, as it was discussed above.

II. Experimental

A. Concept of the smart laboratory

The laboratory consists of several rooms as Figure 1 presents. The concept is to install sensor nodes to given locations and to given workers. When the nodes are installed, the map of the laboratory is accessible from online (web or native app), and a fixed, installed tablet (or an arbitrary mobile device) gives local, immediate access to all information, when entering the laboratory. Figure 1 shows our map with some selected positions of the fixed nodes, and the mobile device at the main entrance.

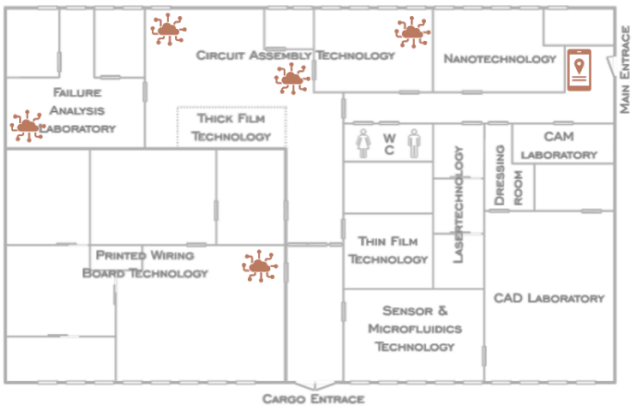


Fig. 1. Topology of our laboratory highlighting the positioned sensor nodes and the user interface on a mobile device.

B. Nodes used for data acquisition

The data is gathered by a Raspberry Pi 4, which is located in the center of the laboratory, and which is working as a server and a data logging system.

For the fixed node we use a proprietary device based on a Fishino Uno with ESP2866 Wifi, which is in connection with the central server (Raspberry Pi 4). The microcontroller board is in connection with a Sensirion SDP810 pressure sensor (interchangeable) to monitor pressure relations in reflow ovens, particularly vapour phase soldering, which is in focus for characterization studies. A MAX31855 K-type thermocouple amplifier breakout board is also added with an BME 680 sensor cluster with breakout board, to monitor temperature, humidity and ambient pressure relations. Furthermore a micro SD/SD card slot, for local logging, a 3,7 V Li-Po 650 mAh battery with proper DC/DC regulation (5V and 3,3V) for working during short outages, and a 128*64 px OLED screen is added for local information available to the operator. Figure 2 shows the first prototype of the fixed node.

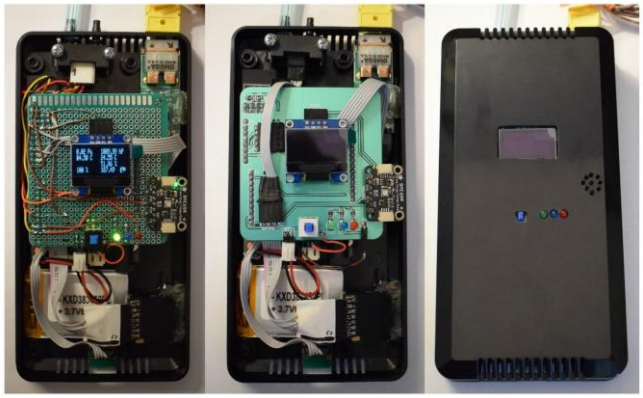


Fig. 2. Fixed node during development stage

Figure 3 shows an extended block diagram of the fixed node.

