MŰEGYETEM 1782

SZAKDOLGOZAT FELADAT

Gyulai Péter
Villamosmérnök hallgató részére

IoT mérőrendszer fejlesztése adatfolyam alapokon

Az *Internet of Things (IoT)*, azaz magyar fordításban elterjedten használt *Dolgok Internete* olyan hálózatba kapcsolt eszközök összességét jelenti, amelyek egy adott feladat megoldására integrálják a különböző hardver alapú és szoftver alapú technológiákat, szoros kapcsolatban vannak egymással és környezetükkel, miközben intenzív adatcserét folytatnak az internet szolgáltatásait kihasználva.

Egy IoT rendszer alkalmazhatósága rendkívül sokrétű, jelen Szakdolgozatban egy időjárási adatokat gyűjtő, feldolgozó és megjelenítő rendszer létrehozása a feladat, figyelembe véve az adatfolyam alapokon történő megvalósítást és az IoT rendszer specialitásaiból fakadó kihívásokat mind a hardver, mind a szoftver tekintetében.

A hallgató feladatának a következőkre kell kiterjednie:

- Ismertesse az IoT rendszereket és azok alkalmazhatóságát
- Válasszon ki egy megfelelő mikrokontrollert és hozzá szenzorokat az adatgyűjtő rendszer számára és indokolja választását
- Tervezze meg az adatgyűjtő egység tápellátását
- Tervezze meg az adattárolást és megjelenítést megvalósító egységeket
- Tervezze meg a nyomtatott áramkört
- Ismertesse az alkalmazott programozási nyelv és platform kiválasztási szempontjait
- Mutassa be az adatfolyam architektúrát
- Ismertesse a szoftverfejlesztést
- Mutassa be az alkalmazott jelfeldolgozást az időjárási adatok kiértékelésére és megjelenítésére

Tanszéki konzulens: Krébesz Tamás István, tanársegéd
Külső konzulens: -

Budapest, 2020.10.10.

……..
Dr. Dabóczi Tamás
tanszékvezető
egyetemi tanár, DSc

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

IoT mérőrendszer fejlesztése adatfolyam alapokon

Készítette
Gyulai Péter (D475X8)

Konzulens
Krébesz Tamás István

2020

TARTALOMJEGYZÉK

HALLGATÓI NYILATKOZAT

Alulírott Gyulai Péter, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2020. 12. 07.

.....Gyulai Péter.....
Gyulai Péter

Összefoglaló

Az Internet of Things (IoT) egy sokak számára nehezen megfogható technológiai fogalom, amely manapság egyre nagyobb szerepet kap a mindennapi életben. A hálózatra kapcsolt eszközök száma évről évre exponenciálisan növekszik, és már régen átlépte az emberi populáció lélekszámát. Egyes előrejelzések szerint az ilyen eszközök száma 2030-ra elérheti az 50 milliárdot is.

Az ilyen eszközök tervezése egy komplex és összetett folyamat, amelyhez rendkívül széleskörű mérnöki tudásra és tapasztalatra van szükség. Komplexitása ellenére gyakorlatilag bármilyen képzettségű mérnök vagy hobbista számára lehetséges egy IoT rendszer elkészítése. A szakdolgozatom célja, hogy útmutatóként szolgáljon az érdeklődők számára egy ilyen eszköz megtervezésében, hardver és szoftver tekintetében egyaránt. Ebben a dokumentumban először áttekintem az IoT irodalmi hátterét, majd a hardvertervezés legfontosabb tudnivalóit, végül pedig az eszközön futó szoftver készítésének alapjait egy viszonylag ritkán használt, de annál sokoldalúbb technológia, az adatfolyam architektúra alkalmazásával. A dolgozat során minden esetben igyekszem bemutatni és összehasonlítani az elképzelhető megoldásokat, egy internethez csatlakoztatott, időjárási paramétereket mérő és megjelenítő eszköz példájával illusztrálva a különböző témaköröket. A dolgozat végére az Olvasó átfogó ismereteket szerezhet az érintett témakörökről és tapasztaltabban kezdheti meg saját IoT eszközeinek fejlesztését.

A bemutatott ismeretek minden mérnöki beállítottságú Olvasó gyakorlati hasznára válhatnak, és kiindulási alapot jelentenek a további, mélyebb tapasztalatszerzés számára. Az ismertetett megoldások mindegyike számos módon tovább fejleszthető, ezzel is inspirálva a mérnöki tudományok további előrehaladását.

Abstract

The Internet of Things (IoT) is a hard to grasp technological term for a lot of people, which nowadays gains an ever increasing significance in our everyday lives. The number of network connected devices raises every year in an exponential manner and has long surpassed the size of the human population. According to some forecasts, the number of such devices can reach 50 billion by the year 2030.

The design of these devices is a complex and compound challenge, which requires an extremely broad spectrum of engineering knowledge and experience. Despite of it's complexity, it is possible for engineers and hobbyists of practically any level, to create IoT systems. My thesis work aims to serve as a guide for people being interested in the design of such devices, from hardware to software engineering alike. In this document, I will first briefly describe the literature of IoT, the must-known aspects of hardware design and finally, the details of the software which will run on the final product, using a not so widely adapted but much more powerful technology, the dataflow architecture. During this work, I try to describe and compare the available solutions in detail, illustrating the covered topics with the example of a network-connected device, monitoring and displaying weather parameters. By the end of this thesis work, the reader can gain comprehensive knowledge about the topics I am covering and can start creating his/her own IoT projekt with a little more experience.

The knowledge gained can serve great for every engineering-minded reader and can provide a solid starting ground for further and deeper study. All of the technologies mentioned in this document can be improved in several ways, inspiring the continuing advancement of engineering and science in general.

1. Bevezetés

Az Internet of Things (IoT), vagy a „Dolgok Internete” egy egyre nagyobb népszerűséget nyerő fogalom, amely hamarosan meghatározhatja a mindennapjainkat. Rendkívül széles felhasználási területtel rendelkezik, mint például az okos otthonok, városok, járművek vagy a különféle iparágak. A szakdolgozatom célja, hogy bemutassa az IoT alkalmazási területeit, valamint útmutatást adjon egy ilyen eszköz hardverének és szoftverének megtervezéséhez. A továbbiakban először röviden ismertetem az IoT alapjait, megszületésének körülményeit és felhasználási területeit. A fejezet végén bemutatom a szakdolgozatom során elkészített IoT eszköz működését, majd a következő fejezetekben a hardver- és szoftvertervezés legfontosabb tudnivalóit.

1.1. Internet of Things

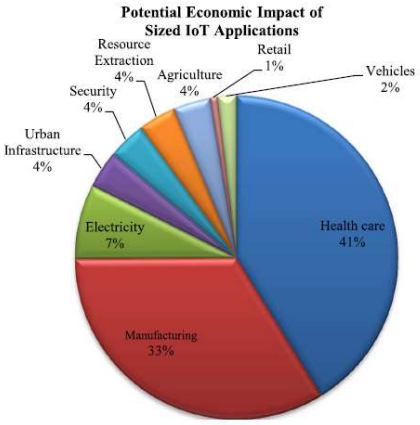
Az Internet of Things kifejezés alatt azon fizikai eszközök hálózatra kapcsolt összességét értjük, amelyek a bennük található szenzorok, beavatkozó egységes és a rajtuk futó szoftver segítségével képesek adatokat gyűjteni a környezetükből, abba beavatkozni, illetve egymás között a gyűjtött adatokat az Internet segítségével megosztani. A tématerület szoros kapcsolatban áll számos más technológiával is, mint például a Big Data, gépi tanulás vagy az automatizáció. A rendszer működéséhez elengedhetetlen a különböző megoldások integrációja, amely lehetővé teszi, hogy a sokféle eszköz képes legyen hatékonyan kommunikálni egymással.

Az első ilyen internetre kapcsolt eszköz 1982-ben készült [1] a Carnegie Mellon Egyetemen, az egyetem egyik hallgatója, David Nichols jóvoltából. Az egyetemen található italautomata túlságosan messze volt, így sokszor előfordult, hogy mire odaért, már elfogyott az ital az automatából, vagy pedig nem volt elég hideg. Nichols két hallgatótársa, valamint az egyetem egyik mérnökének segítségét kérte. A megoldást az automata internethez történő csatlakoztatása jelentette, így távolról is lekérdezhető volt annak tartalma és az italok hőmérséklete. Maga az Internet of Things kifejezést először Kevin Ashton használta [1] 1999-ben, amikor be kívánta vezetni az RFID[1] technológiát a Procter & Gable cégnél, az árukészlet mozgásának nyomon követésére. Az ötlet számos támogatóra talált, a többek között Ashton által is alapított Auto-ID Center már száznál több szponzort tudhatott magáénak [1] a 2000-es évek elejére. Azóta a hálózatba kapcsolt eszközök gondolata széles körben elterjedt, többek között az Intel és a Google is elkötelezte magát a technológia iránt.

Az Internet of Things felhasználási területe rendkívül széles skálát ölel fel, melyet összefoglalóan az 1.1.1. Ábra mutatja. Az egyik legegyszerűbb és kézzelfoghatóbb példája az okos otthonok. Az internet segítségével képesek lehetünk otthonaink automatizálására, többek között okos fényforrások, hűtő-fűtő berendezések, vezérelhető nyílászárók és biztonsági rendszerek révén. Folyamatos és naprakész információkat kaphatunk a háztartási

[1] RFID: Radio Frequency Identification, magyarul: rádiófrekvenciás azonosítás

eszközeink állapotáról, működéséről és fogyasztásáról, ezzel elősegítve az energiahatékonyságot is. Az interneten keresztül akár távolról is vezérelhetjük az otthoni berendezéseket, például bekapcsolhatjuk a fűtést a telefonunk segítségével, mielőtt hazaindulunk.



1.1.1 Ábra – IoT eszközök iparágak szerinti megoszlása 2025-re [2]

Ha éppen nem tartózkodunk otthon, akkor is segítségünkre lehetnek az okos eszközeink, melyek egyre több járműbe is beépítésre kerülnek. A parkolóhelyekbe épített szenzorok segítségével pillanatok alatt megtudhatjuk, hol található a legközelebbi parkolóhely, vagy merre van a legrövidebb út, ezzel csökkentve az utazási időt és kímélve a környezetet. Városaink automatizálásával csökkenthetők a közlekedési dugók, minimalizálhatók a balesetek és felgyorsítható a szállítmányozás. Egészségünk megőrzésére többféle testen viselhető okos eszköz, például karóra áll rendelkezésünkre, melyek folyamatosan mérik a különféle egészségi paramétereinket és naprakész információkkal szolgálnak az állapotunkról. Vészhelyzet esetén képesek lehetnek akár beavatkozás nélkül segítséget hívni a bajbajutottakhoz. A mért adatainkat feldolgozva és tárolva hasznosak lehetnek a különféle betegségek előrejelzésében és az orvosok számára is fontos információval szolgálhatnak. A technológia nem csupán a hétköznapokban, de a munkában is áttöréseket hozhat. Számos iparág élvezheti az IoT előnyeit, amennyiben a vállalatok hajlandók megtenni az első lépéseket. A technológia ipari alkalmazásait Industrial Internet of Things (IIoT) néven emlegetik és szorosan kapcsolódik az ipari automatizációhoz. A mezőgazdaságban kiemelkedő jelentőséggel bírnak az időjárási körülmények, melyek széleskörű megfigyelése összekapcsolt szenzorhálózatokon keresztül lehetséges. A gyártástechnológiában is jelentős költségeket spórolhatunk meg, ha a gyártósoron lévő gépek ellenőrzése automatikusan elvégezhető. Az elkészült termékek útja nyomon követhető a gyártási helytől egészen a felhasználókig. Az otthonokban történő fogyasztás megfigyelésével a szolgáltatók képesek lehetnek a felhasználói igényekhez jobban alkalmazkodni, ezzel jobb minőségű ellátást biztosítani az emberek számára.

A technológia egyes aspektusainak szétválasztására egy réteges modellt vezethetünk be. A tématerület irodalmában számos modellt alkalmaznak, melyek hasonlóságot mutatnak az internet alapját képező TCP/IP modellel. Az újabb publikációkban egy ötrétegű modellt alkalmaznak leginkább [2], amely lehetőséget ad az IoT alkalmazások feladatainak megfelelő kategorizálására.

- **Objektum réteg:** Ebben a rétegben helyezkednek el a valós, fizikai eszközök, szenzorok és beavatkozók, melyek célja a környezetükből történő adatgyűjtés. A mért paraméterek rendkívül sokszínűek lehetnek, pl. hőmérséklet, nyomás, páratartalom, gyorsulás vagy egészségügyi mutatók. A rengeteg különböző hardveres platform viszont megnehezíti az egységes kommunikációt, ezért ezt a problémát a következő rétegben kell megoldani. A rétegek ismertetéséhez példaként egy gyártósori automatizálást hozok fel, ahol jelen rétegbe tartozhatnak a termékek pozícióját rögzítő szenzorok.
- **Objektum absztrakciós réteg:** Ennek a rétegnek a feladata, hogy a hardverek különbözőségéből adódó problémákat áthidalja, és az objektum rétegben keletkezett adatokat biztonságosan továbbítsa a felsőbb rétegek felé. Itt találhatóak a különféle hardverközelibb adatátviteli protokollok, mint például a WiFi vagy a Bluetooth. Az előző pontban említett gyártósor esetében ebben a rétegben valósulhat meg a szenzorok által mért adatok továbbítása a vállalati szerver felé.
- **Szolgáltatás menedzsment réteg:** Miután már rendelkezésre állnak az eszközeink által mért adatok, azokat rendezett formában továbbítani a felhasználók számára. Ebben a rétegben történik meg a különböző hardverek által gyűjtött adatok összepárosítása az azokat igénylő szolgáltatásokkal, egyedi eszközazonosítók segítségével. Ide tartozhat az említett példában az egyes szenzorok párosítása a gyártósor különböző szakaszaival.
- **Alkalmazás réteg:** Az alkalmazás réteg felelős a fogyasztók által kívánt szolgáltatások és azok megbízhatóságának biztosításáért. Ide tartoznak például az internetes alkalmazáshoz kapcsolódó programozói felületek és adatbázisok, melyek segítségével lekérdezhetők az egyes mérési adatok. Az alkalmazás réteg megtalálható gyakorlatilag az összes felhasználási területen. A gyártósorunk esetében ez a réteg valósítja meg a termékek pozíciójának lekérdezését internetes felületen keresztül.
- **Üzleti réteg:** Az üzleti réteg feladata a teljes IoT infrastruktúra összefogása egy közös elérési felületen. Itt történhet meg a mérési adatok analizálása, megjelenítése, illetve a döntéshozatali és döntéstámogatási rendszerek megvalósítása. A fenti példa esetében ez a réteg felelős a termékek pozíciójáról gyűjtött adatok kiértékeléséért, például azonosíthatja a gyártás legkritikusabb pontjait és statisztikákat készíthet a termelés menetéről.

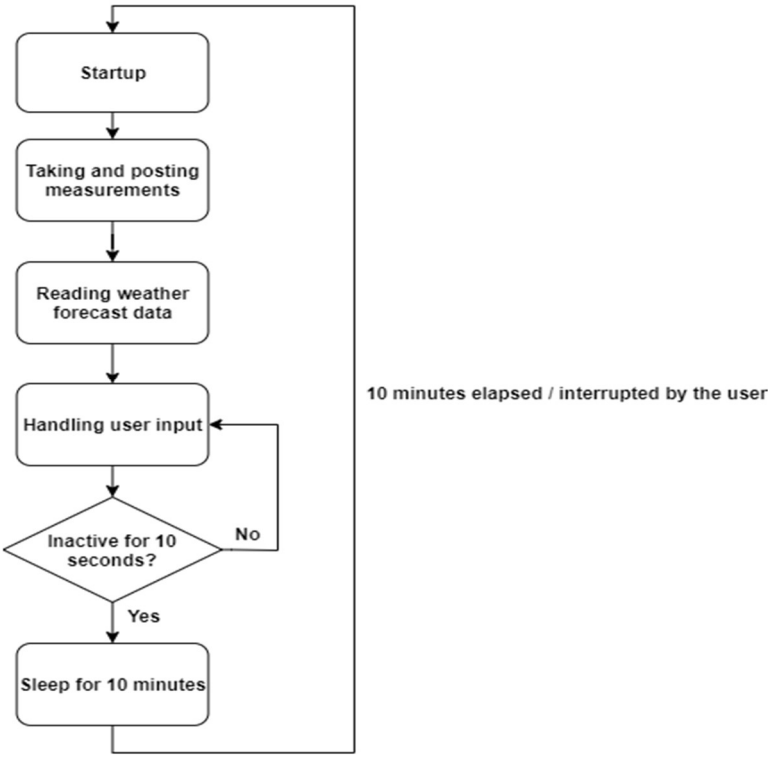
A szakdolgozatom során elkészített eszköz hardvere az objektum rétegben, míg szoftvere az objektum absztrakciós rétegekben helyezkedik el.

1.2. Megvalósított mérőrendszer

A szakdolgozatom során bemutatott tervezési módszerek és eljárások illusztrálására egy időjárási paramétereket mérő és kijelző rendszer tervezését mutatom be. A hardver mikrokontroller alapú, tervezésének lépéseit részletesen a második fejezetben mutatom be, a rajta futó szoftvert pedig a harmadik fejezetben. A megvalósított rendszer képes a hőmér-

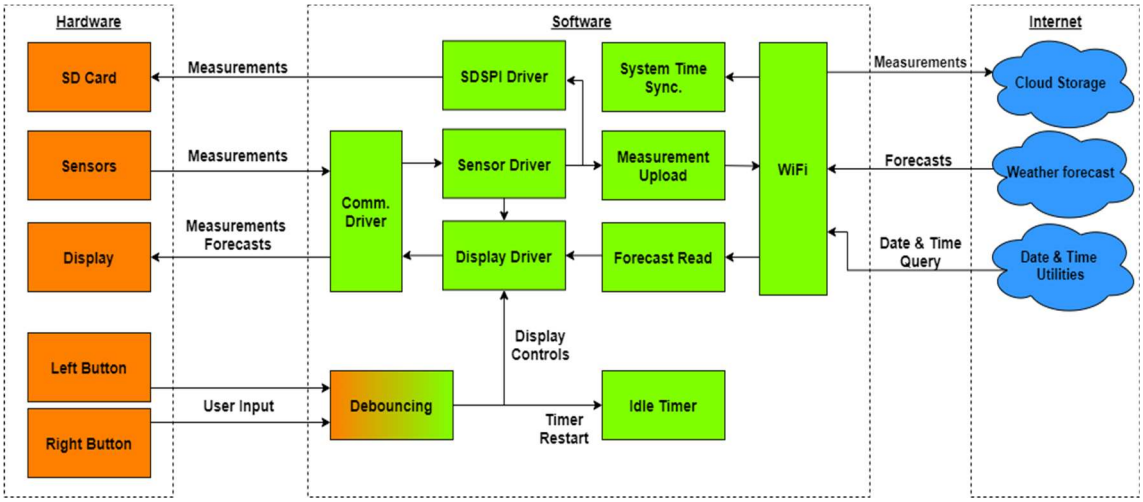
séklet, légnyomás és relatív páratartalom mérésére, a mért értékek grafikus kijelzőn történő megjelenítésére és internetkapcsolaton keresztül a felhőbe történő továbbításra. A mérési eredmények emellett opcionálisan SD-kártyára is mentésre kerülnek.

Az internetre történő csatlakozást vezetéknélküli kapcsolat formájában, WiFi protokoll segítségével végzi. A helyszínen mért paraméterek mellett az eszköz képes az internet segítségével időjárás előrejelzés adatokat is lekérdezni és azokat a kijelzőn megjeleníteni. Az eszköz tápellátása akkumulátoron keresztül történik, melynek töltése USB-csatlakozón keresztül biztosított, és az akkumulátor feszültsége a mikrokontroller által folyamatosan megfigyelésre kerül. A rendszer kezelése a nyomtatott áramköri panelen elhelyezett nyomógombok segítségével történik. A kijelzőn háromféle menüpont közül lehet választani. Az első képernyő a helyszínen mért adatokat, a második az internetes előrejelzés adatait, a harmadik pedig a státuszinformációkat (pl. akkumulátorfeszültség, utolsó frissítés időpontja, stb.) jeleníti meg. Az egyes képernyők között a nyomógombok segítségével lehet váltani mindkét irányban. Az akkumulátor élettartamának növelése céljából az eszköz az idő nagy részében energiatakarékos módban, a mikrokontroller legalacsonyabb fogyasztású alvási módját kihasználva működik. Ebben az állapotban az eszköz nem végez méréseket, a kijelzőn a legutoljára megjelenített képernyő látszik. Az alvó állapotból kétféleképpen ébredhet fel az eszköz: valamelyik nyomógomb megnyomására, vagy pedig automatikusan 10 percnyi inaktivitás után. Az ébredés után a rendszer elvégzi a szükséges méréseket, hálózati kommunikációt és figyeli a nyomógombok állapotát. Amennyiben a rendszer végzett a saját statikus feladataival és a felhasználó sem lép interakcióba a nyomógombokkal 10 másodpercig, újrakezdődik az alvó állapot. Az előzetesen megkívánt működési szekvenciát az 1.2.1. ábra szemlélteti.



1.2.1. Ábra – Az eszköz működési szekvenciája

A rendszer megvalósítása egyaránt igényel hardverben és szoftverben elvégzendő feladatokat, illetve a két réteg közötti kapcsolat megfelelő kezelését is. A hardver feladata a szenzorok, a kijelző, az SD kártya és a nyomógombok biztosítása és ezek megfelelő fizikai illesztése. Az átmeneti réteg biztosítja, hogy a magasszintű szoftveres logika képes legyen kommunikálni a hardverrel, ide tartoznak a különféle kommunikációs illesztőprogramok és a nyomógombok pergésmentesítése is. Utóbbi hardverben és szoftverben egyaránt megoldható, amennyiben szükséges a hardveres megoldás szoftverből kiegészíthető. Az inaktivitási időtartam mérése szintén szoftverben történhet, az időzítőt a nyomógombok megnyomása újraindítja. A szoftver feladata a mérési adatok megfelelő kezelése, azok továbbítása WiFi-n keresztül az internet felé, tárolása az SD kártyán és megjelenítése a kijelzőn. Opcionálisan ide tartozhat a rendszeridő folyamatos frissítése az interneten keresztül, amennyiben a hardveres időzítés pontatlansága ezt szükségessé teszi. A feladat megoldásához szükséges egyes internetes szolgáltatások használata, ilyenek a mérési adatok online tárolása, az időjárás előrejelzés lekérdezése, illetve opcionálisan a dátum és idő lekérdezése. A feladatok hardver-szoftver felosztása és az eszköz funkcionális rendszerterve az 1.2.2. ábrán látható.



1.2.2. Ábra – Az eszköz funkcionális rendszerterve

2. Hardver

Amikor egy IoT eszközre gondolunk, legyen az akár egy okos otthonokba szánt automatizálási egység, viselhető készülék (pl. okos-óra) vagy egy időjárás-állomás, elsőként a konkrét hardveres megvalósítás jut eszünkbe. Ez összhangban áll az IoT alapgondolatával, tehát az internetre kapcsolt valós fizikai eszközök hálózaton keresztül történő együttműködésével. Ezek az alkalmazások sajátos igényeket támasztanak a megtervezendő hardverrel kapcsolatban, ebben a fejezetben az ezzel kapcsolatos jellemző kihívásokkal foglalkozom. Áttekintésre kerülnek többek között az eszköz magját alkotó mikrokontroller kiválasztásával kapcsolatos kérdések, a felhasználni kívánt szenzorok kiválasztásának szempontjai, az eszköz tápellátásának biztosításának különböző módjai és az adattárolás, helyi adatmegjelenítés lehetséges megoldásai is. Végül az elkészült kapcsolási rajz alapján történő nyomtatott áramkörtervezés alapjairól teszek említést. A fejlesztés során a nyílt-forrású tervezés alapelveit követtem, ennek megfelelően a kapcsolási rajz és egyéb dokumentációs anyagok angol nyelven készültek, ezeket változtatás nélkül közlöm.