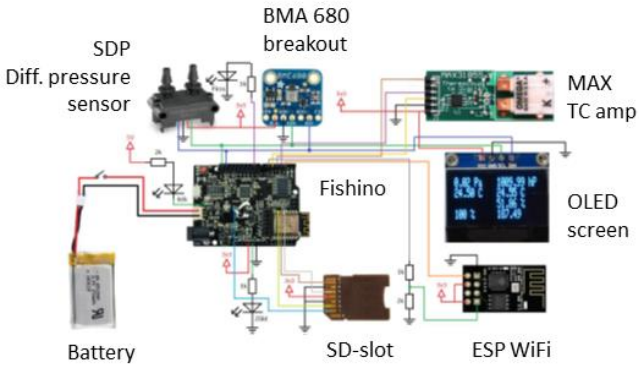


Fig. 3. Fixed node block diagram

For the wearable node, an Arduino Nano is used as the CPU. We used Arduino IDE 1.8.10 for software development. The sensors are the following. A CCS811 (I2C) module was applied, which is used for VOC and eCO2 measurement. Winsen ZPH01 particulate matter sensor was applied with PWM/UART. The Bluetooth module is HC-05. The power is supplied by a 2200 mAh power bank with 5V/1A output. The Bluetooth data is received by the central node (Raspberry Pi). The wearable node is presented in Figure 4.

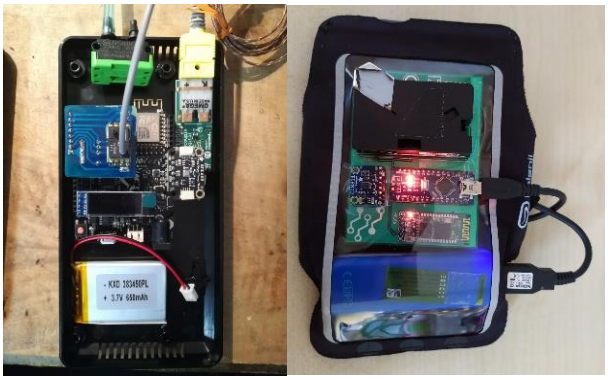


Fig. 4. Nodes in their more advanced prototype boxed-packaged form. Left: fixed node; Right: wearable node with arm fixture.

The wearable node is described in deeper details in [9]. The data is gathered by a Raspberry Pi 4, which is located in the center of the laboratory, and which is working as a server and a data logging system.

C. Backend and frontend

The backend and the frontend is based on the following setup. There is a Django (Python-based) shell to promote fast visualization. In order to improve visualization, an easy integration from Chart.JS to Django is performed. The responsive design is based on a HTML (CSS+JavaScript) system.

Summing up, the data flow is based on an SQL server, which is in connection with Django, then the flow points to Chart.JS+HTML base. Figure 5 presents the setup visually.

Fig. 5. Database to visualisation

It must be noted that the visualization is continuously improving from the aspect of design, the given results show a snapshot of the work during development.

D. Backend and frontend

The mobile app was developed in Android Studio (Kotlin), where standard libraries were used for http, graphical and other functions. A naviagtion bar is used to list the selectable options:

- laboratory map
- standard XY plot visualization
- progress bar/gauge bar view on the sensors of the nodes
- information about the sensors for educational purposes
- credits of the application.

The total system is represented in the block diagram of Figure 6.

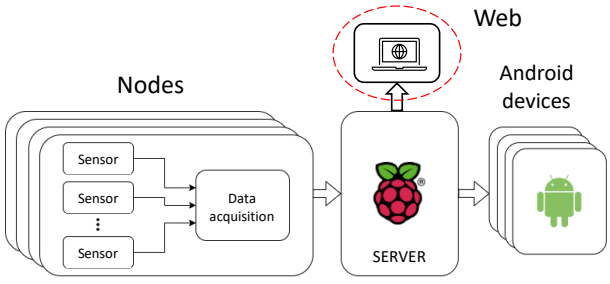


Fig. 6. Block diagram of the system .

III. RESULTS

A. Data Aquisition

Data acquisition was stable and firm with one fixed node and one wearable node. It must be noted that the size of the database is increasing with every days of worktime. With 1 s data acquisition time step, approximately 50 days of data resulted in 4.3 million SQL lines. The data can be handled by the computer, but it is necessary to backup and restart the database from time to time, in order to avoid congestion.

B. Mobile App

Figure 7 represents the mobile application view. In this view the laboratory map is the starting point, from where the given node can be selected. Then the sensor data can be selected from the data acquisition and the resulting database.

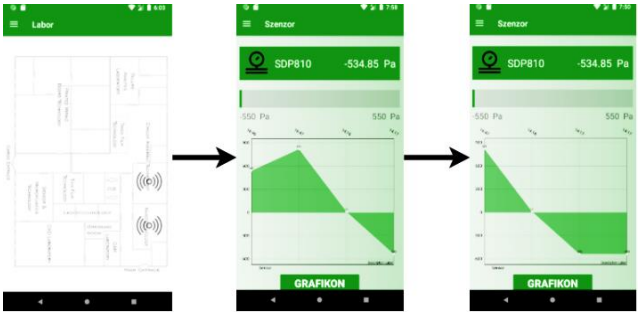


Fig. 7. Mobile app view (left: laboratory map, center and right: exemplary sensor data).

The next figure (Fig. 8) represents the bar chart (progress bar) and throttle/gauge chart visualization of the data.

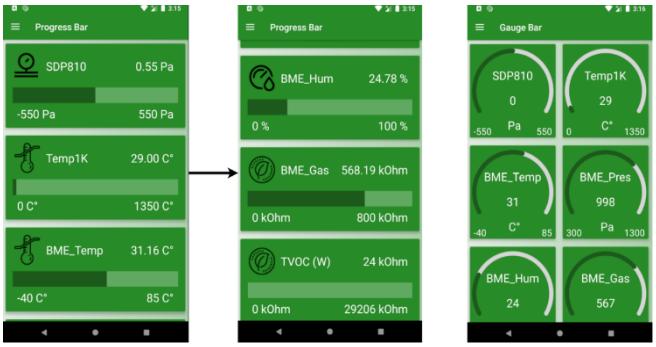


Fig. 8. Mobile app view with progress bar view (left, center) and gauge bar view (right)

The mobile app is currently working both from Android phones and from emulators on the PC.

C. Website view

Figure 9 presents a snapshot view on the web visualization. The figure presents timely snapshots over two nodes, where the different sensor values are presented in a time scale.

Node A represents a wearable measurement node in the lab, with two highlighted sensor results: CCS811 TVOC (Total volatile organic compounds) in (ppb) concentration value and ZHP01 particle count on the left. Node B on the right presents the monitoring of a reflow soldering oven with SDP810 differential pressure sensor and the ambiance around the oven with a BOSCH BME680 cluster, where the temperature is measured next to the oven for OSH purposes.

The next Figure 10 presents another overview on the sensors, which are described with pictures and a short note on their working principles. It is important for the students and any visitors to see, what is the role of each node, and what principles are the applied sensors working with.

Fig. 9. Recorded data visualization of two nodes (Node A – on the left, with TVOC measurement and particle count in the ambiance; Node B – on the right with differential pressure measured inside a reflow oven and a temperature increase in the ambiance of the oven.)

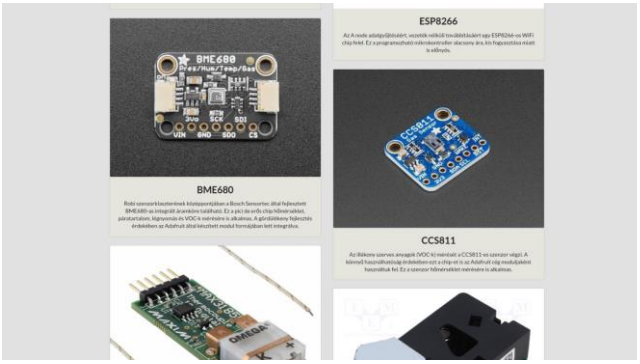


Fig. 10. Sensor description view on the applied types, breakout boards, parameters, working principles, etc.

IV. Conclusions

Now we can conclude the work with the following statements.

- Communication between the sensors were established within the nodes.
- The nodes are communicating with the central unit.
- The data is saved into a database, which is accessible with both a browser view and the proprietary app.
- Both the app and the web-based viewer is up and running, however the design must be improved to adhere to more modern and professional applications.