A szakdolgozatom hardvertervezési fázisa során a KiCad ingyenes, nyílt-forráskódú áramkörtervező szoftverét használtam, a bemutatott kapcsolási rajzok és nyomtatott áramköri panelek ábráit, 3D grafikákat is ebben készítettem. A szoftver működésének, használatának pontos részleteivel a dolgozat keretében nem foglalkozom, amennyiben az olvasó érdeklődik a téma iránt, [3] segítheti az elindulásban:

2.1. Mikrokontroller kiválasztása

Egy IoT eszköz hardverének fejlesztése során az egyik legfontosabb lépés a rendszer magját alkotó mikrokontroller kiválasztása. Manapság számos gyártó sokféle különböző eszközeiből válogathatunk, ezek mindegyikének megvannak a maguk előnyei, illetve hátrányai. A választásunkat több tényező is befolyásolhatja, ezek közül kiemelkedő jelentőséggel bírnak [4] az alábbiak:

- **Teljesítmény:** Egyszerűbb alkalmazásoknál sokszor elegendő alacsony számítási kapacitással rendelkező, olcsóbb mikrokontrollerek használata. Ezek általában kis lábszámú, kisfogyasztású és alacsony órajelfrekvenciájú integrált áramkörök. Amennyiben a termékünk nem igényel bonyolult vagy nagy mennyiségű számításokat, megfontolandó lehetőséget jelenthet egy egyszerűbb mikrokontroller. A piacon is elérhető népszerű választás az Atmel (később Microchip Technology) ATtiny mikrokontrollercsaládja. Magasabb számítási igények esetén ezek az eszközök többnyire már nem elégségesek és szükséges a középkategóriás választék áttekintése. Ebben a kategóriában népszerűek a STMicroelectronics (pl. STM32), Silicon Labs (pl. EFM32) és az Atmel (pl. ATMega328) erősebb eszközei. Vezetéknélküli alkalmazásokban megfontolandók az Espressif mikrokontrollerei (pl. ESP32), vagy akár Raspberry Pi mikroszámítógépek.

- **Fogyasztás:** Sokszor szükségünk van olyan eszközök fejlesztésére, melyek nem rendelkeznek folyamatos hálózati tápellátással. A legtöbb hasonló alkalmazás valamilyen nem-tölthető (elemes) vagy tölthető (akkumulátoros) tápforrással oldja meg ezt a problémát. A kényelmes felhasználhatóság szempontjából nem mellékes, hogy az adott eszközt mennyi időnként szükséges feltölteni, vagy benne elemet cserélni, tehát ezek várható gyakoriságát minimalizálni kell a fejlesztés során. Ennek legegyszerűbb módja egy kisfogyasztású mikrokontroller használata, illetve a választott eszköz alvó-üzemmódjainak alapos tanulmányozása.
- **Költségek:** Az anyagi szempontok természetesen megjelennek a fejlesztés korai szakaszában is. Nem csupán a fizikai eszköz árát soroljuk ide, a fejlesztés költségeit is figyelembe kell vennünk. Ide tartozhatnak például a programozáshoz szükséges hardver és szoftverkomponensek, licenszek és mérnöki munkaórák költségei is. Minden apró pluszköltség befolyásolja a végtermék árát, ennek hatásai tömeggyártás esetén jól megmutatkoznak a nagy gyártási volumen miatt. A költségek becslésének témakörével részletesen [5] foglalkozik.
- **Szoftveres komponensek:** A felhasználni kívánt mikrokontroller kiválasztásakor fontos szempont lehet a rendelkezésre álló szoftverkönyvtárak elérhetősége is. A fejlesztés során sok időt és költséget takaríthatunk meg azzal, ha nem szükséges minden szoftverkomponenst magunknak elkészítenünk, hanem felhasználhatunk már létező forrásokat is. Ide értendők többek között a beágyazott operációs rendszerek, függvénykönyvtárak, eszközillesztők és példakódok is. Fontos megemlíteni továbbá az adott mikrokontrollerrel kapcsolatos korábbi fejlesztési tapasztalatot is.

A szakdolgozatom során, a feladat megoldásához felhasznált mikrokontroller kiválasztásakor a következő szempontokat és megállapításokat vettem figyelembe. A pillanatnyi időjárási körülmények megfigyelése nem igényel jelentősebb számítási kapacitást, azonban a vezetéknélküli internetkapcsolat fenntartása és a hálózati kommunikáció megköveteli a legalább közepes teljesítményű mikrokontroller használatát. Az eszköz akkumulátoros tápellátása miatt fontos szempont, hogy az áramfelvételt inaktív állapotban jelentősen csökkenteni lehessen a különböző alvó-üzemmódok felhasználásával. Az alkalmazás egyszerűsége miatt az olcsóbb mikrokontrollerek alkalmazása kedvezően befolyásolja az ár-érték arányt. A feladat egyéb aspektusai (pl. hardvertervezés) és a szűkös időkorlát miatt jelentős előnyt jelent a kész szoftverkönyvtárak könnyű elérhetősége. A megvalósítás komplexitásának felesleges növelését jelentené, ha a választott mikrokontroller nem támogatná beépített módon a vezetéknélküli kommunikációt, ezért kizárólag olyan eszközök jöhetnek szóba, amelyek integráltan tartalmazzák ezt a lehetőséget. A tervezési fázis során négy mikrokontrollert hasonlítottam össze az alkalmazás igényei alapján, majd az értékeléseket a 2.1.1. Táblázatban foglaltam össze. A továbbiakban néhány mondatban bemutatom a mérlegelt lehetőségeket:

- **Arduino Uno:** Hobbisták és amatőr fejlesztők által kedvelt egyszerű mikrokontrolleres fejlesztői kártya, széleskörű szoftveres támogatással, oktatóanyagokkal és

saját fejlesztői környezettel. Az internetes alkalmazásokat egyszerűen illeszthető bővítőkártyán keresztül támogatja.

- **Raspberry Pi:** Eredetileg oktatási célokra kifejlesztett, nagyobb teljesítményű fejlesztői kártya, bonyolultabb otthon-automatizálási [6] alkalmazásokban előszeretettel használják. Beépített módon támogatja az internetes alkalmazásokat Ethernet vagy WiFi kapcsolaton keresztül.
- **ESP32:** Kifejezetten vezetéknélküli IoT alkalmazásokra szánt, alacsony költségű mikrokontroller. Egyre nagyobb népszerűségnek [7] örvend az utóbbi időben, aktív gyártói támogatással és folyamatos fejlesztésekkel büszkélkedhet. Széleskörű szoftveres támogatással rendelkezik, Arduino könyvtárakat is használhatunk, valamint saját IoT keretrendszerrel is rendelkezik.
- **ESP8266:** Az ESP32 elődje, szintén IoT alkalmazásokra kifejlesztett mikrokontroller. A későbbi változathoz képest kisebb teljesítményű, kevesebb lábbal és beépített perifériával rendelkezik. A meglévő perifériák lábkiosztása rögzített, az ESP32-vel ellentétben nem konfigurálható.

	Arduino Uno	Raspberry Pi	ESP32	ESP8266
Teljesítmény	alacsony	magas	közepes	közepes
Órajel frekvencia	16 MHz	1.4 GHz	160 MHz	80 MHz
Költség	közepes	magas	alacsony	alacsony
Fogyasztás	közepes	magas	közepes	közepes
Szoftver	kiváló	közepes	kiváló	közepes
Tapasztalat	közepes	kevés	magas	kevés
Internetkapcsolat	bővítőkártya	beépített	beépített	beépített

2.1.1. Táblázat – Mikrokontrollerek összehasonlítása

A különböző szempontok és lehetőségek mérlegelése után a választásom az Espressif Systems ESP32 mikrokontrollerére esett. Ennek teljesítménye az időjárási paraméterek megfigyeléséhez tökéletesen elegendő, ára alacsony, fogyasztása nem kiemelkedő és az alvó-üzemmódok figyelembevételével akár alacsony is lehet. Ezek mellett az elérhető modulok beépített módon támogatják a WiFi és Bluetooth kapcsolatok kezelését, valamint szoftveres támogatásuk is kiváló, ami jelentősen egyszerűsíti a fejlesztést. Az ESP32 a piacon olcsón és egyszerűen beszerezhető, modulok, fejlesztőkártyák széles választéka áll rendelkezésre. Annak érdekében, hogy a mikrokontroller tényleges alkalmazáshoz történő illesztése minél könnyebb legyen egy olyan modult választottam, amely a mikrokontroller működéséhez szükséges minimális külső áramköröket már tartalmazza. A tervezés pillanatában két hasonló modul volt egyszerűen beszerezhető, a 2.1.1. ábrán látható ESP32-WROOM-32D és az ESP32-WROOM-32U. A kettő közötti

különbséget egyedül a vezetéknélküli kommunikációhoz használt antenna jelenti, előbbi integrált PCB[1] antennát tartalmaz, utóbbihoz pedig szabványos csatlakozón keresztül saját antenna illeszthető.

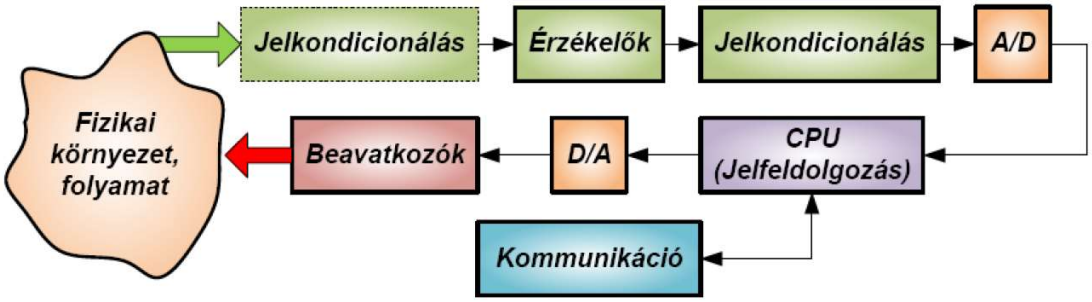
2.1.1. Ábra – ESP32-WROOM-32D

2.2. Szenzorok kiválasztása

Egy IoT eszköz fejlesztése során általában az elsődleges célunk a minket körülvevő világról történő információgyűjtés és a megszerzett információk hálózaton keresztül történő megosztása, felhasználása. A környezetünkből villamos mennyiségek formájában szenzorok segítségével gyűjthetünk adatokat, ezért kiemelt fontosságú, hogy megfelelően tájékozottak legyünk a szenzorok kiválasztásának szempontjaival. A gyakorlatban is használt eszközök általánosságban két fő kategóriába sorolhatók, léteznek analóg és digitális [8] eszközök. A továbbiakban röviden bemutatom a két csoport közötti lényeges eltéréseket, azok előnyeit és hátrányait, majd a konkrét feladathoz kapcsolódó szenzorok példáján keresztül illusztrálom a mérlegelés folyamatát.

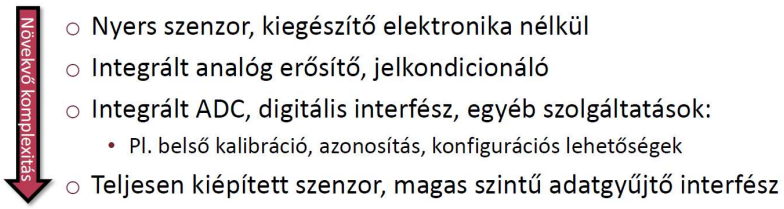
Az analóg szenzorok közös tulajdonsága, hogy bármilyen környezeti paraméter mérését végzik, az információt időben és jelszintben folytonos mennyiség formájában teszik elérhetővé az őket tartalmazó rendszer számára. Ez az információ folyamatosan, bármely időpillanatban tetszőleges pontossággal elérhető. A különböző szenzorok különféle működési elveken alapulnak, ennek megfelelően az információt hordozó analóg mennyiség is sokféle formát ölthet. Kizárólag az elektromos tulajdonságok közül megemlíthetjük az ellenállás, feszültség, áramerősség, frekvencia, illetve kapacitás- vagy töltés kimenettel rendelkező szenzorokat. Analóg kimenettel rendelkező szenzorok esetében általános előnyként említhetjük meg azok egyszerű felépítését és az alacsony költségeket. Hátrányt jelent viszont, hogy az analóg kimenetet magunknak kell visszamérnünk, a mért jelet megfelelően kondicionálnunk, majd digitalizálnunk és végül feldolgoznunk. Az analóg jelfeldolgozás folyamatát a 2.2.1. Ábra mutatja. Ezek mellett általában közvetlen kalibrációs lehetőségünk sincsen, miután megvalósítottuk a szenzort, annak különböző paramétereit nehezen lehet változtatni.

[1] PCB: Printed Circuit Board, magyarul: nyomtatott áramköri panel



2.2.1. Ábra – Analóg jelfeldolgozás beágyazott rendszerekben [9]

A digitális szenzorok az fentiekkel ellentétben feldolgozásra kész formában állítják elő az alkalmazás számára szükséges környezeti információkat. Belső működésüket tekintve többnyire ezek is analóg jelekkel dolgoznak, azonban a külvilág számára az analóg információt digitalizált formában bocsátják rendelkezésre. Többségük valamilyen szabványos kommunikációs protokollal (pl. SPI, I2C, UART) rendelkezik, léteznek soros illetve párhuzamos adatkapcsolati elven működő eszközök, ezek közül a soros kommunikációt alkalmazó szenzorok élveznek szinte kizárólagos népszerűséget. Előnyeik közé sorolható, hogy a szenzorból kinyert környezeti paraméterek többnyire közvetlenül felhasználhatók és a legtöbb esetben több beállítást is dinamikusan elvégezhetünk a beültetés után is. A kényelmesebb felhasználásnak azonban ára van, a digitális szenzorok általában bonyolult felépítésűek, belső működésükbe nem látunk bele, illetve többnyire drágábbak is. A gyakorlatban különböző bonyolultságú szenzorok állnak rendelkezésre, ezek általános felosztását a 2.2.2. Ábra mutatja.

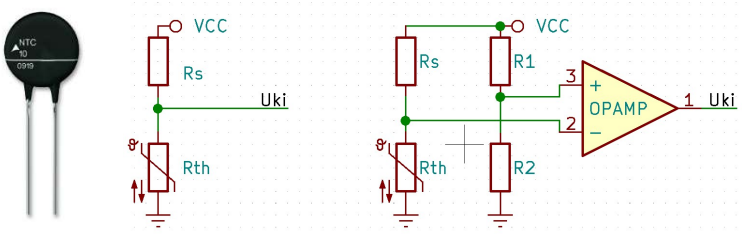


2.2.2. Ábra – Különböző komplexitású szenzortípusok [9]

A szakdolgozatom során megvalósított időjárás-megfigyelő eszköz számára szükséges mérendő környezeti paraméterek a hőmérséklet, légnyomás és relatív páratartalom. A továbbiakban bemutatok néhány eszközt, amelyek képesen ezek mérésére. A különböző időjárási mennyiségek esetében külön-külön értékeltem a lehetséges megoldásokat, néhány szenzor azonban többször is megjelenik, mivel több paraméter mérésére is képesek. A mérlegelés során analóg és digitális szenzorokat hasonlítottam össze, majd az értékeléseket táblázatban összesítettem.

2.2.1. Hőmérséklet

A pillanatnyi hőmérséklet mérésére három megoldást mérlegeltem. Ezek közül az első egy hagyományos analóg típusú, NTC[1] termisztor. Általában kisebb pontossági követelmények esetén használják, a mérés lényege, hogy ezek az eszközök a hőmérséklet változásával arányosan változtatják az elektromos ellenállásukat, szobahőmérsékleten pedig névleges ellenállással rendelkeznek. Az ellenállás megváltozását valamilyen módon mérve következtethetünk a hőmérsékletre. Kevésbé igényes megvalósítás lehet, ha egy egyszerű ellenállás-osztó egyik tagjaként alkalmazzuk, nagyobb pontosságú mérésekhez viszont célszerű mérőhidat építeni. Az említett kapcsolásokat a 2.2.3. Ábra mutatja. Fontos tudnivaló, hogy a megoldás hátránya, hogy az ellenállás és a hőmérséklet kapcsolata nem minden esetben szigorúan lineáris, a vizsgált tartományban szükséges lehet lineáris közelítés alkalmazása. A termisztoros hőmérsékletmérés pontos menetéről [10] [11] szolgálhatnak további információval.



2.2.3. Ábra – Termisztor, ellenállás-osztó (balra) és mérőhíd (jobbra) konfiguráció

A másik lehetőség valamilyen digitális eszköz alkalmazása volt, mint amilyen a 2.2.4. ábrán is látható. Két ilyen eszközt hasonlítottam össze a velük kapcsolatos korábbi tapasztalataim alapján. Mindkét eszköz képes a hőmérséklet mellett egyéb paraméterek mérésére is, ez előnyt jelent az analóg megoldással szemben. Digitális szenzorok használata esetén minimális komplexitású külső áramkörökre van csak szükség, ezért ezek használata lényegesen egyszerűbb. Mindkét szenzor esetében a mérések gyárilag kalibrált formában állnak rendelkezésre kiolvasás után.

2.2.4. Ábra – DHT22 hőmérséklet és páratartalom szenzor

Az eredményeket a 2.2.1. Táblázatban összesítettem.

Szenzor	NTC termisztor	DHT22	BME280
Típus	analóg	digitális	digitális

[1] NTC: Negative Thermal Coefficient, magyarul: negatív hőmérsékleti együttható

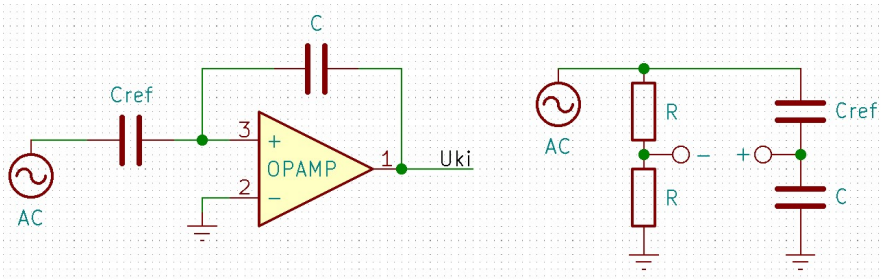
Pontosság	közepes	magas	magas
Konfiguráció	nincs	nincs	lehetséges
Költségek	alacsony	közepes	közepes
Egyéb mérések	nincs	páratartalom	páratartalom, nyomás

2.2.1. Táblázat – Hőmérséklet szenzorok összehasonlítása

2.2.2. Páratartalom

A relatív páratartalom mérésére is léteznek analóg, illetve digitális megoldások. Mindkét típus esetében többnyire gyárilag kalibrált mérésekről beszélhetünk. Az értékelés során – a hőmérséklet méréshez hasonlóan – három eszközt hasonlítottam össze, ezek közül egy eszköz analóg működésű, a másik kettő pedig a korábban említett két digitális szenzor.

A 2.2.6. ábrán látható analóg szenzor kapacitás kimenettel rendelkezik, melynek mérése további áramköri összeállítást igényel. A kapacitás változásának mérésére is többféle kapcsolást használnak a gyakorlatban. A legegyszerűbb mérési összeállítás valamilyen AC[1]-hídkapcsolást alkalmaz. Kapacitás kimenetű szenzorokkal történő méréskor megemlítendő az időmérésre visszavezetett kapcsolások használata, melynek során a változó kapacitást valamilyen oszcillátorkonfiguráció segítségével időmérésre vezetik vissza. A 2.2.5. Ábra két egyszerű kapacitásmérésre használható kapcsolást [11] mutat be, az időmérésre visszavezetett mérésről pedig [12] tartalmaz további információt.



2.2.5. Ábra – Kapacitásmérés erősítővel (balra) és AC-híddal (jobbra)

Az alkalmazás egyszerűsége miatt az összehasonlítás során a digitális szenzorokat preferáltam, mivel a kapacitás mérésének pontos megvalósítása bonyolult mérési összeállítást igényelt volna. Az eredményeket a 2.2.2. Táblázat foglalja össze.

Szenzor	HS1101LF	DHT22	BME280
Típus	analóg	digitális	digitális
Kimenet	kapacitás	digitális	digitális
Pontosság	közepes	közepes	közepes

[1] AC: Alternating Current, magyarul: váltakozó áram

Konfiguráció	nincs	nincs	lehetséges
Költségek	közepes	közepes	közepes
Egyéb mérések	nincs	hőmérséklet	hőmérséklet, nyomás

2.2.2. Táblázat – Páratartalom szenzorok összehasonlítása

2.2.6. Ábra - HS1101LF analóg páratartalom szenzor

2.2.3. Légnyomás

A korábbiakhoz hasonlóan a légnyomás mérésére is összehasonlítottam különböző analóg és digitális megoldásokat, melyek közül kettő a 2.2.7. ábrán látható. Az analóg megoldások általában kapacitív vagy piezorezisztív elven működnek, melyek lényege, hogy a légnyomás változása deformálja a mérési felület alakját, ennek eredményeként pedig megváltozik annak ellenállása, vagy az összeállítás elektromos kapacitása. Az eredményeket a 2.2.3. Táblázatban foglaltam össze.

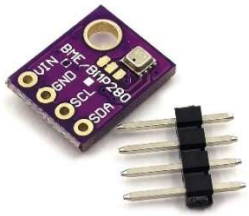
Szenzor	SPD030G	MPL3115A2	BME280
Típus	analóg	digitális	digitális
Kimenet	feszültség	digitális	digitális
Pontosság	magas	magas	közepes
Konfiguráció	nincs	lehetséges	lehetséges
Költségek	magas	közepes	közepes
Egyéb mérések	nincs	magasság	hőmérséklet, páratartalom

2.2.3. Táblázat – Légnyomás szenzorok összehasonlítása

2.2.7. Ábra – SPD030G (balra) és MPL3115A2 (jobbra) nyomásszenzorok

2.2.4. Összefoglalás

Az eddigiek során vizsgált megoldások közül a választásom végül a Bosch BME280-as szenzormoduljára esett, amely a 2.2.8. ábrán látható. Előnyei közé sorolható a digitális kommunikációs felület, a viszonylag alacsony ára és a széleskörű konfigurációs lehetőségek beültetés után is. További előnyként említendő, hogy képes mindhárom az alkalmazásban szükséges környezeti paraméter mérésére. Az eszköz egyetlen jelentős hátránya, hogy önmagában kézi beültetésre kevésbé alkalmas, ezért a tényleges áramkör megépítése során egy tüskesoron keresztül csatlakoztatható, előre elkészített modult alkalmaztam.



2.2.8. Ábra – BME280 meteorológiai szenzormodul

A kiválasztott modul 3.3-5V tápfeszültséggel is képes üzemelni a beépített feszültségszabályzó segítségével. Kommunikációs felületként I2C buszt használ, és támogatja a High Speed üzemmódot, akár 3.2 Mbit/s adatátviteli sebességig. A mérhető hőmérséklettartomány -40 és 80 Celsius fok között található, a légnyomás mérésére pedig 300 és 1100 hPa között képes, ezzel bőven lefedve a kívánt mérési tartományt. Normális működési körülmények között a hőmérsékletmérés pontossága 0.5 Celsius fok, míg a légnyomásra vonatkozó tűrése 1 hPa. A relatív páratartalom mérések pontossága a teljes működési tartományon 3%. Mivel a feladat nem kíván meg rendkívüli pontosságot, ezért ezek a tűréshatárok az alkalmazás szempontjából elfogadhatók. A szenzor áramfelvétele inaktív állapotban mindösszesen 0.1 µA, ezáltal megfelelően alacsony fogyasztással rendelkezik akkumulátoros üzemhez is. Másodpercenként történő folyamatos mérés esetén a fogyasztás 3.6 µA-re növekszik, azonban a felhasználás módjából adódóan a gyakorlatban ennek csak a töredékével kell majd számolni.

2.3. Tápellátás megtervezése

Bármilyen elektronikus eszközről is van szó, a megfelelő tápellátás biztosítása kulcsfontosságú kérdés a tervezési folyamat során. Nincs ez máshogy az IoT eszközök esetében sem, azonban ezek a termékek további kihívásokat támasztanak a fejlesztőkkel szemben. Ugyan nem feltétlenül elvárás velük szemben, hogy folyamatos hálózati tápellátás nélkül is képesek legyenek huzamosabb ideig üzemelni, sokszor mégis törekedni érdemes erre. A hordozható, folyamatos áramellátással nem rendelkező helyekre telepített eszközök esetében a megoldást a helyszínen elérhető megújuló energiaforrások (pl. napelem) kihasználása, illetve az akkumulátoros vagy elemes táplálás jelentheti. A felhasználó szempontjából azonban az eszközök feltöltése, elemeinek cseréje kényelmetlenséget jelent vagy akár nem is megoldható (pl. távoli, nehezen megközelíthető helyszíneken), ezért

fokozottan ügyelni kell az ilyen esetek gyakoriságának csökkentésére. A továbbiakban az elektronikus eszközök megfelelő tápellátásának biztosításainak módjait mutatom be, kezdve a különböző tápforrás csatlakoztatási lehetőségek rövid bemutatásával, részletezem az akkumulátoros táplálás fontosabb tudnivalóit, valamint említést teszek a feszültségszabályozás lehetséges megoldásairól is. Végül röviden bemutatok egy egyszerű megoldást az akkumulátorról működő eszközök tápfeszültségének megfigyelésére. A továbbiakban bemutatott kapcsolási rajz részletek a szakdolgozatom során megvalósított eszköz terveiből származnak, a megvalósított eszköz is ezeket a kapcsolásokat tartalmazza.

2.3.1. Csatlakozók, ESD védelem

Akár folyamatos hálózati tápellátással rendelkező, akár akkumulátoros eszközt tervezünk, valamilyen módon csatlakoztatnunk kell az eszközt az elektromos hálózathoz. Magasabb feszültségről üzemelő eszköz esetén elképzelhető közvetlen csatlakozás a hálózathoz, ennek megvalósítása azonban meghaladja jelen szakdolgozat kereteit. A legtöbb esetben eszközeinket valamilyen DC tápforráson keresztül tápláljuk, illetve töltjük, mely bizonyos körülmények között lehet dedikált tápegység, fali AC/DC adapter, vagy USB-port. A felhasználástól függően többféle szabványos csatlakozó áll rendelkezésünkre, ezek közül a legelterjedtebbek a 2.3.1. ábrán látható Barrel-jack és USB-csatlakozók.

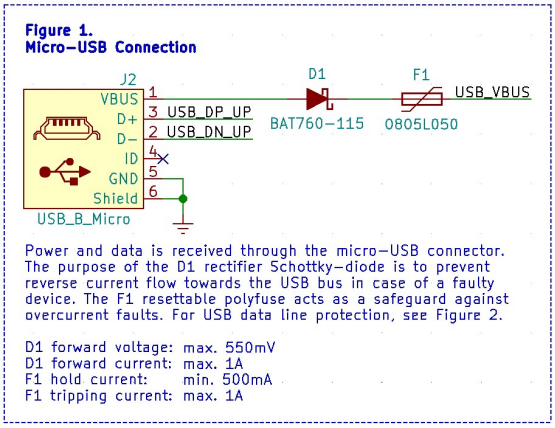
2.3.1. Ábra – Barrel-jack (balra) és Mini-USB (jobbra) csatlakozók

Bármilyen csatlakozót is választunk, néhány dologra feltétlenül szükséges odafigyelnünk az áramkörünk tervezésekor. A legtöbb esetben ezek a csatlakozók jelentik az eszköz egyetlen közvetlen fizikai érintkezési felületét a külvilággal. Az ilyen felületek különösen veszélyeztethetik az eszközünk működését, mivel az emberi testtel történő érintkezés során előforduló ESD[1]-események rövid, de nagy energiájú elektromos kisüléseket eredményezhetnek, és ideiglenes rendkívül magas feszültségek (akár több mint 20 kV) is kialakulhatnak. Veszélyt jelenthet továbbá a felhasználói hibából eredő nem megfelelő használat is, például a fordított polaritású fali adapter használata is. Ezek ellen feltétlenül meg kell védenünk az eszközeinket.

Ahogy a 2.3.2. ábrán is látható, egy micro-USB csatlakozót választottam tápellátás (és kommunikációs) kapcsolódási pontnak. Annak érdekében, hogy belső hiba esetén ne indulhasson el fordított irányú áram az USB-port felé, annak meghibásodását kockázhatva, egy alacsony nyitófeszültségű Schottky-diódát terveztem az áramútba. Amennyiben az eszközben valahol rövidzár keletkezik, veszélyesen megnövekedhet az áramfelvétel. Ennek megakadályozására egy áramkorlátozó Polyfuse-t kötöttem sorosan az áram útjába, melynek garantált kioldási árama 1A, azonban a tervezett maximális áramfelvételig

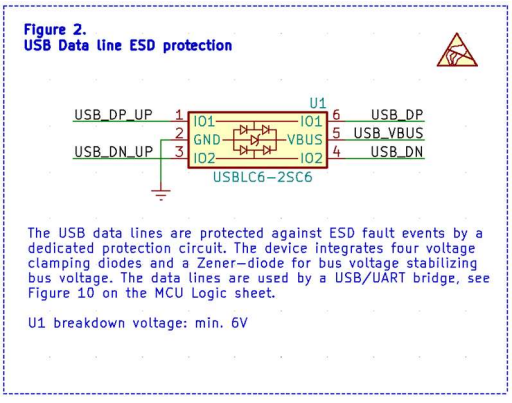
[1] ESD: Electrostatic Discharge, magyarul: elektrosztatikus kisülés

(0.5A) biztosan működőképes. Ezek a biztosítékként szolgáló áramköri elemek túlzott áram hatására megváltoztatják belső szerkezetüket, ellenállásuk drámainak megnövekszik, gyakorlatilag megszakítják az áramkört. A túláram okának megszűnése után szerkezetük rövid idő után magától helyreáll, így nem igényelnek külső beavatkozást.



2.3.2. Ábra – USB csatlakozó és áramvédelmek

Az ESD-események okozta meghibásodások ellen általában különféle ún. TVS[1]-diódákat alkalmaznak. Ezek a diódák rövid ideig nagyon magas áramokat is képesek vezetni és megakadályozzák a feszültség elfogadhatatlanul magas megemelkedését, azonban tartós hibaállapot esetén nem képesek megfelelő védelmet nyújtani. A tartós feszültségemelkedés megakadályozására általában párhuzamos Zener-diódát szokás alkalmazni. Ezek általános jellemzője, hogy nem engedik a rajtuk eső feszültséget az ún. letörési feszültségük fölé emelkedni. A szakdolgozatomban megvalósított eszköz egy kifejezetten USB-csatlakozókhoz szánt ESD-védelmi áramkört használ, mely integráltan tartalmazza a TVS- és Zener-diódákat is, valamint egyaránt véd pozitív és negatív kisülések ellen. A szakdolgozatom hardveréhez egy USB 2.0 buszhoz dedikált, 6V-os letörési feszültségű ESD védelmet használtam, melynek kapcsolását a 2.3.3. Ábra mutatja.



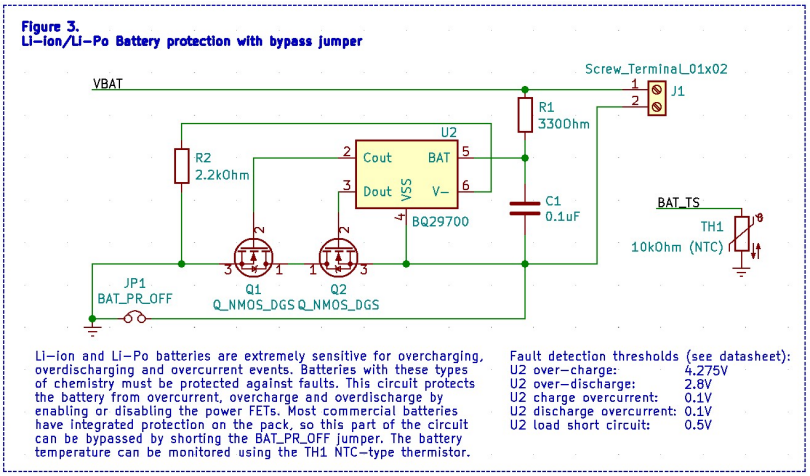
2.3.3. Ábra – ESD-védelem USB-csatlakozóhoz

[1] TVS: Transient Voltage-Supressor, magyarul: tranziens feszültség-elnyomás

2.3.2. Akkumulátoros üzem

Akkumulátoros táplálás esetén sok különféle belső kémiával rendelkező típus közül választhatunk, a különféle kémiával rendelkező eszközök összehasonlítása meghaladja a szakdolgozatom kereteit, ezért a továbbiakban csak a lítium alapú technológiával foglalkozom. Ezek közül talán a leggyakrabban használt variánsok a Li-ion és Li-Po (polimer) nevet viselik. Ezek az akkumulátortípusok magas energiasűrűséggel rendelkeznek, ennek köszönhetően kis méretben is elérhetőek a szükséges kapacitású darabok. Legnagyobb előnyük egyben az egyik legnagyobb hátrányuk is, mivel a magas energiasűrűségük miatt rendkívül érzékenyek. Megfelelő védelem nélkül könnyen tűz- és robbanásveszélyesek is lehetnek, ezért különös figyelmet kell fordítani rájuk.

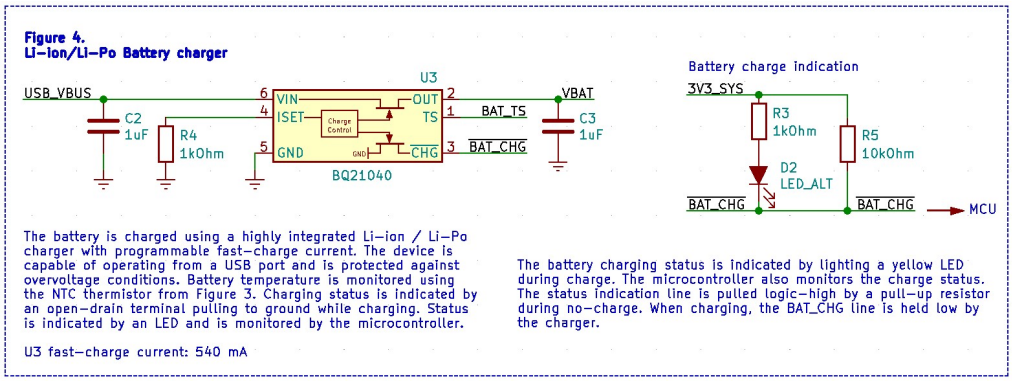
A gyakorlatban különböző kapacitású akkumulátorok kaphatók. Egy akkumulátor névleges kapacitását általában Ah-ban (amper-óra) vagy mAh-ban (milliamper-óra) adják meg, ennek jelentése, hogy mennyi ideig képesek adott áramerősséget szolgáltatni, mielőtt teljesen lemerülnek. Például egy 1200mAh kapacitású akkumulátort 1.2 A-es terheléssel névlegesen 1 óráig lehet üzemeltetni. A valóságban többnyire ez nem megvalósítható, a hőmérséklettől és egyéb körülményektől függően a ténylegesen kivehető kapacitás kevesebb a névlegesnél. A mérnöki gyakorlatban egy akkumulátor kapacitását általában C-vel szokás jelölni, és a megengedhető legnagyobb töltőáram és terhelés nagyságát is C-ben adják meg. Ökölszabályként elmondható, hogy egy Li-ion vagy Li-Po akkumulátort nem javasolt [13] tartósan 1C-nél nagyobb terheléssel vagy töltőárammal üzemeltetni. A szakdolgozatomban megvalósított eszköz is rendelkezik beépített akkumulátor-védelemmel, melyhez direkt erre a célra tervezett integrált áramkörök kaphatók. Mivel a legtöbb akkumulátor rendelkezik beépített védelemmel, az áramkörnek ez a része egy jumper[1] segítségével kikerülhető. A külső akkumulátor védelmére szolgáló kapcsolást a 2.3.4. Ábra szemlélteti.



2.3.4. Ábra – Akkumulátor-védelem

[1] jumper: az áram útját választható módon záró vagy megszakító áramköri elem

A korábban említett tulajdonságaik miatt a lítium alapú akkumulátorok töltése is különös körültekintést igényel. Erre a célra léteznek dedikált integrált áramkörök, amelyek biztonságos módon képesek elvégezni a töltési folyamatot. A legtöbb hasonló eszközön konfigurálható az alkalmazott töltőáram nagysága, valamint egyéb védelmekkel is el vannak látva. Az alábbi ábrán az általam felhasznált kapcsolás látható, a töltőáram az USB-szabványnak is közel megfelelő maximálisan 500mA, az akkumulátor melegedésére a 2.3.4. ábrán látható NTC termisztor figyelmeztet. A töltés állapotáról egy LED tájékoztatja a felhasználót, valamint a mikrokontroller felé is található elvezetés. A legtöbb dedikált töltőáramkör esetében ún. open-drain kimenetű lábakon kapunk tájékoztatást a töltés állapotáról, melynek lényege, hogy inaktív állapotban a kimenet nagyimpedanciás állapotban van, az áramkör megszakított, ilyenkor a feszültséget egy felhúzóellenállással magasban tartjuk. A töltés alatt az eszköz zárja az áramkört és közvetlenül alacsonyba húzza a kimeneti feszültséget. A töltést és annak visszajelzését a 2.3.5. Ábra illusztrálja.



2.3.5. Ábra – Akkumulátortöltés és töltés-visszajelzés

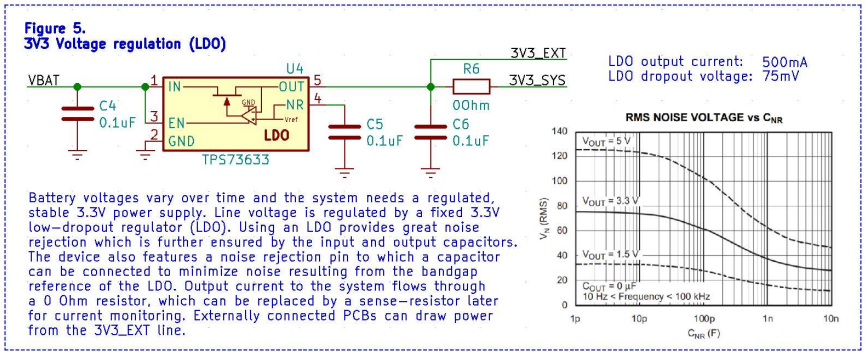
2.3.3. Feszültségszabályozás

Akár közvetlen táplálású, akár akkumulátoros tápellátásról beszélünk, mindkét esetben szükség van a tápfeszültség megfelelő stabilizálására a megkívánt feszültségszinten. A fent bemutatott akkumulátoros esetben a rendszer tápfeszültsége stabilizáció előtt megegyezik az akkumulátor feszültségével, amely azonban folyamatosan változik. A feszültség szabályozására általánosságban két lehetőségünk [4] van, kapcsolóüzemű konvertert vagy lineáris szabályzót használhatunk. Szükség esetén a két megoldást kaszkádba kapcsolva egyesíthetjük azok előnyeit is, ilyenkor tipikusan a kapcsolóüzemű átalakító kimenetét kötjük a lineáris szabályzóra.

Kapcsolóüzemű átalakítók: Ezeknek a stabilizátoroknak közös jellemzője, hogy az áramút folyamatos, nagysebességű kapcsolgatásával állítják elő a kívánt feszültséget. Legnagyobb előnyük a magas hatásfok és hogy szinte tetszőleges feszültség előállítható velük, akár alacsonyabb, akár magasabb, mint a bemeneti feszültségük. Legfőbb hátrányuk, hogy általában zajos kimeneti feszültséget állítanak elő, mely főként a kapcsolóüzemű működésből adódik, továbbá sokszor igényelnek külső, ritkábban használt áramköri elemeket, például induktivitást.

Lineáris szabályzók: Ezek a stabilizátorok egy nagyteljesítményű FET[1] segítségével szabályozzák a kimenetükön található feszültséget. A tranzisztor vezérlésére használt feszültséggel változtatják annak áteresztőképességét, a szabályozást a kimenetről visszacsatolt erősítő végzi. Legnagyobb előnyük a kapcsolási zajtól mentes, tiszta kimenet, azonban csak a bemenetnél alacsonyabb feszültségre tudnak szabályozni és folyamatosan kénytelenek elfogyasztani a többletenergiát, ezért könnyebben melegszenek. A lineáris szabályzóknak szükségük van rá, hogy a bemeneti feszültségük valamivel magasabb legyen, mint az elvárt kimeneti feszültségük. Ha elvárás, hogy a bemeneti feszültség minél inkább megközelíthesse a kimeneti feszültséget, érdemes LDO[2]-t használni.

A szakdolgozatom során az alkalmazás egyszerűsítése és a zajmentes tápfeszültség érdekében LDO használata mellett döntöttem, amely a 2.3.6. ábrán látható. A folyamatos fogyasztásból adódó hátrányokat az áramfelvétel minimalizálásával és megfelelő hűtéssel kompenzáltam. Az áramfelvétel későbbi esetleges mérését elősegítendő az áram útjába terveztem egy 0 Ohm értékű ellenállást, mely később feszültségmérés céljából cserélhető és az ellenállás mindkét oldalára terveztem elvezetéseket.



2.3.6. Ábra – Low-dropout lineáris szabályzó

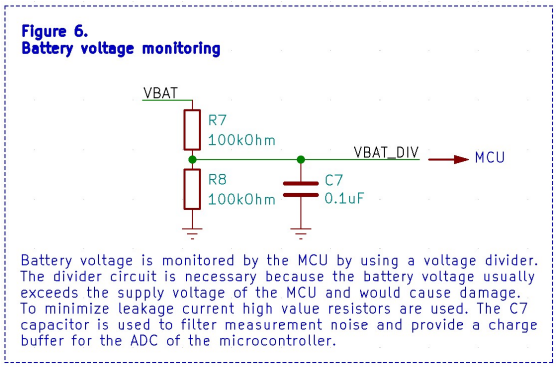
2.3.4. Tápellátás megfigyelése

Egy akkumulátoros üzemű IoT eszköz fejlesztése során szükség lehet arra, hogy az eszköz folyamatosan megfigyelhesse a saját tápellátásának állapotát. Egyszerűbb esetben ez megnyilvánulhat az akkumulátorfeszültség méréseként, bonyolultabb esetekben pedig teljeskörű töltés- és egészségi állapot megfigyeléseként is. Az utóbbi tématerület folyamatos kutatás tárgyát képezi és egyáltalán nem magától értetődő folyamat. A direkt erre a célra dedikált áramkörök az akkumulátor feszültségének és áramának nyomon követésével következtetnek annak pillanatnyi egészségi állapotára és megbecsülik a még rendelkezésre álló pillanatnyi kapacitást. Ennek megvalósítása általában rendkívül bonyolult és a dedikált integrált áramkörök költségei is viszonylag magasak.

A szakdolgozatom során ezért az egyszerűbb megoldást, az akkumulátorfeszültség mérését alkalmaztam, melynek megvalósítása a 2.3.7. ábrán szerepel. A feszültség figyelésének legegyszerűbb módja, ha a mikrokontrolleren található ADC[3] segítségével mérjük a

[1] FET: Field-effect transistor, magyarul: térvezérlésű tranzisztor
[2] LDO: Low Dropout, magyarul: kis feszültségesésű stabilizátor
[3] ADC: Analog-digital converter, magyarul: analóg-digitális átalakító

feszültséget, amely az analóg feszültséget egy digitálisan feldolgozható számmá alakítja mintavételezés segítségével. Mivel a mikrokontroller 3.3V feszültségről üzemel, az akkumulátor feszültsége pedig teljesen feltöltött állapotban akár a 4.2V is lehet, ezért a közvetlen mérés nem lehetséges. A probléma áthidalására egy ellenállás-osztót alkalmaztam, amely megfelezi az akkumulátor feszültségét az ADC bemenetén. A felhasznált ellenállások értékét megfelelően magasra választottam, ezzel minimalizálva az osztón elszivárgó áram nagyságát, így csekély mértékben merítve csak az akkumulátort. Névleges akkumulátorfeszültséget feltételezve a várható szivárgó áram nagysága mindössze 18.5 µA. Az ADC bemenetére továbbá egy 100 nF értékű kondenzátort is elhelyeztem, mivel a nagy ellenállások miatt folyamatos mérés esetén nem jutna elég töltés az ADC belső kapacitását feltölteni, valamint a kondenzátor segít a tápfeszültség mérési zajának csökkentésében is. Mivel általában a mikrokontrollerek ADC-i viszonylag zajos méréseket végeznek, így esetlegesen szükség lehet szoftveres utófeldolgozásra a későbbiekben.



2.3.7. Ábra – Akkumulátorfeszültség mérése

2.4. Adattárolás és megjelenítés

Az internetes hálózatra történő adattovábbítás mellett szükségünk lehet helyi adattárolás megvalósítására. Ennek több módja létezik, melyek közül a tárolni kívánt adatok mennyisége, formátuma és a tárolás időtartama alapján választhatjuk ki a megfelelőt.

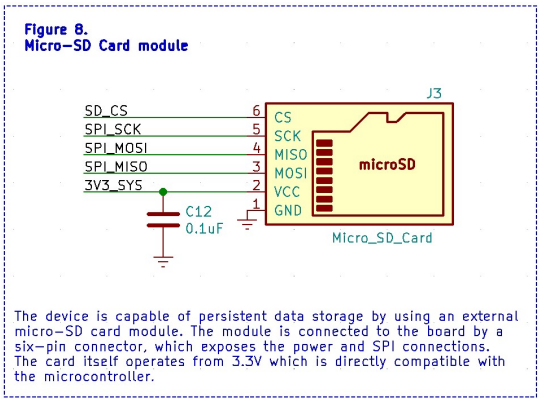
- **RAM[1]:** Amennyiben a tárolás csak ideiglenes és nagyon rövid ideig van szükségünk az adatokra, megtehetjük, hogy a mikrokontroller dinamikus memóriájában, RAM-ban tároljuk azokat. A gyakorlatban sokszor előfordul, hogy a mérési adatokat ideiglenesen bufferelјük a memóriában, így azok továbbításakor egyszerre sok adatot is elküldhetünk, így minimalizálva a hálózaton keresztül történő műveletek számát. A megoldás előnye, hogy az így tárolt adatok egyszerűen hozzáférhetők, hátránya viszont, hogy csak limitált adatmennyiség (kb. néhányszáz kB) rövid idejű tárolására alkalmas.
- **FLASH:** Manapság a legtöbb komolyabb mikrokontroller rendelkezik valamennyi szabad FLASH-memóriával, néhány esetben pedig külön integrált áramkörként illeszthetjük azt a rendszerünkbe. Ennek a tárolási módnak egyik fontos előnye, hogy nagyobb mennyiségű adat (kb. néhány MB) tárolását teszi lehetővé

[1] RAM: Random-access memory, magyarul: véletlenszerű elérésű memória

hosszú ideig. Hátránya, hogy FLASH-memóriák többnyire csak nagyobb blokkokban írhatók, az írások száma limitált és az adatok nem hordozhatók.

- **SD-kártya[1]:** A mai modern SD-kártyák kapacitása messze meghaladja az átlagos FLASH-memóriák méretét, akár GB nagyságrendben képesek adatok tárolására. Az így lementett mérések hordozhatók és más eszközökön is megtekinthetők. Hátrányuk, hogy jelentősen drágábbak az egyszerűbb megoldásoknál és kezelésük is bonyolultabb.

A mérési adatok egyszerű felhasználhatóságának és hordozhatóságának érdekében SD-kártyás tárolás mellett döntöttem. Ezek az eszközök saját kommunikációs protokollt alkalmaznak, de akár hagyományos SPI[2] buszon keresztül is vezérelhetők. A megvalósított hardver esetében a beültetés és forrasztás megkönnyítése érdekében egy előre elkészített SD-kártya modult használtam, amely kivezeti az SPI kommunikációs vonalakat. A kapcsolás a 2.4.1. ábrán látható.