For the future, the following tasks are planned. A new modular cluster must be designed for a more efficient wearable device, which can also be optimized for various wearable approaches. Also, localization with Bluetooth technology could be investigated for the wearable devices. The website can be improved for more information and exemplary data recorded previously. Also, the node count should be improved so that all workplaces can be monitored around the lab. The demonstration is planned to be presented in the laboratory during the next semester to the students, where ~300 students are involved in different lab-classes in a revolving stage manner. The sensors may enable better and continuous characterization of our running processes, which data is planned to be utilized in future researches – e.g. the novel characterization of vapour phase soldering with pressure sensors.

Acknowledgment *(Heading 5)*

The help of Bence Szabó and László Gál (UniPCB) is appreciated in the laboratory.

The research reported in this paper and carried out at the BME has been supported by the NRDI Fund based on the charter of bolster issued by the NRDI Office under the auspices of the Ministry for Innovation and Technology.

The vapour phase soldering related research was supported by the National Research, Development and Innovation Office—NKFIH, FK 132186.

References

[1] Gordon Peter; Bojta Peter; Hertel Lorant; Kallai Istvan; Lepsenyi Imre; Varnai Laszlo; Illyefalvi-Vitez Zsolt;Progress in electronics packaging virtual laboratory development, Electronic Components and Technology Conference: Proceedings, Piscataway, USA, IEEE (2000) 1 756 pp. 1293-1299.

[2] S. Khriji, D. El Houssaini, R. Barioul, T. Rehman and O. Kanoun, "Smart-Lab: Design and Implementation of an IoT-based Laboratory Platform," 2020 IEEE 6th World Forum on Internet of Things (WF-IoT), 2020, pp. 1-5, doi: 10.1109/WF-IoT48130.2020.9221143.

[3] A. Fernández-Pacheco, S. Martin and M. Castro, "Implementation of an Arduino Remote Laboratory with Raspberry Pi," 2019 IEEE Global Engineering Education Conference (EDUCON), 2019, pp. 1415-1418, doi: 10.1109/EDUCON.2019.8725030.

[4] A. Maiti, T. Byrne and A. A. Kist, "Teaching Internet of Things in a Collaborative Laboratory Environment," 2019 5th Experiment International Conference (exp.at'19), 2019, pp. 193-198, doi: 10.1109/EXPAT.2019.8876480.

[5] R. Lin, "IoT Experimental Classroom Project under the Mode of Industry-University-Research Collaboration," 2020 5th International Conference on Smart Grid and Electrical Automation (ICSGEA), 2020, pp. 433-436, doi: 10.1109/ICSGEA51094.2020.00099.

[6] Krzysztof Tokarz; Piotr Czekalski; Gabriel Drabik;Jarosław Paduch; Salvatore Distefano; Riccardo Di Pietro; Giovanni Merlino; Carlo Scaffidi; Raivo Sell; Godlove Suila Kuaban, Internet of Things Network Infrastructure for The Educational Purpose, 2020 IEEE Frontiers in Education Conference (FIE)

[7] M. J. C. Samonte, F. A. G. Mendoza, R. Pablo and S. M. P. Villa, "Internet-of-Things Based Smart Laboratory Environment Monitoring System," 2021 IEEE 8th International Conference on Industrial Engineering and Applications (ICIEA), 2021, pp. 497-502, doi: 10.1109/ICIEA52957.2021.9436758.

[8] A. K. Kanál and K. Tamás, "Assessment of Indoor Air Quality of Educational Facilities using an IoT Solution for a Healthy Learning Environment," 2020 IEEE International Instrumentation and Measurement Technology Conference (I2MTC), 2020, pp. 1-6, doi: 10.1109/I2MTC43012.2020.9129231.

[9] A. Géczy, L. Kuglics, L. Jakab and G. Harsányi, "Wearable Smart Prototype for Personal Air Quality Monitoring," *2020 IEEE 26th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, 2020, pp. 84-88, doi: 10.1109/SIITME50350.2020.9292309.