2.4.1. Ábra – SD-kártya modul SPI buszon

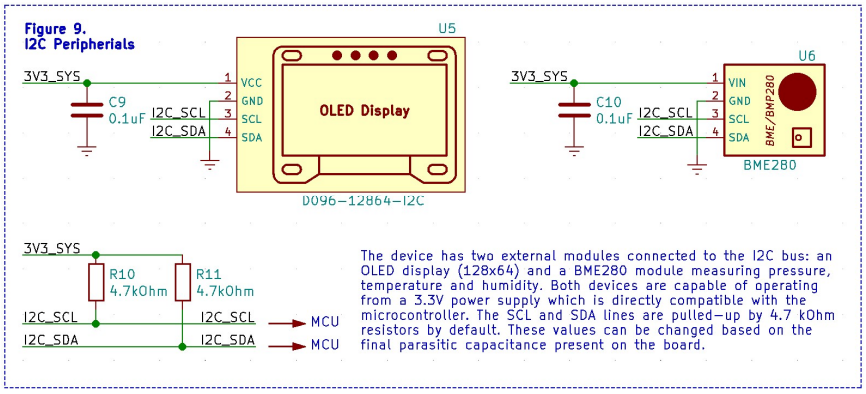
Előfordulhat, hogy az eszközünk által mért vagy egyéb módon gyűjtött adatokat szeretnénk az internetre történő továbbítás mellett helyben is megjeleníteni. Erre a célra általában valamilyen kijelzőt alkalmazunk. A gyakorlatban többnyire az alábbi kijelző típusok jöhetnek szóba.

- **Szegmens-kijelzők:** Egyszerű, rövid numerikus értékek, esetleg kisfelbontású karakterek megjelenítésére alkalmas eszköz. Működési elve, hogy a kijelzést néhány előre meghatározott formájú szegmens alkotja, melyeket ki-be kapcsolhatunk. Tipikus példája a régi digitális ébresztőórák kijelzői.
- **Karakteres LCD[3]-kijelzők:** Bonyolultabb szövegek, számok, esetleg szimbólumok megjelenítésére használható. A kijelző viszonylag nagyméretű pixelekből áll melyek csak aktív állapotban láthatóak. Hátrányuk, hogy magasabb fogyasztásuk van, mivel többnyire folyamatos háttérvilágítást igényelnek.

[1] SD: Secure Digital, magyarul: biztonságos digitális
[2] SPI: Serial Peripherial Interface, magyarul: soros periféria interfész
[3] LCD: Liquid-crystal display, magyarul: folyadék-kristályos kijelző

- **OLED[1]-kijelzők:** Ezek a kijelzők általában sokkal több és kisebb méretű pixelekből állnak, ezért alkalmasabb hosszabb szövegek, számok és bonyolultabb ábrák megjelenítésére is. További előnyük, hogy nem igényelnek háttérvilágítást és csak azok a pixelek fogyasztanak, amelyek ténylegesen aktívak, így minimális áramfelvétellel rendelkeznek.

A szakdolgozatom során egy 128x64-es felbontású OLED-kijelző használata mellett döntöttem, amely modul kivitelben állt rendelkezésre és I2C kommunikációs protokollt alkalmaz. Az alacsony fogyasztás tökéletesen megfelel az akkumulátoros felhasználás követelményeinek, valamint a magas felbontás lehetővé teszi bonyolultabb grafikák és nagyobb mennyiségű információ megjelenítését is. A kijelző logikáját az SSD1306 meghajtóáramkör adja, melyhez korábbi tanulmányaimból már rendelkezésre állt illesztőprogram. Az I2C buszhoz csatlakozó perifériákat, valamint a hozzájuk tartozó felhúzóellenállások bekötését a 2.4.2. Ábra mutatja.



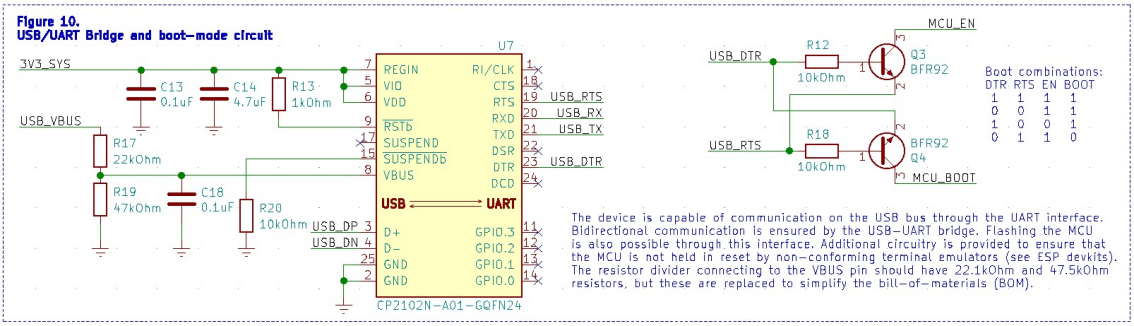
2.4.2. Ábra – Kijelző és szenzormodul az I2C buszon

2.5. Programozás és soros kommunikáció

A legtöbb mikrokontroller esetében az elkészült programkód feltöltése az eszközre nem feltétlenül triviális feladat. Sokszor egy speciális, az eszközön előre telepített kódrészlet, az ún. bootloader feladata, hogy elvégezze az eszköz programozását valamilyen külső kommunikációs felületen keresztül. Sok esetben a mikrokontrollerek programozása az UART periférián keresztül történik. A mikrokontrolleren futó kódot általában számítógép segítségével írjuk, így biztosítani kell, hogy a számítógépünket össze tudjuk kötni a mikrokontrollerrel. A legtöbb fejlesztői kártya esetében ez USB segítségével történik, ehhez azonban biztosítani kell az USB és az UART kommunikáció közötti kétirányú fordítás lehetőségét. A fejlesztés során a hibakeresést is megkönnyíti, ha a mikrokontroller által generált hibaüzenetek, logok és egyéb szabványos kimenetre írt információk számítógép segítségével megtekinthetők. Ezt a feladatot általában USB-UART átalakító integrált áramkörök alkalmazásával szokás megoldani. A szakdolgozatom során a hivatalos Espressif fejlesztőkártyákon is alkalmazott USB-UART átalakítót használtam. Mivel a gyártó által rendelkezésre bocsátott programozó a kód feltöltése előtt boot-állapotba

[1] OLED: Organic Light-emitting diode, magyarul: organikus fénykibocsátó dióda

hozza, majd utána automatikusan újraindítja a mikrokontrollert (a DTR[1] és RTS[2] vonalak felhasználásával), biztosítani kellett az ehhez szükséges áramköröket is. Néhány terminál-emulátor azonban nem működik jól ezzel a konfigurációval, ezért egy kiegészítő áramkör is szükséges, ami megakadályozza, hogy használatuk során folyamatosan letiltott állapotban tartsák az eszközt. A 2.5.1. ábrán szereplő kapcsolási rajz részlet az ESP32-DevKitC fejlesztői kártyán [14] található USB-UART áramköre alapján készült.

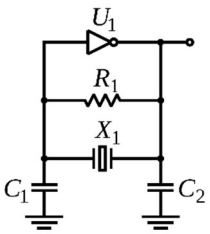


2.5.1. Ábra – USB-UART átalakító és kiegészítő logika

2.6. Egyéb periféria-áramkörök

Az előző alfejezetekben bemutatásra kerültek a legfontosabb áramköri részletek, amelyekre gyakran szükségünk lehet egy IoT eszköz fejlesztése során. Ezeken kívül szükségünk lehet további egyszerűbb áramkörökre is, melyek azonban nem érdemelnek saját fejezetet. Ebben a fejezetben említést teszek a kristályoszcillátorok használatáról, valamint bemutatom a korábban ismertetett áramköri elemek mikrokontrollerhez történő csatlakoztatását is.

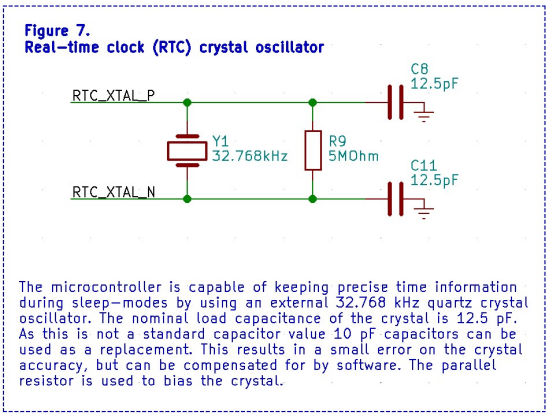
Gyakran szükségünk lehet a dátumra és az idő múlásának mérésére a mikrokontroller segítségével. A szakdolgozatom során felhasznált mikrokontoller alvó-állapotban is képes nyomon követni az idő múlását, azonban alapértelmezés szerint erre a célra egy beépített RC-oszcillátort alkalmaz, melynek pontossága alacsony. Hasonló célokra nagyobb pontosságot biztosító lehetőség valamilyen kvarc-oszcillátor használata. Ezeket a kristályokat általában Pierce-oszcillátor kapcsolásokban alkalmazzák [15], és minimális kiegészítő áramkör szükséges a működtetésükhöz. Az oszcillátorkapcsolás általános felépítését a 2.6.1. Ábra mutatja.



2.6.1. Ábra – Pierce-oszcillátor

[1] DTR: Data Terminal Ready, magyarul: adatterminál készen áll
[2] RTS: Ready To Send, magyarul: adatküldésre készen áll

A kvarckristályok adatlapjaiban megtalálhatjuk azok terhelési kapacitását, melynek alkalmazása kulcsfontosságú a megfelelő pontosság eléréséhez. A kristállyal párhuzamosan, annak mindkét oldalára szokás hangoló kondenzátorokat elhelyezni, melyek értéke ideális esetben megegyezik a kristály adatlapján megadott értékkel. Előfordulhat azonban, hogy a szükséges értékű kapacitás nincs forgalomban, ilyenkor minimálisan eltérhetünk a megadott értéktől a kristály pontosságának csökkenése árán. A szakdolgozatomban felhasznált kristály terhelési kapacitása 12.5 pF, mivel ilyen értékű kondenzátor nincsen forgalomban, ezért 10 pF értékű kapacitásokat alkalmaztam. Mivel a kondenzátorok értékének szórása általában véve is nagy, ezért kisebb darabszámnál ezek mérése is érdemes lehet. Az adatlapon szereplő értékektől való nagyobb eltérés azonban nem elfogadható, mivel ez elronthatja az oszcillátor stabilitását vagy akár el sem tud indulni az oszcilláció. Fontos megemlíteni, hogy a ténylegesen előforduló kapacitást nem csupán ezek a kondenzátorok határozzák meg, hanem a végleges nyomtatott áramkör és a mikrokontroller belső felépítése által okozott parazitakapacitások is. A párhuzamos kondenzátorok mellett általában szokásos egy nagy értékű (MOhm nagyságrendű) visszacsatoló ellenállást is elhelyezni, melynek feladata az oszcillátorkapcsolásban található erősítő szaturációjának megakadályozása és annak lineáris tartományban tartása. A szakdolgozatom során megvalósított oszcillátor kapcsolás a 2.6.2. ábrán látható.

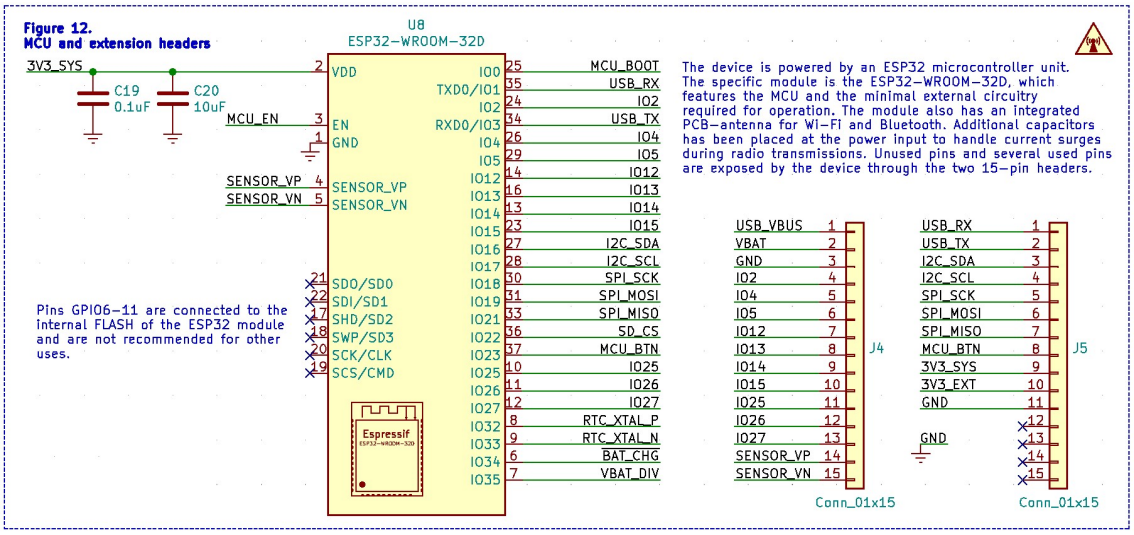


2.6.2. Ábra – Kristályoszcillátor kapcsolás

Miután megterveztük az eszközünk számára szükséges valamennyi áramköri részletet, a folyamat végén hozzáláthatunk azok a mikrokontrollerhez történő csatlakoztatásához. Ez többnyire egy iteratív folyamat, sokszor előfordul, hogy a nyomtatott áramkör tervezése során vesszük észre, hogy egy-egy csatlakozás elhelyezkedése kényelmetlen vagy nem praktikus. Az egyes feladatok megvalósításakor célszerű az adott probléma megoldásához minimálisan elegendő kivezetéseket felhasználni. Próbáljuk meg elkerülni tehát a különleges funkcióval (pl. ADC, PWM[1]) rendelkező lábak használatát, amennyiben a feladathoz egy „egyszerűbb” láb is rendelkezésre áll. A folyamat végén általában érdemes a nem felhasznált lábakat és egyes felhasznált lábakat is valamilyen csatlakozó segítség-

[1] PWM: Pulse Width Modulation, magyarul: impulzusszélesség moduláció

ével kivezetni, ezzel elősegítve az esetleges későbbi bővíthetőséget és az áramkör tesztelését, hibakeresést. A dolgozat során megtervezett hardver által alkalmazott végleges lábkiosztás a 2.6.3. ábrán szerepel.



2.6.3. Ábra – Mikrokontroller lábkiosztás és kivezetések

2.7. Nyomtatott áramkörtervezés

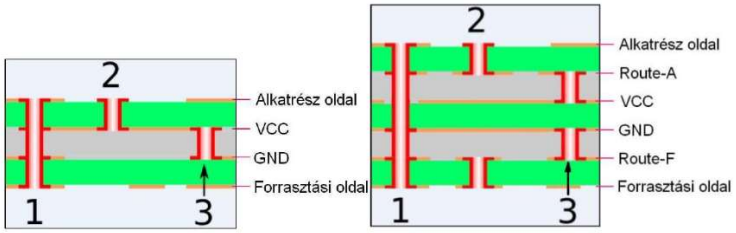
A kapcsolási rajz terveinek elkészítése után következhet végül a tényleges nyomtatott áramkör megtervezése. Ez a folyamat talán az egyik legbonyolultabb feladat egy elektronikai eszköz fejlesztése során, rendkívül sok dologra kell odafigyelnünk. A tématerülettel egész könyvek [16] [17], publikációk foglalkoznak, a továbbiakban jelentősen egyszerűsítve összefoglalom a legfontosabb tudnivalókat a különféle rétegektől az egyszerűbb ökölszabályokig.

2.7.1. Rétegezés

A gyakorlatban alkalmazott nyomtatott áramköri panelek általában rétegezett kivitelben kerülnek gyártásra. A rétegek száma alapján megkülönböztethetünk egyszerűbb és bonyolultabb felépítésű paneleket [4]. A legegyszerűbb kialakításnál csak a nyomtatott áramkör egyik oldalán találhatóak vezetékek és alkatrészek, ezeket nevezzük egyoldalas áramköröknek. A megoldás előnye, hogy gyártása rendkívül egyszerű és olcsó, ha nem alkalmazunk túlságosan kicsi vezetéktávolságokat, akár otthon is elkészíthetők a megfelelő eszközökkel. Legjelentősebb hátrányuk, hogy csak síkba rajzolható hálózatok megvalósítására alkalmasak, tehát ha a tervünk tartalmaz egymást keresztező vezetékeket, mindenképpen legalább kétoldalas kialakításra van szükségünk. Kétoldalú nyomtatott áramkörök esetén lehetőségünk van az alsó és a felső oldalon is vezetékeket és alkatrészeket elhelyezni, a két oldal közötti kapcsolatot furatok vagy viák[1] teremtik meg. Gyakorlatilag bármilyen nyomtatott áramkör elkészíthető kétoldalas kialakítással a megfelelő méretben, általában kompromisszumos megoldásként így tervezzük az egyszerűbb áram-

[1] via: belső oldalán vezetőréteggel bevont, kis átmérőjű furat

köröket. A tervezési folyamat megkönnyítése és a panel méretének (ezáltal gyártási költségének) csökkentése érdekében akár kettőnél több réteget is alkalmazhatunk, gyakoriak a 4-6 réteges megoldások is. Ezek leggyakoribb felhasználási módja, hogy a tápfeszültségen és földpotenciálon lévő pontok a két belső rétegen kapnak helyet, 4 rétegnél bonyolultabb kialakításnál pedig több lehetőségünk van a vezetékek nyomvonalának optimalizálására. Ennek a kialakításnak egyik legnagyobb előnye, hogy jelentősen egyszerűsíti a jelvezetékek elhelyezését, kedvezően befolyásolja az eszközünk földelési tulajdonságait és csökkenti az eszköz által kibocsátott elektromágneses zajt. Hátránya a kétoldalas nyomtatott áramkörökhöz képest a bonyolultabb és ezáltal költségesebb gyártás. A kettőnél több réteget alkalmazó panelek esetében a viák kialakítása is bonyolultabb, mivel előfordulhat, hogy nem a két külső réteget kell összekötniük. Egy külső és egy belső réteg összekapcsolása esetén zsákviáról, két belső réteg esetén pedig eltemetett viáról beszélhetünk. A különféle réteg- és viakialakításokat a 2.7.1. Ábra mutatja.



2.7.1. Ábra – Négy- (balra) és hatrétegű (jobbra) kialakítások, furattípusok (via: 1, zsákvia: 2, eltemetett via: 3) [4]

A nyomtatott áramköri panelt általában valamilyen CAD[1] szoftver segítségével tervezzük meg a kapcsolási rajz alapján. A szoftver általában a megfelelő hozzárendelések után automatikusan elhelyezi az alkatrészek rajzolatát (footprint) a panelen, azonban a pontos elrendezést nekünk kell kialakítanunk. A komolyabb szoftverek már rendelkeznek automatikus vezetékelhelyezési funkcióval is, de a legtöbb esetben manuálisan kell összekötnünk az alkatrészeket. A tervezés során több, elektronikus és nem elektronikus funkciójú réteget is felhasználunk, a legfontosabbak az alábbiak:

- **Top:** A nyomtatott áramköri panel felső oldala, általában alkatrészelhelyezésre és vezetékek elhelyezésére használjuk.
- **Bottom:** A panel alsó oldala, kettő vagy több réteges kialakításnál szintén alkatrészeket és vezetékek kialakítására használható.
- **Silkscreen:** Az alsó és felső oldalon is elhelyezhető, szöveges feliratok és grafikus ábrák elhelyezésére használható, általában az alkatrészek körvonalának jelölésére, azok azonosítására vagy magyarázatokhoz használjuk.
- **Edge cuts:** A panel szélének jelölésére használható, több áramkör együttes gyártása esetén az egyes paneleket egymástól el kell választani, ebben segíthet, ha pontosan jelöljük az áramkör széleit.

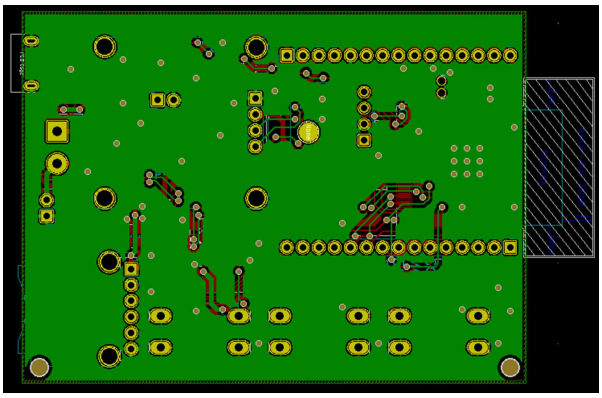
[1] CAD: Computer-aided Design, magyarul: számítógéppel segített tervezés

A felsoroltakon kívül általában további rétegek is rendelkezésünkre állnak, némelyik akár automatikusan frissülhet, azonban ezek a tervezéshez használt szoftvertől függően változhatnak.

2.7.2. Ökölszabályok

A nyomtatott áramkörök tervezése bonyolult folyamat, a gyártástechnikai és fizikai szempontok széles skálája befolyásolja a tervezés menetét. Az elkészült áramkör viselkedésének teljeskörű szimulációja rendkívül körülményes, a legtöbb esetben viszont egyszerű ökölszabályok alkalmazásával is jó eredményt érhetünk el. A továbbiakban ismertetek néhány fontos követendő módszert, melyek hasznosak lehetnek bármilyen nyomtatott áramkör tervezésre során.

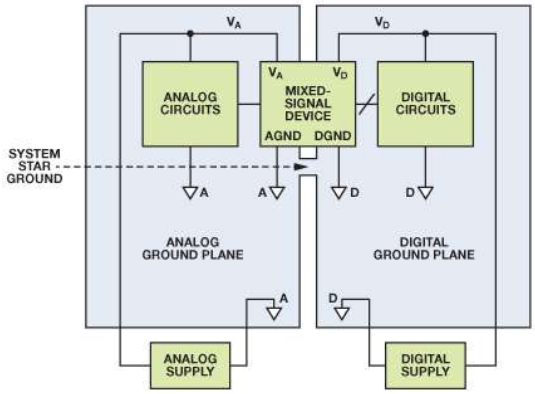
Az egyik legfontosabb aspektusa a nyomtatott áramköri panelek tervezésének, a megfelelő földelés és tápellátás kialakítása. Kettőnél több réteg esetén ennek megvalósítása nem jelent problémát, mivel dedikált föld- és tápfelületek állnak rendelkezésre a belső rétegeken. Kétoldalas kialakításnál azonban gyakran használt megoldás, hogy kizárólag a felső oldalon helyezünk el alkatrészeket és vezetékeket, illetve az alsó oldalon teljes egészében földkitöltést alkalmazunk. Ez megfelelő árnyékolást és alacsony impedanciás áramvisszavezetést biztosít az eszköz számára és sok esetleges későbbi problémát orvosolhat. Természetesen szükségünk lehet vezetékek kialakítására az alsó oldalon is, ilyenkor igyekezzünk ezeket minél rövidebbre választani, hogy a lehető legkevésbé zavarják földelési pont felé visszafolyó áram útját, ahogy a 2.7.2. Ábra is mutatja.



2.7.2. Ábra – Egybefüggő alsó földkitöltés, ritkán elhelyezett vezetékekkel

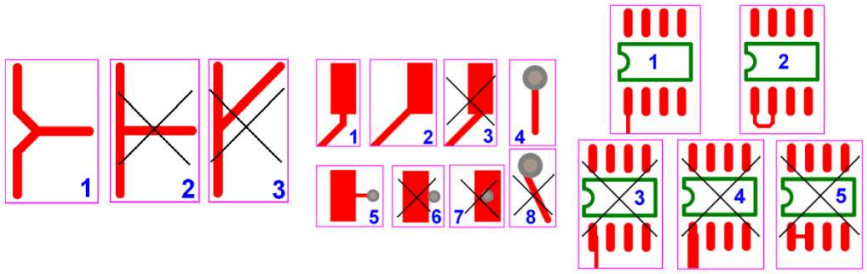
Igyekezzünk elkerülni az ún. földhurkok kialakítását, ugyanis az ilyen hurkokban folyó áramok változást okozhatnak az egyes pontok földpotenciáljában, ami komoly és nehezen orvosolható problémákat okozhat. Földhurkok általában akkor alakulnak ki, ha az egyek komponensek földelését több pontban is összekötjük, mielőtt az alacsony impedanciás földeléshez csatlakozunk. Amennyiben analóg és digitális áramköri elemeket is használunk, ezek földelését érdemes lehet különválasztani a 2.7.3. ábrának megfelelően, hogy a digitális alkatrészekből származó nagyfrekvenciás zajok ne zavarják az analóg áramkörök viselkedését. Ebben az esetben sokszor különálló földfelületeket alakítanak ki, majd ezeket egyetlen pontban, a tápforrás földelési pontjában, vagy vegyes analóg-digitális esz-

közök közelében egyesítik. Az alkatrészek és vezetékek végleges elhelyezése után szokásos földkitöltést alkalmazni a fennmaradó üres területekre, ez segíti a megfelelő árnyékolás és kedvező termikus tulajdonságok kialakítását.



2.7.3. Ábra – Elválasztott analóg és digitális földelések [18]

A vezetékek és alkatrészek elhelyezésénél is fontos néhány dologra odafigyelni, melyeket a 2.7.4. ábra foglal össze [4]. Amennyiben lehetséges, ne törjük meg a nyomvonalakat derékszögben, ugyanis a 90 fokos törések reflexiót okoznak a nagysebességű jelvezetékekben. Szintén kerüljük el a vezetékek kis szögben történő csatlakoztatását, ugyanis az elektrosztatikából ismert csúcshatás jelentős potenciáleltéréseket okozhat. A termikus és beültetési problémák elkerülésének érdekében tartózkodjunk az alkatrészlábak közvetlen viával történő csatlakoztatásától a belső nagyfelületű tápfelületekhez. Hasonló okokból elkerülendő az alkatrészek lábainak csatlakoztatásakor a lábak felületének méretét megváltoztató vezetékszélesség alkalmazása is. Igyekezzünk továbbá a tápfeszültségszűrő kondenzátorokat az alkatrészek lábaihoz minél közelebb elhelyezni, illetve figyeljünk oda a differenciális jelvezetékek esetében az azonos vezetékhosszúság biztosítására.



2.7.4. Ábra – Vezetékelhelyezési ökölszabályok [4]

2.7.3. Ellenőrzés, gyártási fájlok

Miután elkészültünk a PCB terveivel, a végső ellenőrzés után megkezdődhet a gyártás. A gyárthatósági paramétereket mindenképpen ellenőriznünk kell, ennek célja, hogy észrevegyük azokat a hibákat, amik miatt az áramköri panel gyártása esetleg nem lehetséges. A leggyakoribb hibák a különféle távolsági követelmények megszegése (pl. túl közel lévő

vezetékek, viák, alkatrészek). Ezek ellenőrzésére a CAD szoftverek automatikus támogatást nyújtanak, ezek a DRC[1] funkciók. Amennyiben házilag szeretnénk elkészíteni a nyomtatott áramkört, különleges felszerelésre van szükségünk és a folyamat is viszonylag bonyolult. Az esetek döntő többségében valamilyen erre specializálódott gyártót keresünk, akinek a nyomtatott áramkör terveit megfelelő formában biztosítva legyártják nekünk a panelt. A PCB gyártók többsége azonban nem nyers formában várja a terveket, ahogy azt az általunk választott CAD szoftverben elkészítettük, a gyártás automatizálására szabványosított fájlformátumok terjedtek el. A legfontosabb gyártási fájlok valamennyi szoftverből exportálhatók:

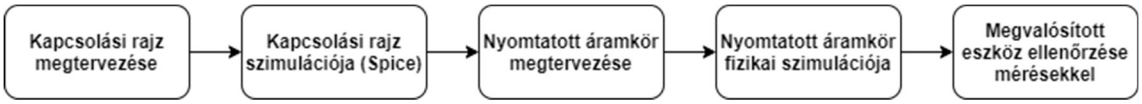
- **Gerber fájlok:** A nyomtatott áramköri panel egyes rétegeinek kialakítását írja le. Minden egyes rétegben meghatározza a hozzá tartozó vezetékek, alkatrészlábak és egyéb vezető felületek alakját, gyártási maszk formájában. Ezeket a maszkokat a gyártás során használt fotolitográfiás eljárások során használják.
- **Drill fájlok:** A panelen található valamennyi furat és via pontos pozícióját és méreteit tartalmazza, a fúráshoz használt CNC[2] gépek vezérlésére használják. A furatok méretének kiválasztásakor fontos figyelembe venni, hogy a tényleges furatátmérő galvanizálás után kisebb lesz, mint a specifikált méret.
- **Mechanikai fájlok:** Ezek a fájlok tartalmazzák a panel mechanikai paramétereit, például annak külső körvonalát, esetleg a panel belsejében található kivágott részek alakját.
- **Centroid fájlok:** Az alkatrészbeültetés során használják, Pick and Place fájloknak is nevezik. Az alkatrészeket ahol automatizált beültetőgépek helyezik el a panelen, ezek pozícióját, típusát, értékeit és forgatását tartalmazza.
- **BOM[3] fájlok:** A nyomtatott áramköri panelon található alkatrészek pontos, tételes listája. Feltüntetjük rajta az alkatrész kapcsolási rajzon használt azonosítóját, értékét és pontos típusát. Általában valamilyen táblázatkezelők által használt formátumban készítik el, például Excel-fájl vagy CSV[4].

2.8. Értékelés, fejlesztési lehetőségek

Általában a megtervezett hardver gyártásba kerülése előtt számos egyéb ellenőrzési fázison kell, hogy végighaladjon. Egy komolyabb áramkör kapcsolási rajzának megtervezése után az első lépés az áramkör viselkedésének részletes és pontos szimulációja. Ennek segítségével képet kaphatunk az áramkörünk viselkedéséről még a tényleges fizikai megvalósítás előtt. Erre a célra általában valamilyen Spice szoftvert szoktunk alkalmazni, sokszor a kapcsolási rajz szerkesztő program beépítve tartalmaz szimulációs lehetőségeket. Egy mikrokontrollert is tartalmazó, dinamikus logikával rendelkező eszköz teljeskörű szimulációja nehéz feladat, mivel a mikrokontrolleren futó szoftver viselkedését a szimulációs programok nem képesek közvetlenül modellezni. Az ilyen esetekre különböző, mesterségesen előállított teszteseteket kell definiálnunk. Mivel a szakdolgozatom

[1] DRC: Design Rule Check, tervezési szabályok ellenőrzése
[2] CNC: Computer Numerical Control, numerikus értékekkel vezérelt számítógépes program
[3] BOM: Bill of Materials, magyarul: alkatrészlista
[4] CSV: Comma-separated Values, magyarul: vesszővel elválasztott értékek, táblázatkezelő fájlformátum

során megtervezett áramkör viszonylag egyszerű, ezért a részletes szimulációtól ebben a példában eltekintettem, az érdeklődő olvasó számára [19] jó kiindulási alapot jelenthet. A kapcsolási rajz áramköri szimulációja után, megtervezzük a nyomtatott áramkört. A megtervezett panel további szimulációkat igényel, ugyanis a konkrét fizikai kialakítás olyan másodlagos impedanciaviszonyokat teremthet, amelyet a kapcsolási rajz alapján lehetetlen vizsgálni. Az egyszerűbb, kisfrekvenciás alkalmazásoknál ezek hatása nem jelentős, azonban a GHz-es tartományban már fontos ezek vizsgálata. Amennyiben az alkalmazás megköveteli, ügyelnünk kell az elektromágneses kompatibilitással (EMC) kapcsolatos problémákra is, melyet szintén a nyomtatott áramkör fizikai szimulációjával érdemes kezdeni. Végül az elkészült panel valóságos mérésekkel történő vizsgálata következik. A tervezés és az ellenőrzés folyamatát a 2.8.1. Ábra mutatja.



2.8.1. Ábra – Hardvertervezés és validáció folyamatábrája

A korábbi fejezetekben megtervezett nyomtatott áramkör a fent említett vizsgálatok eredményeinek függvényében fejleszthető, illetve egyes rejtett hibái javíthatók. Az elkészült hardver tesztelése során nem tapasztaltam semmiféle rendellenességet, ennek ellenére további mérések szükségesek a működés helyességének igazolásához. Egy ennél komolyabb alkalmazásban mindenképpen javasolt kettőnél több rétegű nyomtatott áramköri panel tervezése, a különféle szimulációk elvégzése és részletes dokumentációja. Sorozatgyártásba kerülő termékek, vagy automatikus alkatrészbeültetés alkalmazása esetében érdemes kisebb méretű alkatrészeket használni, ezzel csökkentve az áramköri panel méretét és növelve az alkatrészsűrűséget. Ennek leginkább költséghatékonysági előnyei vannak. A gyártás szempontjából további szempont, hogy igyekezzünk azonos kivitelű, méretű és típusú alkatrészeket használni, mivel beültetéskor a különböző típusú alkatrészek közötti váltás növeli a gyártás fajlagos költségeit és időtartamát. Hasonlóképpen a furatok és viák tervezésekor is érdemes azonos furatátmérőt alkalmazni, mert a fúrófejek cseréje is időigényes folyamat és növeli a gyártási költségeket.

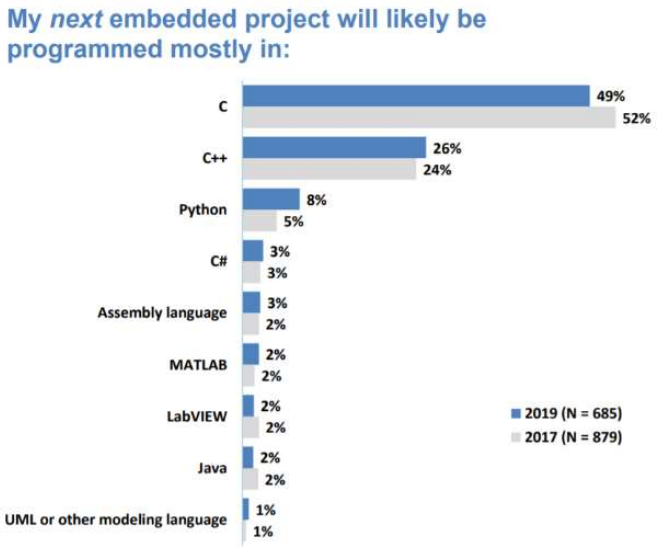
A korábbi fejezetek során összefoglaltam az IoT eszközök legfontosabb hardvertervezési tudnivalót. Az Olvasó megismerkedhetett az eszközt működtető mikrokontroller és szenzorok kiválasztásának szempontjaival, a megfelelő tápellátás megtervezésével, az adattárolás és megjelenítés különböző módjaival, illetve a hardveres kommunikációs lehetőségekkel. Végül ismertetésre kerültek a nyomtatott áramkörtervezés alapjai. A megtervezett kapcsolási rajz és teljes nyomtatott áramkör tervei a dolgozat végén az 5.1 és az 5.2 függelékekben megtalálhatók. A következő nagyfejezetben az elkészült hardveren futó szoftver tervezésének lépéseit mutatom be.

3. Szoftver

Az előző nagyfejezetben megtervezett hardver önmagában nem képes a kívánt funkcionalitás megvalósítására. Gyakorlatilag az összes IoT alkalmazásban szükségünk van valamilyen szintű szoftveres támogatásra. A szoftver feladata, hogy a hardver által nyújtott szolgáltatások használatával, rendezett formában megvalósítsa a kívánt funkcionalitást, illetve biztosítsa a működéshez szükséges kiegészítő logikát. Ebben a nagyfejezetben a beágyazott szoftverfejlesztés alapjaival foglalkozom.

3.1. Programozási nyelv és platform kiválasztása

A beágyazott szoftverfejlesztés egyik legelső lépése a programozási nyelv és platform kiválasztása. A mai modern mikrokontrollerek általában többféle nyelven is programozhatók, de a gyakorlatban leginkább a C és C++ nyelvek a legelterjedtebbek. Ennek egyrészt történelmi okai vannak, a C alapú nyelvek voltak az első széleskörben alkalmazott programozási nyelvek a beágyazott rendszerek születésekor. Annak ellenére, hogy manapság már sok különböző modernebb nyelv is létezik, a 3.1.1. Ábra alapján továbbra is megfigyelhető a C és C++ dominanciája a beágyazott rendszerek piacán. Ennek szakmai oka is van, ugyanis a beágyazott rendszerek többnyire hardverközeli programozást igényelnek, amihez a magasabb szintű nyelvek ritkán nyújtanak megfelelő támogatást. Ennek ellenére megfigyelhető trend, hogy a különféle szkript-nyelvek (leginkább a Python) évről-évre növekvő népszerűségre tesz szert, melynek oka a minél rövidebb fejlesztési- és piacra kerülési időre történő törekvés a vállalatok részéről.



3.1.1. Ábra – Beágyazott programozási nyelvek népszerűsége 2019-ben [20]

A magasabb szintű nyelvek mellett azonban mindenképpen meg kell említenünk az Assembly fejlesztést is. Ez a beágyazott környezetben alkalmazott nyelvek közül a legalacsonyabb szintű, leginkább hardverközeli programozási nyelv. A legtöbb komolyabb be-

ágyazott rendszer tartalmaz valamennyi Assembly nyelven írt forráskódot. Ennek egyszerű oka van, a magasabb szintű nyelvek gyorsabb és kényelmesebb fejlesztést tesznek lehetővé, azonban gyakran előfordulnak rendkívül teljesítményigényes feladatok is, melyek magasszintű megvalósítása túl lassú kódot eredményezne, vagy egyenesen lehetetlen a nyelvi támogatás hiánya miatt (pl. a legtöbb watchdog-időzítő felhúzása speciális utasítássorozatot igényel, melyek magasszintű nyelven nem megvalósíthatók). Ilyenkor az alkalmazás legnagyobb részét valamilyen magasszintű nyelven készítjük el, és a speciális feladatokat Assembly-betétek formájában valósítjuk meg (feltéve, hogy ezekhez elérhető támogatás a magasszintű nyelv részéről).

A szakdolgozatom során választott programozási nyelv a C++ volt. Ennek oka, hogy a szoftver olyan absztrakciókat vezet be, melyek megvalósítása sokkal elegánsabb objektum-orientált környezetben, illetve a fejlesztés jelentős része könnyebben automatizálható a C++ által biztosított sablon-alapú programozási funkcionalitás által. A szakmában dolgozó mérnökök körében gyakran heves vita tárgyát képezi a C és C++ alkalmazhatóságának kérdése. Sokszor hangoztatott érv utóbbi ellen, hogy túlságosan lassú, illetve kiszámíthatatlan memóriafelhasználással rendelkezik. Ez az érvelés alapvetően abból a feltételezésből indul ki, hogy ha a fejlesztőnek lehetősége van rosszul használni a nyelvet, akkor ezt meg is teszi. Ezzel ellentétben, azonban ha megfelelően körültekintően járunk el a választott programozási nyelv alkalmazásakor, szinte valamennyi hátulütőjét el tudjuk kerülni. A C++ nyelv a C bővített változata és hátrafelé kompatibilis, tehát bármilyen C nyelvű kód futtatható C++ környezetben is. Utóbbi nyelv által (elegánsabb szintaktikával) biztosított objektumorientált funkcionalitás egy-az-egyben megvalósítható a C eszköztárával is, így értelmetlen megtagadni az olvashatóbb kód által nyújtott előnyöket. Természetesen megfelelő körültekintéssel kell eljárnunk a C++ egyes részeinek (pl. iostream, tárolók, dinamikus memóriakezelés) használatakor, de a hátrafelé kompatibilitás miatt ilyen esetekben visszatérhetünk a C által biztosított megoldásokhoz, különösen a leginkább hardverközeli feladatok megvalósításakor [21].

A feladat megoldására alkalmazott programozási nyelv kiválasztása után mérlegelnünk kell az adott mikrokontrollerhez elérhető fejlesztési platformokat, keretrendszereket is. Természetesen ez az egymásutániság nem minden esetben életszerű, sokszor az elérhető platformok és keretrendszerek, beágyazott operációs rendszerek határozzák meg a programozási nyelvet, ennek kezelése valamennyire rugalmas gondolkodást kíván meg a fejlesztők részéről. A felhasznált programozási keretrendszerek mérlegelését jelentősen meghatározza a hardvertervezési fázisban kiválasztott mikrokontroller típusa. Manapság a legtöbb mikrokontrollerhez elérhető valamilyen programozási platform, amely a gyártó által előre elkészített, adott mikrokontrollerre szabott szoftverkomponensekkel segíti a szoftver fejlesztését. Ezek a keretrendszerek tipikusan absztrakciót biztosítanak a mikrokontroller perifériáinak (pl. ADC, I2C, UART, PWM) és egyéb funkcionalitásainak (pl. alvó üzemmódok, vezetéknélküli kapcsolatok, eszközillesztők) kezeléséhez. Természetesen ezeket a feladatokat elvégezhetjük saját magunk is, gyakorlatilag bármilyen mikrokontroller kezelhető a regiszterek szintjén, azonban ez a megközelítés rendkívül időigényes és az eszköz (sokszor többezer oldalas) adatlapjának alapos tanulmányozását igényli.

Az előző fejezetben megtervezett hardver az Espressif ESP32 mikrokontrollerét tartalmazza, ehhez két jelentősebb fejlesztési platform áll rendelkezésre:

- **Arduino:** Széleskörben elterjedt, nyílt forráskódú és egyszerűen használható szoftveres platform, amelyet eredetileg az Arduino hardverekhez fejlesztettek, de azóta rengeteg különböző hardvertípushoz születtek portolt verziók. Ennek oka, hogy ezek a kvázi-szabványosított szoftvercsomagok könnyen felhasználhatók, egyszerű és intuitív felépítésűek. A platform használata gyorsan megtanulható és számtalan internetes forrás, segédanyag érhető el hozzá. Egyik jelentős hátránya éppen az egyszerűségéből adódik, komplexebb feladatok megoldására kevésbé alkalmas, illetve általános célú szoftvercsomagként kevés hardverspecifikus funkcionalitást tesz elérhetővé a fejlesztők számára, így főleg kezdők számára ideális.
- **ESP-IDF:** A mikrokontroller gyártójának saját, kifejezetten erre a célra testreszabott szoftveres keretrendszere [22], melyet célzottan IoT alkalmazásokhoz terveztek (IDF – IoT Development Framework). Az ESP32 mikrokontroller valamennyi funkcionalitását elérhetővé teszi a fejlesztők számára, egységes és integrált módon, valamint beépítve támogatja a FreeRTOS beágyazott operációs rendszer használatát is. Megvalósítása nyílt forráskódú, C nyelven készült, így akár C++ környezetben is alkalmazható. Hátránya, hogy kezelése jóval bonyolultabb, mint az Arduino keretrendszeré és nagyobb tapasztalatot kíván meg a fejlesztők részéről, így az elsajátítása is több időt vesz igénybe. Utóbbi problémában segíthet a gyártó által biztosított részletes dokumentáció és példagyűjtemény.

A lehetőségek mérlegelése után végül az ESP-IDF keretrendszer alkalmazása mellett döntöttem. Ennek oka, hogy a feladat megvalósítása során szükséges lehet a mikrokontroller speciálisabb beállításainak, funkcionalitásának kezelése. A keretrendszer használatához szükséges előzetes tapasztalat esetemben nem jelentett problémát, korábban már fejlesztettem beágyazott szoftvereket ESP-IDF használatával.

3.2. Beágyazott szoftverarchitektúrák

Beágyazott rendszerekben, mint amilyenek az IoT eszközök is, többféle elterjedt szoftverarchitektúra létezik, amelyek alapján megtervezhetjük a mikrokontrolleren futó programot. Az asztali számítógépen futó programokhoz képest jelentős eltérés, hogy a beágyazott szoftverek általában vég nélkül futnak, mivel azok befejezésekor nincsen más futó alkalmazás. Amennyiben a szoftver mégis befejezi a működését és visszatér, a mikrokontroller jellemzően újraindul. A gyakorlatban többféle egyszerűbb és bonyolultabb architektúrát is alkalmazunk, ezek kiválasztása a megvalósítandó feladat bonyolultságától és egyéb igényeitől függ. A megfelelő szoftverarchitektúra kiválasztása kulcsfontosságú, ugyanis a legtöbb esetben nem létezik minden eshetőségre alkalmazható megoldás. A fő probléma a beágyazott rendszerek alapvető működésében keresendő, miszerint a feladat a külvilágból érkező jelek, ingerek észlelése, feldolgozása és az azokra történő reakció. A mikrokontroller processzorának működése szekvenciális, azonban a külvilág eseményei aszinkron módon érkeznek. A váratlan időpontokban érkező események kezelése robosztus szoftverarchitektúrát igényel, valamint fontos feladat a megfelelő reakcióidő

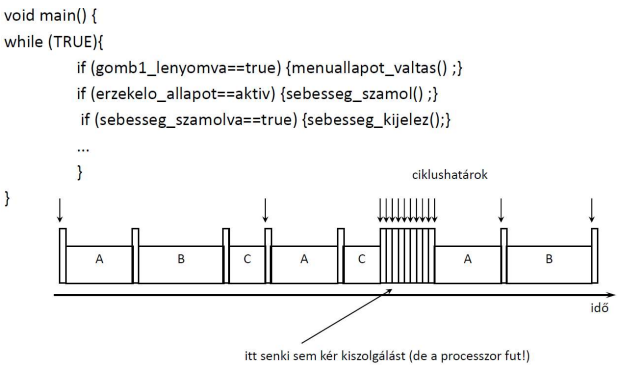
biztosítása is, ami egyes alkalmazásokban akár kritikus is lehet (pl. automatikus vészfékezés akadály hirtelen észlelésekor). A beágyazott szoftver által végrehajtott feladatokat tehát ütemezni kell, melynek megvalósítására sokféle megközelítés létezik. A szoftverarchitektúra kiválasztásánál többek között az alábbi szempontokat [9] érdemes figyelembe venni:

- **Számítási kapacitás:** Fontos szempont az ütemezési algoritmus kiválasztásánál, hogy mekkora számítási kapacitást igényel kizárólag a feladatok ütemezése. Amennyiben egyszerűbb algoritmust választunk, az minimális többletterhelést jelent a rendszerünk számára, viszont számolnunk kell az egyszerű megvalósítások hiányosságaival. Bonyolultabb ütemezés esetén általában lehetőségünk van közel optimális megoldást elérni, viszont ennek nyilvánvalóan a komplexitással egyre növekvő számítási igények szabnak határokat.
- **Prioritások kezelhetősége:** A rendszereinkben általában nem csak egy feladatunk van, hanem léteznek fontosabb és kevésbé fontos feladatok. Egy intelligens jármű esetében például fontosabb feladat lehet a blokkolásgátló működtetése, míg kevésbé kritikus feladat az ablaktörlő lapátok automatikus bekapcsolása. A fontosabb feladatokat általában gyakrabban kell végrehajtani, vagy biztosítani kell, hogy egy alacsonyabb prioritású feladat végrehajtása közben is át tudjunk váltani a fontosabb feladatra, amennyiben erre szükség van. Az elvégzendő feladatok számára tehát prioritásokat kell definiálnunk és a szoftverarchitektúrának képesnek kell lennie ezek figyelembe vételére.
- **Reakcióidő, megszakíthatóság:** Időkritikus beágyazott rendszerekben kiemelt fontosságú, hogy a külvilágból érkező egyes aszinkron eseményekre egy előre kiszámítható, véges és rövid határidőn belül reagáljon a szoftver. Ennek jelentőségét nem lehet alábecsülni, mert akár emberéletek múlhatnak azon, hogy a rendszer időben észrevegye a külvilágból érkező jelzéseket és reagáljon azokra. Mindazonáltal a rendszerben előfordulhatnak alacsony prioritású feladatok is, melyeket időközönként el kell végezni. Amennyiben a program éppen egy alacsony prioritású feladat végrehajtásán dolgozik, akkor is biztosítanunk kell, hogy a fontosabb események kezelése időben megtörténjen, akár a korábban elkezdett feladat félbehagyásának árán is. Erre kiváló lehetőséget biztosítanak a mikrokontrollerek megszakításvezérlői.
- **Erőforráskihasználtság:** A beágyazott rendszerek egyik legfontosabb közös tulajdonsága az erőforrások szűkössége. Általában kevés memóriával és számítási kapacitással rendelkeznek, ezért mindenképpen figyelnünk kell ezek megfelelő kihasználására. Amennyiben a feladat megkívánja, biztosítanunk kell, hogy a szoftverarchitektúra minimális extra memóriát használjon fel a feladatok ütemezéséhez. A memória mellett a másik fontos erőforrás az ütemezéshez rendelkezésre álló processzoridő, ami általában szintén limitált. Akkumulátorról táplát eszközök esetében szintén előnyös, ha az architektúra nem igényel folyamatos futást a szoftver részéről, így a felszabaduló időben a mikrokontroller akár alvó állapotba is tehető, ezzel is energiát megtakarítva és növelve az akkumulátor töltésének gyakoriságát.

- **Bővíthetőség, tervezhetőség:** A fejlesztési folyamat során gyakran szembesülünk újabb problémákkal, amelyekre kezdetben akár nem is gondoltunk. Ez előfordulhat egyszerű figyelmetlenség miatt, de akár rajtunk kívülálló okokból is, például a megrendelői igények megváltozása esetén. Ebben a helyzetben rendkívüli fontosságú, hogy a meglévő feladatokat könnyedén módosíthassuk, illetve egyszerűen adhassunk hozzá újabb feladatokat a rendszerhez. Nem elhanyagolható szempont továbbá, hogy az újonnan definiált feladatok hozzáadása hogyan befolyásolja a jelenlegi feladatok működését, az alkalmazott szoftverarchitektúrának biztosítania kell, hogy az eddig működő szoftver integritása ne sérüljön az új feladatok hozzáadása után sem.

3.2.1. Ciklikus programszervezés

A legtöbb ütemező algoritmus valamilyen ciklikus működési elven működik. A legegyszerűbb megvalósítás a hagyományos körforgó-elv (round-robin) alkalmazása, melynek lényege, hogy az előforduló feladatokat egymás után sorozatosan végrehajtjuk, majd ezt a folyamatot ismételgetjük. A megvalósítás legegyszerűbb formáját a 3.2.1. Ábra szemlélteti. A megoldás legnagyobb előnye, hogy felépítése rendkívül egyszerű, memóriaigénye minimális, az új feladatok könnyedén hozzáadhatók, és a feladatok közötti kommunikáció is egyszerűen megvalósítható megosztott változók használatával. Mivel ebben a felállásban sosem futhat egyszerre két feladat, csak szigorúan egymás után, ezért az erőforrások kezelése is egyszerű, nem szükséges kölcsönös kizárást biztosítani. A mikrokontroller perifériáit folyamatos, periodikus lekérdezéssel tudjuk kezelni. Ez a megoldás azonban jelentős hátrányokat is hordoz magában, ugyanis a feladatok számának növekedésével folyamatosan romlik a rendszer válaszideje, mivel egy feladat következő végrehajtása csak az összes többi feladat befejezése után történhet, illetve önmagában a prioritásokat sem képes kezelni az architektúra.

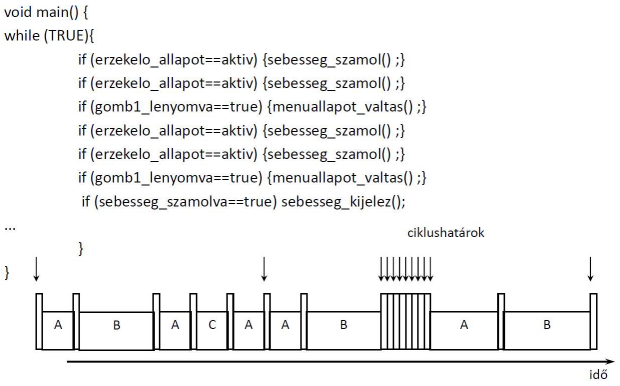


3.2.1. Ábra – Ciklikus körforgó architektúra [9]

Egy egyszerű megoldással képesek lehetünk prioritások bevezetésére, ha ugyanazt a feladatot többször is elhelyezzük a végrehajtási sorban a 3.2.2. Ábra szerint, azonban ennek megvalósítása rendszerint csak manuális beavatkozással lehetséges és kevésbé optimális megoldás. További hátrányt jelent a preempció[1] hiánya, így a megoldás időkritikus feladatokat tartalmazó rendszerben nem működőképes. Az architektúra energiafelhasználási

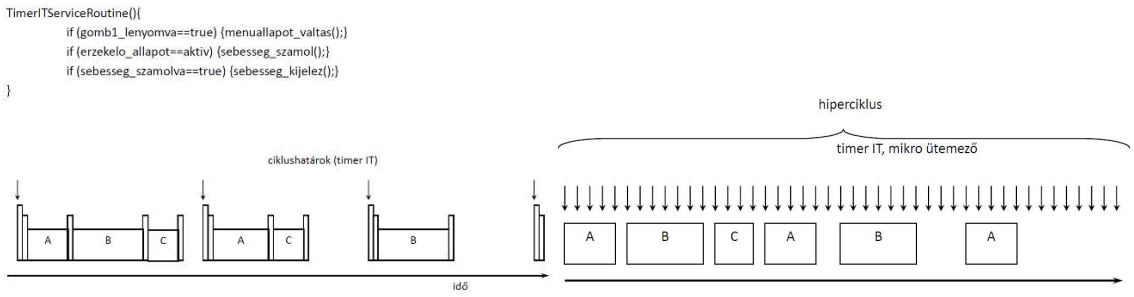
[1] preemptív: egy fontosabb feladat megszakíthatja egy alacsonyabb prioritású feladat végrehajtását

szempontból és kifogásolható, mivel az ütemezés folyamatos számítási kapacitást igényel, így a processzor akkor is fut, amikor egyik feladatnak sincs szüksége kiszolgálásra, ez akkumulátoros üzemű eszközöknél komoly hátrányt jelent. Összességében elmondható, hogy az egyszerű ciklikus megvalósítás csak olyan esetekben alkalmazható, ahol a válaszidő vagy az energiafelhasználás nem kritikus tényező.



3.2.2. Ábra – Ciklikus körforgó architektúra prioritásokkal [9]

Az egyszerű ciklikus szervezés egyik legnagyobb problémája az alacsony szintű energiahatékonyság, azonban ennek javítása általában egyszerűen kivitelezhető idővezérelt architektúra alkalmazásával. A megoldás lényege, hogy a működést megvalósító ciklust csak előre meghatározott időközönként futtatjuk, a fennmaradó időben pedig a mikrokontroller alvó állapotban várakozhat, ahogy ez a 3.2.3. ábrán megfigyelhető. Ebben az esetben a ténylegesen feladatvégrehajtással töltött idő nem változik, viszont csökkenthető az üresjárási állapotokban történő energiafogyasztás. Felmerülhet a kérdés, hogy milyen gyakorisággal futtassuk az ütemezőt ebben az esetben. Ennek megválaszolása igényli az elvégzendő feladat pontos követelményeinek ismeretét, azonban általánosan elmondható, hogy minél gyakrabban végezzük, annál jobban közelíti a hatékonyság az egyszerű körforgós megoldását, viszont túl ritkára sem választható az időzítés, mivel ez a válaszidő növekedését jelenti. További szempont, hogy az időzítés a feladatok végrehajtásától függetlenül történik, tehát a legrosszabb esetben is a feladatoknak bele kell férniük a két ütemezés közti időkeretbe.

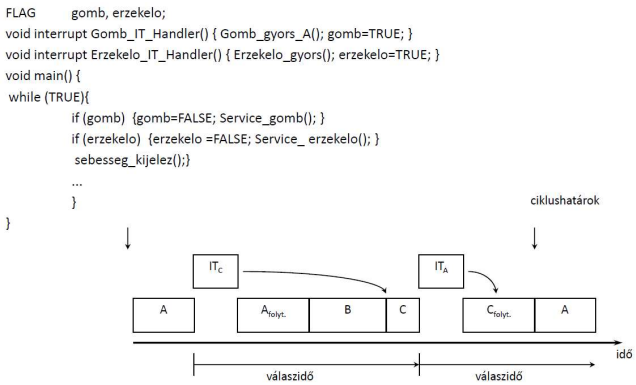


3.2.3. Ábra – Idővezérelt ciklikus (balra) és szigorúan idővezérelt architektúra [9]

Ezt a megoldást általában mintavételező és feldolgozó rendszerekben használják. Az architektúra egyik komplexebb változata a szigorúan idővezérelt körforgó, melynek lényege, hogy a feladatok pontosan meghatározott időpontokban indulnak, az idő múlását egy finom osztású időzítés figyeli, majd a megfelelő pillanatban indítja a végrehajtandó

feladatot. Ezutóbbi megoldás nagyon pontos időzítéseket tesz lehetővé, azonban nehezen bővíthető, mivel egy új feladat hozzáadásakor a teljes ütemezést át kell alakítani, általában valós-idejű rendszerekben használják, mivel alacsony válaszidő és szigorú időzítési korlátok érhetők el vele.

Az eddig ismertetett architektúrák közös negatív tulajdonsága, hogy időkritikus aszinkron események kezelésére kevésbé alkalmasak. Ennek a hiányosságnak a legfőbb oka, hogy valamennyi megoldás nélkülözi a preemptív viselkedést, tehát a feladatok nem képesek egymás futását megszakítani. Minden mikrokontrollerben rendelkezésre állnak úgynevezett megszakítások (interrupt, IT), melyek lényege, hogy valamilyen aszinkron esemény hatására a program futása megszakad és egy előre definiált másik ponton folytatódik. A megszakítást kiváltó esemény kezelése után a program automatikusan visszatérhet a félbehagyott feladat végrehajtásához. Sokszor előfordul, hogy egy-egy aszinkron eseményre nagyon gyorsan kell reagálnunk, azonban az eseményre adott válasz csak egy része időkritikus. Ebben az esetben általános megoldás, hogy az időkritikus reakciót a megszakításkezelő függvényben valósítjuk meg, a többi rész végrehajtását pedig a hagyományos architektúrára bízzuk, általában valamilyen eseményjelző változó (flag) beállításával jelezve annak bekövetkeztét. Ezt a megoldást hívjuk megszakítással kiegészített programszervezésnek, és általában valamennyi architektúrával vegyíthető, ahogy a 3.2.4. Ábra is szemlélteti. Fontos azonban kihangsúlyozni, hogy a megszakítás kontextusában elvégzett részfeladatok nem tarthatnak sokáig, ugyanis a megszakításkezelő függvény futása alatt a program normális futása nem folytatódhat, ezzel jelentősen rontva a rendszer válaszidejét. A megszakítással kiegészített ütemezés legnagyobb előnye, hogy időkritikus reakciók megvalósítására alkalmas, ami rendkívül gyakori igény, ezért elterjedt megoldás. Hátránya viszont, hogy elrontja a program determinisztikusságát, valamint szükségessé teszi a megszakítások és a főprogram által közösen használt változókra a kölcsönös kizárás biztosítását.



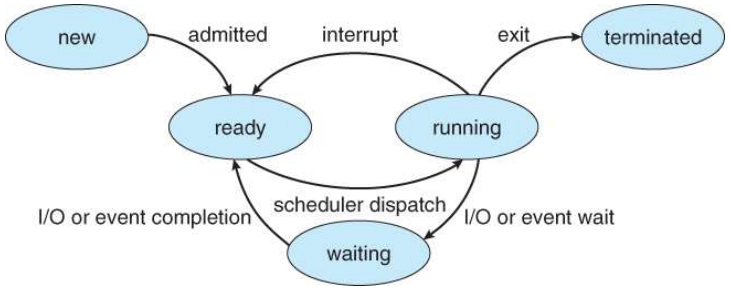
3.2.4. Ábra – Megszakításokkal kiegészített körforgó architektúra [9]

3.2.2. Ütemezett végrehajtás, operációs rendszerek

Az eddigi megoldások során az egyes feladatok végrehajtási sorrendje előre meghatározott, ennek kialakítása pedig a programozó feladata a fejlesztés során. Előfordulhat azonban olyan szituáció, amikor a feladatok végrehajtása kizárólag a külvilág eseményeitől

függ, illetve futási időben is változhat az aktuális igényeknek megfelelően. Hasonló működés megvalósítására ad lehetőséget a függvénysoros ütemezést alkalmazó architektúra, melynek lényege, hogy a végrehajtandó feladatokat egy ütemezési sorban tároljuk. A végrehajtás egy alapértelmezett feladattal kezdődik, majd az igényeknek megfelelően (pl. aszinkron esemény, megszakítás) hatására egy új feladat kerül a végrehajtási sorba. Az ütemező ezt az új feladatot észreveszi, a feladatot kiemeli a sorból és végrehajtja. A feladatok végrehajtása során új feladatok kerülhetnek a sorba, és ezek végrehajtási sorrendje dinamikusan változhat és a rendszer nem tölt időt olyan feladatok futtatásával, amelyekre pillanatnyilag nincs szükség. Mivel az új feladatokat általában megszakításkezelő függvényekben adjuk a végrehajtási sorhoz, a mikrokontroller alvó állapotba tehető amint a sor kiürült, mivel a következő megszakítás általában felébreszti. Természetesen ezt ellenőrizni kell a mikrokontroller adatlapjában, mivel előfordulhat, hogy nem minden energiatakarékos üzemmódban alkalmazható valamennyi megszakítás. A végrehajtási sor a nevével ellentétben nem feltétlenül FIFO[1] működésű, megvalósíthatók prioritásos sorok is.

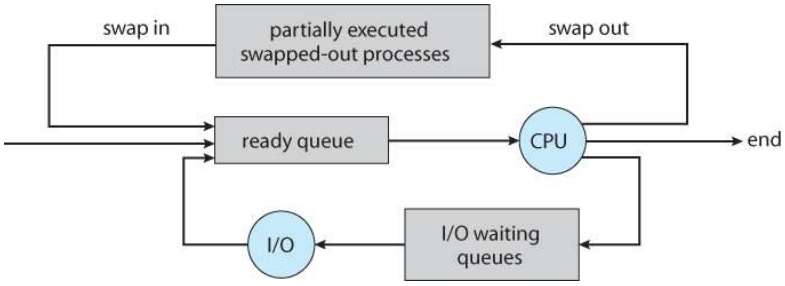
Sok esetben szükségünk lehet az egyes feladatok látszólag párhuzamosan történő végrehajtására is. A korábban ismertetett megoldások mind egyetlen szálon futnak és bár a megszakítások által biztosított preempció lehetővé teszi az időkritikus feladatok megfelelő kezelését, párhuzamosítható feladatvégrehajtásra nem alkalmasak. A függvénysoros ütemezés megvalósításából kiindulva viszont eljuthatunk egy sokoldalú megoldáshoz, az operációs rendszerekhez. A modern operációs rendszerek, bár sok egyéb funkcionalitást is megvalósíthatnak, legfőbb feladatuk a különböző feladatok végrehajtásának menedzselése. A legegyszerűbb esetben egy operációs rendszer csak egy ütemezőt és a feladatok nyilvántartására használt adatszerkezetet tartalmaznak. Attól függően, hogy a különböző feladatok megszakíthatják–e egymás futását, beszélhetünk preemptív és nem-preemptív operációs rendszerekről. Egy operációs rendszerben a feladatok többféle állapotban lehetnek attól függően, hogy készen állnak a futásra vagy sem. A leggyakoribb állapotátmeneteket a 3.2.5. Ábra mutatja. A feladatok általában akkor nincsenek futásra kész állapotban, ha valamilyen esemény bekövetkeztére várnak, ami lehet egy megszakítás, vagy IO[2] művelet végrehajtásának befejezésre. Nem-preemptív esetben a feladatok önként is lemondhatnak a futásjogról, ilyenkor automatikusan visszakerülnek a végrehajtási sorba, preemptív ütemezésnél pedig az operációs rendszer erőszakkal is elveheti a futás jogát, hogy lehetőséget adjon egy másik feladat számára.



3.2.5. Ábra – Feladatok lehetséges állapotátmenetei [23]

[1] FIFO: First In First Out, magyarul: a hamarabb érkezett feladatot hajtjuk végre először
[2] IO: Input-output, magyarul: bemenet-kimeneti művelet (például felhasználói beavatkozás vagy kijelzés)

A futásra kész feladatok egy végrehajtási sorban várakoznak arra, hogy az ütemező biztosítsa számukra a futásjogot. Ezek a sorok megvalósítástól függően lehetnek FIFO-jellegűek vagy prioritásosak, általában utóbbi a jellemző. Preemptív operációs rendszerekben a feladatok látszólagos párhuzamos végrehajtásáért az ütemező felelős. A legtipikusabb megvalósításban az ütemező meghatározott időnként, periodikusan fut egy hardveres időzítő megszakításaként. Ilyenkor az ütemező megszakítja az aktuálisan futó feladat végrehajtását, elmenti a feladat és a processzor belső állapotát, majd kiválasztja a következő feladatot a futásra kész feladatok sorából. Amennyiben csak egyetlen végrehajtó egység áll rendelkezésre (pl. egy magos processzorok), valójában egyszerre csak egy feladat futhat és a párhuzamosság csak látszólagos, viszont többmagos processzorok esetében akár több feladatot is képesek lehetünk valóban párhuzamosan végrehajtani. Az operációs rendszerek ütemezőjének általános felépítését a 3.2.6. Ábra szemlélteti.



3.2.6. Ábra – Operációs rendszerek feladatütemezőjének működése [23]

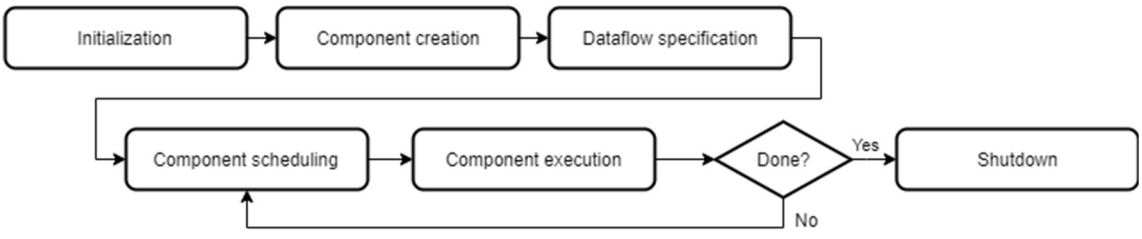
A szakdolgozatom során a már korábban ismertetett ESP-IDF programozási platformot használtam, ami a FreeRTOS operációs rendszeren alapul, ezért ezt az architektúrát alkalmaztam.

3.3. Adatfolyam architektúra

Szoftverfejlesztés során általában az imperatív programozási paradigmát alkalmazzuk, ami azt jelenti, hogy a programot egymás után szekvenciálisan végrehajtandó utasítások sorozataként képzeljük el – nincs ez máshogy a beágyazott szoftverekkel sem. Az imperatív programozási paradigma a programvezérlési út szempontjából közelíti meg a problémát, a hangsúlyt az egyes utasítások végrehajtási sorrendje jelenti. Mivel az IoT alkalmazások központjában többnyire az adatok kezelése található, ezért ebből az irányból is megközelíthetjük a problémáinkat. Ez a megoldás egyfajta deklaratív programozási forma alkalmazását teszi lehetővé, ahol a középpontban nem a vezérlési szerkezetek, hanem a program által feldolgozott adatok és az ezek közötti kapcsolatok, transzformációk, illetve függőségek állnak. A hasonló megközelítést alkalmazó eljárásokat adat-orientált programozásnak, az architektúrákat pedig összefoglaló néven adatfolyam architektúráknak nevezzük. Ez az architektúra megvalósítható hardverben és szoftverben egyaránt, a lényeg a szemléletmódon van, a szakdolgozatom során a szoftveres megközelítést alkalmaztam. Az általam javasolt módszer folyamfeldolgozás és reaktív programozás néven is ismert. A szakdolgozatom során megvalósított adatfolyam architektúra hibrid megközelítést alkalmaz, míg a szoftver logikáját deklaratív módon a komponensek és a köztük

lévő adatkapcsolatok határozzák meg, a komponensek belső működését imperatív leírásmóddal adtam meg. Egy adatfolyam alapú szoftver fejlesztése hasonlít egy szinkron logikai hálózat tervezéséhez, az egyes komponensek ütemezése és frissítése analóg módon megfeleltethető egy logikai rendszer órajelével. A lényeges különbség, hogy míg egy logikai rendszerben igaz-hamis értékekkel dolgozunk a bitek szintjén, addig egy adatfolyam rendszerben komplex értékeket és adatstruktúrákat is felhasználhatunk.

Az adatfolyam architektúra lényege, hogy a program önálló, zárt számítási egységekből épül fel, a logikát pedig az egyes műveletvégző egységek közötti adatkapcsolatok határozzák meg. A műveletvégzők önállósága és zártsága abban nyilvánul meg, hogy jól definiált bemenetekkel és kimenetekkel rendelkeznek, valamint többnyire egyetlen pontos funkcionalitást valósítanak meg. Egymással kizárólag a köztük definiált adatkapcsolatokon keresztül kommunikálhatnak, illetve rendszerint nem módosíthatják a program globális állapotát. Ez a megoldás nem definiálja a programvezérlési utat, kizárólag a program által kezelt adatokat és a rajtuk végzett műveleteket. Mivel a gyakorlatban használt számítógépek, mikrokontrollerek ettől függetlenül a Neumann- vagy Harward-architektúra szerinti felépítést követik, a vezérlési struktúra megvalósítása elengedhetetlen szükséglet. A szoftveres adatfolyam alapú megoldások ezt a problémát általában valamilyen ütemező segítségével oldják meg, amely a kezdeti inicializáció, a műveletvégző komponensek létrehozása és az adatkapcsolatok definiálása után folyamatosan lépteti az egyes komponenseket, ahogyan a 3.3.1. ábrán megfigyelhetjük. Egy ilyen frissítés alkalmával a komponensek megvizsgálják, hogy a működésükhöz szükséges feltételek teljesülnek –e (pl. bemeneti adatok rendelkezésre állnak), majd a bemenetek olvasása után elvégzik a funkciójukat és az eredményeket a kimenetükre írják. A komponensek közötti adatkapcsolatokat alacsony szinten általában egy FIFO jellegű tároló valósítja meg, melynek mindkét oldalán egy-egy komponens kimeneti, vagy bemeneti portja található. Az architektúra legnagyobb előnyei, hogy deklaratív, könnyen értelmezhető leírást tesz lehetővé és implicit módon párhuzamosítható. Utóbbi rendkívül hasznos tulajdonság, ugyanis az explicit párhuzamos programozás különös odafigyelést igényel és egyáltalán nem intuitív. Az egyes komponensek önmagukban szekvenciálisan működnek, azonban bármely komponens futtatható párhuzamosan, a köztük lévő szinkronizációt a bemeneti adatok elérhetősége automatikusan garantálja, így a programozónak nem szükséges a kölcsönös kizárás megvalósítására ügyelnie és elkerülheti a párhuzamos programozás egyéb bonyolult aspektusait is. A feladatok párhuzamosítását az ütemező automatikusan képes elvégezni a rendszerben található erőforrások dinamikus felhasználásával, akár heterogén hardveres architektúrák esetén is.



3.3.1. Ábra – Adatfolyam alapú szoftver általános vezérlési gráfja

A szakdolgozatom során egy egyszerű adatfolyamok reprezentálására alkalmas keretrendszert implementáltam a fent említett alapelveknek megfelelően. Első lépésként az egyes komponensek között továbbításra kerülő üzenetek formátumát kellett meghatároznom. A formátummal szembeni egyik legnagyobb elvárásom az volt, hogy lehetőleg minél szélesebb körű adattípusok továbbítására legyen alkalmas, illetve ezeket az adatokat rendszerezett formában, intuitív szintaktikával lehessen elérni. Az implementációhoz az ötletet két forrásból [24] [25] szereztem, mindkét megoldás fa-struktúrába rendezi az üzeneteket, ezáltal organizált, hierarchikus formában kezelhetők az üzenetekben tárolt adatok. A tetszőleges adattípusok továbbításához az üzenetek egyes elemei (*Node*) képesek bármilyen típus tárolására, majd azok típushelyes módon történő kiolvasására a C++ nyelvből ismert *std::any* mintájára. Ennek megvalósítása manuálisan történt a C++11 szabvány eszköztárának felhasználásával (az *std::any* csak a C++17 szabványban jelent meg, amit sok mikrokontrolleres fordító egyelőre nem támogat), és erőteljes sablon metaprogramozást igényelt. A típushelyesség garanciáját a virtuális függvények megvalósításánál megszokott virtuális táblák segítségével oldottam meg, minden alkalmazott típushoz egyetlen ilyen (a felhasználó elől rejtett) tábla tartozik. A *Node*-okban tárolt adattípustól függően mindig az adott típushoz tartozó tábla memóriacíme is eltárolásra kerül, melyet értékadáskor frissítünk, olvasáskor pedig a típuskonverzió során ellenőrzünk. A virtuális tábla tartalma néhány függvénypointer, amely a típushelyes destruktorok automatikus meghívásáért, az adatok másolásáért és (amennyiben támogatott) a tárolt objektum szabványos kimenetre történő írásáért felelősek. Annak érdekében, hogy a lehető legkevesebb többletmemóriát kelljen a fenti funkciókhoz felhasználni, az implementáció része a „kis buffer optimalizáció”: A *Node* objektum alapesetben dinamikus memóriát foglal a tárolandó adatoknak, de ha az adatok mérete kisebb, mint a hivatkozáshoz használt pointer mérete, akkor a pointer helyett közvetlen az adatot tárolja.

```
// Creating Node objects from various datatypes
Node int_node = 42;
Node string_node = "Hello";

// Copying and assigning nodes with new values
Node int_node2 = int_node;
string_node = "World";

// Printing nodes to the standard output
std::cout << int_node << " " << string_node;
```

A *Node* objektumok egymásra hivatkozva építek fel az említett fa-szerkezet, az üzeneteket egy ilyen fa gyökere reprezentálja. Az üzenetstruktúra iterátorok segítségével (preorder jellegű, depth-first módon) is egyszerűen bejárható, ehhez megfigyeltem, hogy a *Node* objektumok fája leképezhető bináris faként a memóriában úgy, hogyha a minden objektum esetében a gyerekei tőle jobbra, a testvérei pedig tőle balra helyezkednek el. Az intuitív szintaktika érdekében az üzeneteken belüli hierarchiát a tömbök indexeléséhez hasonló módon lehet olvasni, nem létező bejegyzések írásakor azok automatikusan létrejönnek (kivéve csak-olvasható üzenetek esetében, ilyenkor kivétel keletkezik). Az üzenetek fa-struktúrája egyaránt kezelhető az egyes *Node*-ok nevével vagy indexével hivatkozva, utóbbi különösen a tömbökhöz hasonló kezelést teszi kényelmesebbé. A következő kódrészlet megmutatja az üzenetek kezelésének szintaktikáját, egy olyan adatfolyam

komponensből származik, amely a BME280 szenzor által mért értékeket hálózaton történő továbbításra alkalmas üzenetformátumba alakítja (erről részletesen a 3.5.4. fejezetben lesz szó):

```
// Node to read messages with
Node message;

// Reading message from the input port
ports["in"].receive(message);

// Debug printing the message structure
for(auto it = message.begin(); it != message.end(); it++) {
    std::cout << std::string(it.level(), ' ') << it->name() << ": " << *it << std::endl;
}

// The printed message looks like this, all values have the type of double,
// except for the root Node which does not hold a value but acts as a parent
// root: null
//     temperature: 24.25621
//     pressure: 101523.21
//     humidity: 24.11245

// Extracting temperature, pressure and humidity data from the message
// Casting to the wrong datatype throws an std::bad_cast exception (if enabled)
double temperature = (double) message["temperature"];
double pressure    = (double) message["pressure"];
double humidity    = (double) message["humidity"];

// Creating output message, as the assigned values have the type double,
// the reader will also have to cast them to double
message.clear();
message["update"][0] = temperature;
message["update"][1] = pressure;
message["update"][2] = humidity;

// Debug printing the modified message structure
for(auto it = message.begin(); it != message.end(); it++) {
    std::cout << std::string(it.level(), ' ') << it->name() << ": " << *it << std::endl;
}

// The printed message looks like this:
// root: null
//     update: null
//         [0]: 24.25621
//         [1]: 101523.21
//         [2]: 24.11245

// Sending output message
ports["out"].send(message);
```

Az adatfolyam komponensek egy közös absztrakt alaposztályból (*Component*) történő örökléssel valósíthatók meg. Ez az osztály biztosítja a kommunikációs portok létrehozását és elérését a leszármazottak számára és definiálja a minden komponens által megvalósítandó interfész függvényeket. Az osztály *process(void)* metódusának felüldefiniálásával valósítható meg az adott komponens konkrét funkciója.

A portokhoz történő hozzáférés az alaposztályban definiált tárolón keresztül történhet, melyet minden leszármazott megörököl. A portok elérésének szintaktikája a tömbök indexeléséhez hasonló, a port nevének megadásával történhet. A lekérdezés eredménye a komponensen belül közvetlenül a portot reprezentáló objektum, azon kívül pedig egy segédobjektum, amelyen keresztül összekapcsolhatók az egyes portok. Az összekapcsolás szintaktikája kényelmes használatot tesz lehetővé, az összeköttetések láncba fűzhetők, a portlekérdezés eredménye ismételten indexelhető, végeredményében az egymás mellett álló portok kerülnek összekötésre.

```
// The following two methods of component connection are equivalent:

// Chaining connections
wifiConnection["out"] >> forecastReader["in"]["out"] >> display["in"];

// Multiple single connections
wifiConnection ["out"] >> forecastReader["in"];
forecastReader["out"]  >> display["in"];
```

A portok közötti adatkapcsolatokat a mikrokontrolleren futó operációs rendszer (FreeRTOS) üzenetsorainak segítségével valósítottam meg, ezzel biztosítva a szálbiztos kommunikációt az egyes komponensek között. Mivel ezek az üzenetsorok a C-nyelvű implementáció miatt csak triviális típusok kezelésére alkalmasak (például nem hívnak másoló konstruktorokat és destruktorokat), ezért az üzeneteket dinamikusan foglalt memóriában helyeztem el, és a memóriacímük segítségével kerülnek átadásra. Az adatfolyam keretrendszer ebben az első, viszonylag kezdetleges megoldásban minden komponens számára saját szálat hoz létre az operációs rendszerben, így a komponensek ütemezésének feladatát az operációs rendszer belső ütemezője biztosítja. A későbbiekben az implementáció számos ponton bővíthető és fejleszthető, például saját ütemező algoritmussal, vagy kevesebb dinamikus memóriafoglalást igénylő megvalósítással. Megfontolandó lehet a jelenlegi üzenet reprezentáció típusellenőrzésének szigorítása is, mivel pillanatnyilag a hibás típuskonverziókból eredő hibák csak futásidőben derülnek ki. További fejlesztési lehetőség a vizuális programozás támogatása, ugyanis az architektúra egyszerűen lehetővé teszi az előre elkészített komponensekből grafikus szoftver segítségével történő adatfolyam hálózat megtervezését és az abból történő kódgenerálást. A módszer akár lehetővé teheti az adatfolyam futási időben történő átkonfigurálását is, ezzel gyorsítva a fejlesztési folyamatot is.

3.4. Beágyazott eszközillesztők használata

Beágyazott rendszereink fejlesztése során gyakorlatilag mindig szükségünk van a rendszerünkben található egyes eszközök, perifériák kezelésére. Ennek tekintetében két nagy kategóriát különböztethetünk meg, a mikrokontrolleren belül- és az azon kívül található eszközöket. Mivel a mikrokontrolleren belüli eszközök kezelését a legtöbb esetben a választott keretrendszer már megvalósítja, ezért ebben a fejezetben utóbbi kategóriával foglalkozom. Ilyen eszközök lehetnek a különféle külső szenzorok, beavatkozó egységek (pl. motorok) és kommunikációs perifériák. Ezeknek a perifériáknak a kezelése bár elvégezhető közvetlenül, alacsony szinten az alkalmazáskódban is, de a legtöbb esetben külön szoftveregységekbe, ún. beágyazott eszközillesztőkbe szervezzük. Egy eszközillesztő szoftverkomponens komplexitása az alkalmazás számára legszükségesebb funkcionalitás megvalósításától a teljeskörű vezérlésig terjedhet igény szerint. A továbbiakban bemutatom a beágyazott eszközillesztő programok fejlesztésének legfontosabb lépéseit, általános tudnivalóit, majd a szakdolgozatom során felhasznált integrált meteorológiai szenzoregység (BME280) példáján keresztül illusztrálom a leírtakat. Végezetül ismertetem, hogy hogyan lehet egyszerűen az előző fejezetben bemutatott adatfolyam architektúrába illeszteni egy imperatív módon elkészített eszközillesztőt.

Bár a mérnöki gyakorlatban használt külső perifériák, szenzorok rengeteg különböző kivitelben elérhetőek és kezelésük is sokféle lehet, általában két nagy kategóriába sorolhatók vezérlés szempontjából, megkülönböztethetünk regiszterszervezésű és parancs-alapú kezelési móddal rendelkező eszközöket. A két kategória között nem feltétlenül éles a határvonal, léteznek vegyes megoldások is. A regiszterszervezésű eszközök közös tulajdonsága, hogy saját regiszterkészlettel rendelkeznek, melyeken keresztül vezérelhetjük az eszközök működését. A periféria regiszterei sokféle célt szolgálhatnak, általában beállítások tárolására és szenzorok esetében a mérési eredmények tárolására használják őket. Ezek a regiszterek általában egyaránt írhatók és olvashatók (pl. beállítások), de léteznek csak olvasható regiszterek is (pl. mérési eredmények). Az ilyen eszközök kezelése többnyire ezeknek a regisztereknek az írásából és olvasásából áll. A másik jelentős kategória a parancsokkal vezérelhető eszközök, melyek vezérlése a kommunikációs felületen kiadott parancsokkal történik. Utóbbi megoldás a korábban említett módon kiegészülhet regiszterszervezéssel is, ebben az esetben általában külön parancs szolgál a regiszterek írására és olvasására. A szakdolgozatom során mindkét módszert alkalmaznom kellett, a BME280 szenzormodul regiszterszervezésű, míg az OLED kijelző meghajtója (SSD1306) parancs-alapú vezérlést támogat. Az eszközillesztő szoftver fejlesztésének első lépése a periféria adatlapjának tanulmányozása után a regisztercímek (és bitmaszkok) vagy parancsok kigyűjtése. Ezt C-szerű környezetben enumerációkkal vagy makródefiníciókkal szokás megoldani. Általános szabály, hogy a definiált kifejezéseket célszerű az eszköz típusának megfelelő prefix-el ellátni. A következő kódrészlet a BME280 szenzor eszközillesztőjének fejlécéből származik, és bemutatja a regisztercímek és konstansok definiálásának egy lehetséges módját C-nyelvű környezetben.

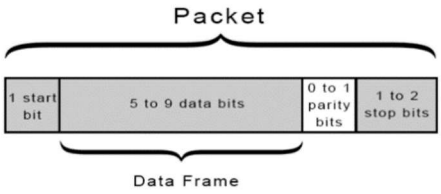
```
/** @brief Defines the default I2C address. */
#define BME280_I2C_ADDR (uint8_t)(0x76)

/** @brief Address of HUM_LSB register. */
#define BME280_HUM_LSB (uint8_t)(0xFE)

/** @brief Address of HUM_MSB register. */
#define BME280_HUM_MSB (uint8_t)(0xFD)
```

A külső perifériák használata esetében mindig szükséges valamilyen szabványos kommunikációs felületet alkalmazni. A kommunikáció történhet soros- vagy párhuzamos formában is, többnyire a soros protokollok az elterjedtebbek az alacsony vezetékigényük miatt. A gyakorlatban általában a következő soros kommunikációs protokollokat alkalmazzák: UART, I2C, SPI

UART/USART[1]: Soros kommunikációs protokoll, általában 2 (aszinkron) vagy 3 (szinkron) vezetéken keresztül. Pont-pont kapcsolatot valósít meg, akár full-duplex (egyszerre kétirányú) kommunikációra is alkalmas. Két adatvonallal rendelkezik, ezek a TX (adott eszköz számára kimenet) és az RX (adott eszköz számára bemenet). Az adatvonalakat a kommunikáló felek között fordítva kell bekötni, így az egyik fél kimenete a másik fél bemenetéhez csatlakozik. Szinkron kommunikáció esetében a két adatvonal kiegészül egy harmadik órajel-vezetékkel is, melyet a két fél közötti kommunikáció szinkronizálására használnak. A kommunikáció a 3.4.1. ábrán szereplő, meghatározott formátumú

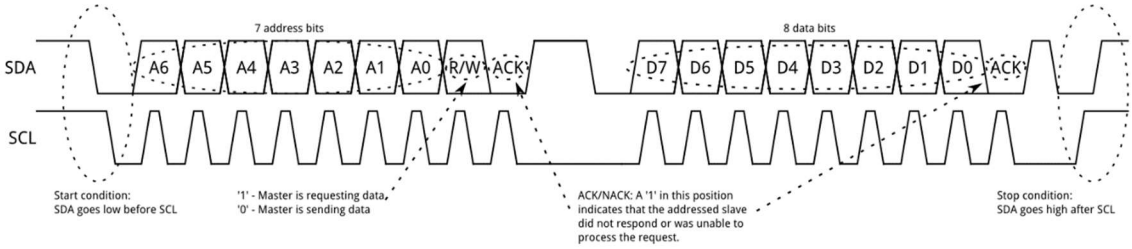
[1] USART: Universal Synchronous/Asynchrounous Receiver-Transmitter, magyarul: univerzális szinkron/aszinkron adó-vevő

adatkeretek küldésével történik, a leggyakrabban használt formátum a 8N1 (8 adatbit keretenként, paritásbitet nem használják, 1 STOP bittel zárják a keretet).



3.4.1. Ábra – UART adatkeret felépítése [26]

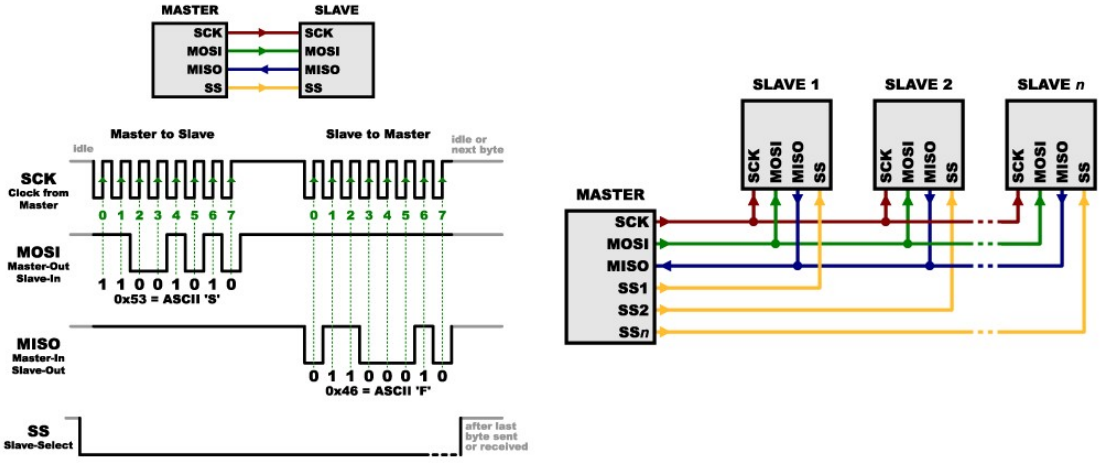
I2C[1]: Soros kommunikációs protokoll, két vezetéket igényel. Fél-duplex kapcsolatot valósít meg (egyszerre csak egy irányban), azonban busz jellege miatt kettőnél több eszközt is csatlakoztathatunk. Architektúráját tekintve Master-Slave alapú. A két vezeték az SCL (órajel, a Master generálja) és az SDA (adatvezeték, a kommunikáció irányától függően a Master vagy a Slave hajtja meg). Nyugalmi állapotban mindkét vezetéket logikai magas szinten tartják egy-egy felhúzó ellenállás segítségével. Az adatcsere mindig bájtonként történik, a kommunikációt a Master kezdeményezi. Az adatcsere egy ún. START-feltétellel kezdődik, ezt követi a kívánt eszköz megcímzése (7/10 bit) és az adatcsere irányának jelzése (írás/olvasás, a Master szemszögéből nézve). A sikeres címzést egy jóváhagyó (ACK, acknowledge) bit jelzi, melyet a kiadott cím alapján magára ismerő Slave generál. A címzési fázis után következhet az adatbájtok átvitele, szintén jóváhagyással kiegészítve. A kommunikáció végét az ún. STOP-feltétel jelzi. A kommunikáció idődiagramja a 3.4.2. ábrán szerepel.



3.4.2. Ábra – I2C kommunikációs protokoll diagramja [27]

SPI[2]: Soros kommunikációs protokoll, legalább négy vezetéket igényel, full-duplex jellegű adatcserére alkalmas, busz jellege miatt több eszközt is csatlakoztathatunk hozzá, architektúrája Master-Slave alapú. A kommunikáció szinkron jellegű, az órajelet a Master generálja az SCK vonalon. Az adatcsere a MISO (Master input – Slave Output) és MOSI (Master output – Slave Input) vezetékeken történik bitenként. A különböző Slave eszközök engedélyezését a CS (Chip Select, vagy SS – Slave Select) vonal végzi, egyszerre mindig csak egy Slave lehet aktív, így minden eszköz saját CS bemenettel rendelkezik. A kommunikáció idődiagramját és a busz hardveres felépítését a 3.4.3. Ábra mutatja.

[1] I2C: Inter-integrated circuit, magyarul: integrált áramkörök közötti
[2] SPI: Serial peripherial interface, magyarul: soros periféria interfész



3.4.3. Ábra – SPI protokoll idődiagramja és a busz felépítése [28]

Előfordulhat, hogy egy külső periféria több kommunikációs interfészt is támogat. A beágyazott eszközillesztő szoftverek elkészítésekor célszerű és általánosan alkalmazott módszer, hogy a kommunikációs interfészt valamilyen absztrakció mögé rejtve kívülről kell biztosítania a felhasználónak. Ennek a megoldásnak a legfőbb előnye, hogy lehetővé teszi az eszközillesztő kódjának hordozhatóságát különböző mikrokontrollerek és keretrendszerek között, melyek esetenként rendkívül különböző absztrakciókat kínálnak a kommunikációs perifériákhoz. C-nyelvi környezetben ez leggyakrabban valamilyen függvénypointer megadását jelenti az illesztő inicializálásakor, többnyire külön függvényeket kell megadni az íráshoz és olvasáshoz. Az illesztőprogram belsejében feltételezzük, hogy a felhasználó által megadott függvények megfelelően elvégzik az általunk előírt kommunikációs funkciójukat. A következő kódrészlet egy I2C kommunikációs interfész lehetséges absztrakcióját mutatja, a felhasználó definiálja az írást és olvasást végző függvényeket az alábbi fejlécekkel majd ezekkel inicializálja az eszközillesztőt:

```
/**
 * @brief Function pointer signature for reading from the I2C bus.
 * @param address  [in] The 7-bit I2C device address (right-justified).
 * @param data     [out] Pointer to the array where the data is read.
 * @param length   [in] The number of bytes to read from the I2C bus.
 */
typedef void (*I2C_read)(uint8_t address, uint8_t data[], uint8_t length);

/**
 * Function pointer signature for writing to the I2C bus.
 * @param address [in] The 7-bit I2C device address (right-justified).
 * @param data    [in] Pointer to the data to write to the I2C bus.
 * @param length  [in] The number of bytes to write.
 */
typedef void (*I2C_write)(uint8_t address, uint8_t data[], uint8_t length);

/**
 * @brief Initializes the driver with the I2C communication functions.
 * @param readFunction  [in] Pointer to the reader function.
 * @param writeFunction [in] Pointer to the writer function.
 */
void MyDriver_Initialize(I2C_read readFunction, I2C_write writeFunction);
```

Természetesen az adott eszköz által alkalmazott protokoll függvényében elképzelhető, hogy a fenti példánál kevésbé általános módon definiáljuk a felhasználó által biztosítandó függvényekkel szemben támasztott elvárásokat. Például ha az adott eszköz regiszterszervezésű, a függvények paramétereként megadhatjuk az írni/olvasni kívánt regiszter címét

is, ilyen esetekben viszont mindenképpen fontos az elvárások pontos és részletes dokumentációja. Elképzelhető a fenti példától eltérő típusú megvalósítás is, a szakdolgozatom során a kommunikációs absztrakciókhoz objektum-orientált megközelítést alkalmaztam.

A kommunikációs protokollokat osztályokban valósítottam meg, melyek egy közös ősosztályból öröklődnek és lekérdezhető tőlük a konkrét megvalósított protokoll. Az eszközillesztők egy alaposztályra mutató pointer átvétele után lekérdezik a megvalósított protokoll típusát és annak megfelelően használják az interfészt, így például a BME280 illesztőprogramja akár I2C vagy SPI protokoll használatával is képes működni. Az I2C protokollt az *I2C_Master* osztály valósítja meg, mely a kommunikáció paramétereit a konstruktorában veszi át, az egyes elvégzendő műveleteket egy függvénysorban tárolja, majd kérésre az eddig hozzáadott műveleteket sorrendben végrehajtja, emellett rendelkezik kölcsönös kizárásra megvalósítására alkalmas függvényekkel is, így párhuzamosan több eszköz is használhatja ugyanazt az interfészt.

A következő kódrészlet a BME280 regiszter-olvasását mutatja a fenti objektum-orientált absztrakció felhasználásával.

```
void BME280::register_read(uint8_t registerAddress, uint8_t buffer[], uint8_t byteCount) {

    // Checking the type of the communication interface
    if(m_interface->getType() == DriverInterface::Type::I2C) {

        // Locking the interface
        ((I2C_Master*)m_interface)->lock();

        // Start condition
        ((I2C_Master*)m_interface)->queue_start();

        // I2C bus address
        ((I2C_Master*)m_interface)->queue_write_byte((BME280_I2C_ADDR << 1) | I2C_MASTER_WRITE, true);

        // Device register address
        ((I2C_Master*)m_interface)->queue_write_byte(registerAddress, true);

        // Restart condition
        ((I2C_Master*)m_interface)->queue_start();

        // I2C bus address
        ((I2C_Master*)m_interface)->queue_write_byte((BME280_I2C_ADDR << 1) | I2C_MASTER_READ, true);

        // Reading data
        ((I2C_Master*)m_interface)->queue_read(buffer, byteCount, I2C_Master::acknowledge_t::nack_last);

        // Stop condition
        ((I2C_Master*)m_interface)->queue_stop();

        // Executing command sequence
        ((I2C_Master*)m_interface)->execute();

        // Unlocking the interface
        ((I2C_Master*)m_interface)->unlock();
    }

    ...
```

A megfelelő kommunikációs absztrakció definiálása után, megvalósíthatjuk az eszközillesztő által támogatott valamennyi funkcionalitást a felhasználó által biztosított lehetőségek használatával. Fontos megemlíteni, hogy sok beágyazott eszközhöz nem szükséges magunknak megírni az illesztőprogramot, léteznek mások által előre elkészített szoftvercsomagok is. Ezek vagy közvetlenül beilleszthetők az alkalmazásunkba (ha ugyanahhoz a keretrendszerhez készültek, mint amit mi használunk, leggyakrabban Arduino esetén fordul elő), vagy az említett absztrakció megvalósítását követően használhatjuk őket. A

legtöbb esetben ezzel befejeződik az eszközillesztő fejlesztése, azonban a szakdolgozatom során szükség volt az elkészült illesztőprogram adatfolyam-architektúrába illesztésére is. Ennek érdekében egy egyszerű adatfolyam-komponenst hoztam létre, amely magában foglalja a szenzor illesztőprogramját, az illesztőt megvalósító osztályból történő öröklés által. A *DF_BME280* komponens induláskor az INTERFACE bemeneti portján blokkoló módon beolvassa a kommunikációs protokollt megvalósító objektum címét, majd ezzel inicializálja az eszközillesztőt. Ezek után folyamatosan várakozik az IN bemeneti portján érkező üzenetekre. Egy ilyen üzenet érkezésekor kezdeményezi egy mérés elvégzését, majd felfüggeszti a futást a mérés időtartamára. A mérés időtartamának letelte után a komponenst futtató folyamat felébred, és az OUT kimeneti portján elküldi a mérési eredményeket. Az említett működést az alábbi kódrészlet valósítja meg:

```
void DF_BME280::process()
{
    // Node object to read messages
    Node message;

    // Reading the INTERFACE port for driver interface pointer
    m_ports["interface"].receive(message);

    // Setting up the driver interface for the sensor
    setDriverInterface((DriverInterface*) message);

    // Initializing the sensor
    initialize();

    // Reading messages from the IN port
    while(true) {

        // Receiving message from the IN port
        m_ports["in"].receive(message);

        // Starting the measurements
        set_mode(BME280::Mode::Forced);

        // Waiting for the measurements to finish
        vTaskDelay((get_measurement_delay_ms()) / portTICK_RATE_MS);

        // Creating output message
        message.clear();
        message["temperature"] = (double) get_temperature();
        message["pressure"]    = (double) get_pressure();
        message["humidity"]    = (double) get_humidity();

        // Sending the measurement data to the output port
        m_ports["out"].send(message);
    }
}
```

Az illesztőprogram I2C interfész használatával kommunikál a szenzorral, ahhoz, hogy ezt az adatfolyam-alapú leírásba tudjam illeszteni, létrehoztam a *DF_I2C_Master* komponenst, amely az *I2C_Master* osztályból történő örökléssel tartalmazza a szükséges funkcionalitást, egyetlen INTERFACE kimeneti porttal rendelkezik, melyen induláskor elküldi a saját memóriacímét, majd felfüggeszti a futást. Ezt a kimeneti portot csatlakoztattam a szenzor és a kijelző modul INTERFACE bemeneteihez, így azok elérhetik a számukra szükséges funkcionalitást.

A szakdolgozatom keretein belül implementálásra került továbbá a SSD1306 meghajtóval működő OLED kijelző illesztőprogramja is. Az illesztőprogram megvalósításának pontos részletei túlmutatnak a dolgozat keretein, az alábbiakban röviden összefoglalom a működés legfontosabb tulajdonságait. Az illesztőprogram csak a legalapvetőbb funkcionalitást valósítja meg, kizárólag inicializálásra, a kijelző tartalmának törlésére és pixelek

adott koordinátákra történő kirajzolására alkalmas. Kommunikációra a korábban ismertetett I2C absztrakciót alkalmazza. A grafikus funkcionalitást egy absztrakt alaposztály, a *GraphicsDevice* tartalmazza, amely grafikus primitívek (vonalak, alakzatok) és szövegek, valamint szimbólumok kijelzésére képes. Az említett funkcionalitás végrehajtására a leszármazott osztályok által definiált pixelrajzolást megvalósító függvényeit alkalmazza. Az *SSD1306* osztály leszármazik a *GraphicsDevice* osztályból, így a pixelek kirajzolására alkalmas függvények megvalósítása mellett, ezzel párhuzamosan megkapja az alaposztály komplexebb funkcióit is. A szimbólumok, szövegek kirajzolásához egy saját készítésű szimbólumleírást alkalmaztam. A szimbólumokat egy JSON formátumú fájlban szövegesen definiáljuk pixelenként a kiegészítő metainformációkkal (karakterkészlet mérete, szimbólumok neve, stb.) együtt. Ezt a leírást egy asztali környezetben futó C++ alkalmazás értelmezi és C-nyelvű kódot generál belőle, amely két tömböt tartalmaz. Az egyik a szimbólumok metainformációt definiálja, a másik pedig egy bináris adathalmaz, amelyben a szimbólumok minden pixeléhez egy bitet rendelünk. Az így létrehozott bináris adatsorból aztán a *GraphicsDevice* osztály a metainformációk alapján képes kirajzolni a kívánt szimbólumot. A megvalósított eszközben több grafikus szimbólum mellett két teljes ASCII karakterkészletet definiáltam, egyiket 6x6, a másikat pedig 8x8 pixel felbontással. A szimbólumkódoláshoz használt fájl formátumát a 3.4.4. Ábra szemlélteti.

3.4.4. Ábra – Szöveges szimbólumkódolás JSON fájlban

3.5. Hálózati kapcsolatok

IoT rendszerek fejlesztése során az egyik kulcsfontosságú tényező a megfelelő kommunikációs felületek kiválasztása, melyek segítségével az eszközünk képes adatcserét bonyolítani a környezetével, más eszközökkel és az internet segítségével. A gyakorlatban rengeteg különböző kommunikációs lehetőség közül választhatunk, melyek a kommunikáció más-más rétegeiben helyezkednek el. A továbbiakban a vezetéknélküli alkalmazásokra koncentrálva – a rétegzett modellt valamennyire leegyszerűsítve – bemutatom az

egyes kommunikációs rétegekben alkalmazható megoldásokat, illetve azok előnyeit és hátrányait. Végül bemutatom a szakdolgozatom során felhasznált hálózati kommunikációs protokollok alkalmazását és kitérek az elosztott hálózati számítási módszerekre is.

3.5.1. Fizikai réteg

A hálózati kommunikációs rétegek közül a legalacsonyabb szinten található a fizikai réteg. Ebben a rétegben valósul meg a tényleges adatátvitel, valamilyen fizikai médiumon keresztül. A közeg, melyben az információ terjed vezetékes alkalmazásokban lehet például egy telefonvonal, illetve vezetéknélküli esetben vákuum vagy levegő. Vezetéknélküli alkalmazásokban szinte kizárólagosan elektromágneses hullámok segítségével történik az eszközök közötti kommunikáció. A legelterjedtebb kis hatótávolságú protokollok a WiFi és a Bluetooth, valamint létezik néhány IoT eszközökre specializált megoldás is, mint például a Zigbee vagy a Z-Wave.

Az egyik legismertebb, hétköznapi életben is alkalmazott kommunikációs protokoll a WiFi. Általában otthoni vezetéknélküli internetes hálózatokban használják, működését tekintve elektromágneses hullámokon alapul és többnyire a mikrohullámú, 2.4 GHz-es sávban helyezkedik el. Pontosabb megfogalmazásban az eredeti szabvány neve az IEEE[1] 802.11, de léteznek későbbi szabványok is, melyek között a legjelentősebb különbségek az adatátviteli sebességben, az alkalmazott frekvenciasávban és a maximális hatótávolságban keresendők. Legnagyobb előnyei a széleskörű elterjedtség, az egyszerű használat és az alacsony energiafogyasztás, hátránya viszont, hogy alacsony fogyasztása ellenére a folyamatos energiafelhasználás miatt nem minden esetben alkalmazható és hatótávolsága is erősen limitált, beltérben mindössze kb. 20 méter, amit a különböző tereptárgyak jelentősen csökkenthetnek.

A másik szintén gyakran alkalmazott vezetéknélküli fizikai protokoll a Bluetooth. Szintén rövid hatótávolságú alkalmazásokhoz használható. Manapság gyakorlatilag minden okostelefon és laptop képes Bluetooth kommunikációra, így könnyen elérhető. Működése a mikrohullámú tartományban található elektromágneses hullámokon alapul, és szintén a 2.4 GHz-es sávban helyezkedik el. Legnagyobb előnye az alacsony energiafogyasztás, létezik kifejezetten erre a célra fejlesztett változata a Bluetooth Low Energy (BLE), valamint a nagysebességű adatátvitel. Hátránya szintén az alacsony hatótávolság (általános esetben kb. 10 méter), azonban a WiFi-nél nagyobb távolságok esetén is használható az energiafogyasztás megnövekedésének árán.

Kevésbé közismert protokoll a Zigbee [33], melyet kifejezetten IoT alkalmazásokban használnak előszeretettel. Fizikailag szintén a 2.4 GHz-es frekvenciasávban foglal helyet. Működtetése rendkívül alacsony energiafelhasználást igényel, ezért ideális megoldást jelent kis kapacitású akkumulátoros eszközök esetén. Hátránya, hogy az elérhető maximális adatátviteli sebesség erősen korlátozott, mindössze kb. 250 kbit/s, azonban ha nincs szükség nagy adatforgalomra, ez is megfelelő lehet. Felhasználása többnyire például okosotthonokban található szenzorok és egyszerűbb beavatkozó egységekben jellemző, ahol

[1] IEEE: Institute of Electrical and Electronics Engineers

az alacsony energiafelhasználás hosszú önálló üzemidőt tesz lehetővé és az alkalmazás nem igényel szélessávú kapcsolatot. Hatótávolsága kb. 20 méter.

A másik gyakori IoT protokoll a fizikai rétegben a Z-Wave [34], a Zigbee-nél olcsóbb és egyszerűbb megoldás. Ugyancsak rádióhullámokkal működik, azonban a korábbiakkal ellentétben a 800-900 MHz-es sávban helyezkedik el, így kisebb valószínűséggel lesznek interferencia problémái, mint az előző protokolloknak, mivel ebben a sávban többnyire kevesebb eszköz működik. Míg az eddig felsorolt protokollok mind pont-pont kapcsolatot képeznek, vagy egy központi egységhez csatlakoznak, a Z-Wave eszközök ún. mesh-hálózatot hoznak létre. Ennek lényege, hogy nem szükséges minden eszköznek egy vezérlőegységhez csatlakozni, az eszközök egymás között tetszőleges kapcsolatrendszert hozhatnak létre, amely topológiailag egy fa-struktúrát eredményez, ezzel jelentősen növelve az hálózat lehetséges hatótávolságát. Legnagyobb előnyei az alacsony energiafogyasztás, és viszonylag nagy hatótávolság (akár 100 méter), valamint a rendkívül jó kompatibilitás. Hátrányai az alacsony adatátviteli sebesség (kb. 100 kbit/s) és a csatlakoztatható eszközök limitált mennyisége (232 darab).

A felsorolt, viszonylag rövid hatótávolságú protokollokon kívül léteznek még egyéb megoldások is, melyek jóval nagyobb távolságokban is működőképesek, mint például a 4G, vagy az 5G és a Narrowband IoT, azonban ezek tárgyalása túlmutat a szakdolgozatom keretein.

3.5.2. Szállítási réteg

Az ISO/OSI modellben a szállítási réteg felelős a kommunikáló alkalmazások közötti adatkapcsolat biztosításáért. Ebben a rétegben általában alacsonyszintű kommunikációs funkciókat szoktunk megvalósítani, többnyire kizárólag olyankor, amikor saját alkalmazásszintű protokollt használunk. A teljesség kedvéért a továbbiakban röviden összefoglalom a legfontosabb tudnivalókat. A két legjelentősebb szállítási rétegbeli protokoll a TCP és az UDP.

A TCP[1] egy kapcsolat-alapú átviteli protokoll, melyet adatok folyamszerű továbbítására használnak. Biztosítja a kommunikáló felek között az üzenetek garantált és az elküldés sorrendjében történő megérkezését, forgalomszabályozást. Hibás, vagy hiányzó üzenetek esetén biztosítja, hogy a küldő fél újra elküldje a kérdéses üzenetet, a vevő oldal túlterhelődése esetén pedig korlátozza a küldő felet. Általában olyan alkalmazásokban szokták használni, ahol mindenképpen fontos az adatok biztos és sorrendhelyes átvitele, mint például fájlok továbbítása. Hátránya viszont, hogy a megbízható kapcsolat fenntartása végett extra erőforrásokat igényel, összességében kisebb sávszélességet biztosít, mint a datagram alapú protokoll, az UDP.

Az UDP[2] egy nem kapcsolat-alapú átviteli protokoll, melynek segítségével kisebb üzeneteket, ún. datagramokat küldhetünk a másik fél felé. Az UDP nem biztosítja sem a csomagok megérkezését, sem pedig a sorrendhelyes átvitelt, tehát a csomagok elveszhetnek, megsérülhetnek, vagy más sorrendben érkezhetnek meg, mint ahogyan elküldtük őket.

[1] TCP: Transmission Control Protocol
[2] UDP: User Datagram Protocol

Olyan alkalmazásokban, ahol az adatok sérülése nem megengedhető, nem alkalmazható, viszont előnye, hogy mivel nem szükséges biztosítania a megbízható átvitelt, ezért rendkívül gyors. Tipikusan olyan alkalmazásokban használják, ahol sok folyamatosan érkező adatot kell továbbítanunk, amelyek közül néhány megsérülhet, mint például hang vagy videótovábbítás.

Amennyiben a szállítási rétegben kell kommunikációt megvalósítanunk, a fent felsorolt két lehetőség áll rendelkezésünkre. Mindkét esetben úgynevezett socket-ek segítségével végezhetjük a kommunikációt, melyekhez a legtöbb internetes kommunikációt támogató keretrendszerben elérhető beépített támogatás. Gyakran előfordul, hogy ezek a keretrendszerek nem biztosítanak saját absztrakciót ehhez a réteghez, hanem az egyik elterjedt szabványos interfészt valósítják meg, mint például a POSIX alkalmazás programozói felületet.

3.5.3. Alkalmazási réteg

A legtöbb hálózati programozási feladatot az alkalmazási rétegben szoktuk megoldani, ezek azok a protokollok, amelyek alkalmazások közötti kommunikáció pontos menetét magas szinten meghatározzák. Ebben a rétegben rengeteg különböző protokoll közül választhatunk, általában a megvalósítandó feladat határozza meg az alkalmazandó megoldást. Ebben a rétegben a leggyakoribb feladat valamilyen távoli erőforrással történő kommunikáció az interneten keresztül, a protokollt többnyire az elérni kívánt erőforrás, vagy szerver határozza meg, amelyről legfőként az API alapos tanulmányozásával tájékozódhatunk. A továbbiakban ismertem két gyakori IoT eszközökben használt protokoll, a HTTP és az MQTT működését, illetve összehasonlítom ezek felhasználhatóságát.

A HTTP[1] protokoll a mai modern internet alapja, működését tekintve kliens-szerver architektúrát alkalmaz. A kéréseket a kliensek továbbítják a szerver felé, amely a kérésekre valamilyen választ küld. A kérések általában valamilyen erőforrás elérésére vonatkoznak, melyet az ún. URL[2] segítségével azonosítunk. Bár megvalósítható bármilyen szállítási rétegben elhelyezkedő protokoll felett, általában TCP kapcsolaton keresztül működik, mivel az információvesztés nem megengedhető. A HTTP protokollok állapotmentesek, a kapcsolat kezdetén a kliens csatlakozik a szerverhez, elküldi a kívánt erőforrás címét, az elvégezni kívánt műveletet és esetleges további paramétereket. A szerver válaszul visszaküldi az erőforrást, majd lezárja a kapcsolatot. Több erőforrás eléréséhez általában több kérést kell intézni a szerver felé, bár létezik perzisztens üzemmód is. Kódolását tekintve szöveges alapú, emberek számára is olvasható formátumban bonyolítja az adatcserét. Legnagyobb előnye a széleskörű elterjedt használata, manapság gyakorlatilag a legtöbb internetes szolgáltatás HTTP alapú ún. REST[3] API-n keresztül elérhető, legyen szó akár valamilyen internetes adatbázis eléréséről, időjárási adatok lekérdezéséről vagy a közösségi médiáról. A legtöbb hálózati kommunikációra képes mikrokontrolleres keretrendszer beépített módon támogatja. Hátránya, hogy a szöveges adatátvitel és az álla-

[1] HTTP: HyperText Transfer Protocol
[2] URL: Uniform Resource Locator
[3] REST: Representational State Transfer

potmentes kapcsolat miatt rendkívül erőforrásigényes, használata jelentős többletterheléssel jár és egyszerre csak egyirányú kommunikációt tesz lehetővé. A protokoll többféle kéréstípus alkalmazását teszi lehetővé, ezek közül a legelterjedtebbek a POST és a GET kérések. A két metódus közötti legjelentősebb különbség, hogy míg a GET kérések esetében a plusz információk az URL-ben kerülnek továbbításra (így nyilvánosak és könnyen visszakövethetők), a POST kéréseknél ezek az üzenet törzsében helyezkednek el, így az biztonságosabb módja az érzékeny adatok (pl. jelszavak) küldésének. A következő kódrészletben a *www.vik.bme.hu* weboldal lekérésére irányuló kérés és a szerver válasza látható. A kérés a használni kívánt metódus (a példában GET) megjelölésével kezdődik, majd az URL-t és a protokoll verzióját adjuk meg, ezt követik az esetleges egyéb paraméterek, az ún. fejlécek. A szerver válaszában az alkalmazott protokollverzió, egy státuszkód és annak szöveges reprezentációja szerepel. Ezeket követik a válaszban található fejlécek és végül a válasz törzse (az ábrán nem látható) amely tartalmazza a weboldal forráskódját.

```
GET http://www.vik.bme.hu HTTP/1.1
Host: www.vik.bme.hu
Connection: Close

HTTP/1.1 200 OK
Connection: close
Date: Wed, 04 Nov 2020 19:47:22 GMT
Server: Apache/2.2.22 (Debian)
Vary: Accept-Encoding
Content-Type: text/html; charset=UTF-8
Last-Modified: Wed, 04 Nov 2020 19:47:22 GMT
X-Content-Type-Options: nosniff
X-Powered-By: PHP/5.4.45-0+deb7u30
```

A másik gyakran alkalmazott protokoll az IoT alkalmazások területén az MQTT[1], amely egy egyszerű, kis erőforrásigényű megoldás. Architektúráját tekintve publikáció-feliratkozás típusú kapcsolatról beszélünk, melyet kifejezetten alacsony erőforrásigényű hardverekhez és alacsony sávszélességű internetkapcsolathoz fejlesztettek ki, így ideális megoldás jelenthet IoT eszközeink számára. Az MQTT protokollban két entitást különböztetünk meg, ezek a bróker és a kliensek. A bróker feladata, hogy a kliensek által generált üzeneteket továbbítsa a többi érdeklődő kliens számára. A kliensek üzenetei ún. topicokba (témákba) rendeződnek, minden üzenet egy adott topic-hoz érkezik. A többi kliens feliratkozhat tetszőleges témákra, és a bróker minden feliratkozott kliens számára továbbítja a hozzá beérkező üzeneteket. A témák fa-szerű hierarchiába rendezhetők, így lehetőség van egy műveletként egy teljes részfába érkező üzenetekre feliratkozni. Ennek az architektúrának az egyik legnagyobb előnye, hogy teljesen szétválasztja egymástól a kommunikáló feleket, a kliensek között laza-csatolást eredményez, és nem követeli meg, hogy a felek egyáltalán tudjanak egymás létezéséről. Amennyiben egy témára üzenet érkezik, de egyetlen kliens sem iratkozott fel rá, a szerver alapértelmezés szerint eldobja az üzenetet, kivéve, ha azt a küldő kliens tartós tároláshoz megjelölte. Ebben az esetben a szerver a legutoljára érkezett üzenetet tartósan eltárolja és a később feliratkozó kliensek

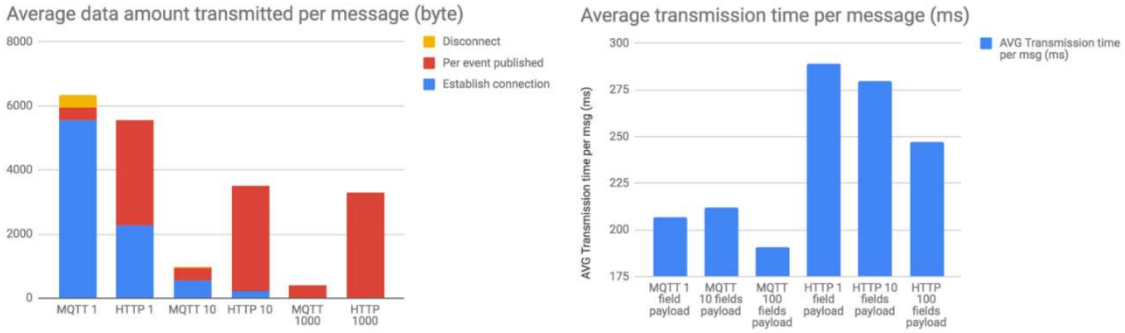
[1] MQTT: Message Queuing Telemetry Transport

számára automatikusan továbbítja. A kapcsolat megbízhatósági követelményeinek megfelelően három szolgáltatási minőség (Quality of Service, QoS) közül választhatunk:

- **Legfeljebb egyszeri kézbesítés:** A bróker az üzenetet csak egyszer próbálja meg kézbesíteni és az esetleges sikertelen küldést nem ismétli meg.
- **Legalább egyszeri kézbesítés:** A bróker az üzenetet többször is megkísérli elküldeni a feliratkozott kliensek számára, egészen addig, amíg egy jóváhagyó üzenetet nem kap.
- **Pontosan egyszer történő kézbesítés:** A bróker és a feliratkozott kliensek kétutas kézfogást valósítanak meg, ezzel biztosítják, hogy az üzenet pontosan egyszer és hiánytalanul megérkezik a feliratkozókhoz.

Az MQTT protokoll legfontosabb előnyei az alacsony erőforrás igény, a kliensek közötti laza csatolás és a rendkívül jó skálázhatóság (néhány tucat klienstől akár több ezer kliensig is terjedhet egy MQTT hálózat). Utóbbi tulajdonsága miatt ideális választást egy nagykiterjedésű, sok limitált erőforrással rendelkező eszközt tartalmazó szenzorhálózat számára, mint amilyen az IoT hálózatok jelentős része. A protokoll hátránya, hogy a centralizált bróker hibája vagy kiesése a teljes hálózat működésképtelenségét okozhatja ezért komolyabb alkalmazásoknál ügyelni kell az esetleges redundancia biztosítására. Ezen felül a bróker éppen a kliensek közötti laza csatolás miatt nem képes intelligens forgalomszabályozásra, illetve képtelen az egyes üzenetek érvényességét ellenőrizni, azokat feltétel nélkül továbbítja a kliensek felé.

Felmerülhet a kérdés, hogy a fent ismertetett protokollok közül melyiket érdemes választanunk egy-egy konkrét alkalmazási példában. A válasz természetesen a körülmények és követelmények részletes mérlegelése után adható meg teljes bizonyossággal, de alább röviden ismertetem egy összehasonlítási példa eredményeit, melyek a 3.5.1. ábrán láthatók.



3.5.1. Ábra – MQTT és HTTP protokollok összehasonlítása [35]

A fenti ábra a két ismertetett protokoll teljesítményének összehasonlítását mutatja a hálózaton továbbított adatmennyiség és a kommunikáló felek közötti reakcióidő tekintetében. A vizsgálat során a szerző két módon tesztelte a protokollok viselkedését: a bal oldali ábra a protokollok által felhasznált adatmennyiséget mutatja az elküldött üzenetek számának függvényében, míg a jobb oldalon az egyes üzenetek hossza alapján vizsgálta a válaszidőket. Megfigyelhetjük, hogy amennyiben csak egyetlen üzenetet akarunk továbbítani, a HTTP protokoll kisebb adatmennyiséget igényel. Ennek oka, hogy az MQTT brókerrel történő kapcsolat felépítése jelentős kommunikációs többletigénnyel jár, amely

az oszlopdiagramon kék színnel szembetűnő módon megfigyelhető. Abban az esetben azonban, ha a már felépült kapcsolatot többször újra felhasználjuk, egyértelműen az MQTT kerül ki győztesen adatfelhasználás szempontjából, ami a HTTP protokoll által minden adatcsere során használt kiegészítő információknak tudható be. Hasonló megfigyeléseket tehetünk, ha az egyes üzenetek hosszát viszonyítjuk a kommunikáció során tapasztalt válaszidőhöz. Ebben az összehasonlításban az MQTT protokoll láthatóan gyorsabban működik a HTTP protokollhoz képest, amely szintén az utóbbi által igényel többletadatoknak köszönhető. Összefoglalva, ha gyakori adatcserére van szükségünk és megválaszthatjuk a használt protokollt, célszerű az MQTT-t választanunk, ebben az esetben viszont ügyelni kell a már egyszer felépített kapcsolat újrahasznosítására. Amennyiben viszont csak ritkán kell adatokat küldenünk vagy fogadnunk, nincs jelentős különbség a két vizsgált megoldás között.

3.5.4. Felhőben történő műveletvégzés

Ahogy az a szakdolgozatom feladatkiírásában is szerepel, az IoT rendszerek egyik jellemzője, hogy az internet szolgáltatásait kihasználva intenzív adatcserét folytatnak egymással és a környezetükkel. Ezek az eszközök általában kevés erőforrással rendelkeznek, ezért könnyen elképzelhető olyan szituáció, ahol a megvalósítandó feladatot az eszköz nem képes egyedül megvalósítani. Ennek oka lehet az elérhető memória, vagy számítási kapacitás szűkössége, vagy az elérni kívánt cél magas komplexitása. Ilyen esetekben az internet segítségével képesek lehetünk a feladatokat több eszköz között megosztani, vagy akár teljesen a felhőbe mozgatni a feladataink egy részét. A szakdolgozatom során praktikussági okokból is alkalmaznom kellett egy hasonló megoldást. A továbbiakban bemutatom a megvalósított időjárásmegfigyelési és előrejelzés megjelenítési rendszer működését, kezdve a paraméterek mérésével, felhőbe történő továbbításával, majd az előrejelzést megvalósító internetes API használatát mutatom be, amelyhez igénybe vettem felhőben futó számításokat is.

Az általam megvalósított eszköz három időjárási paraméter, a hőmérséklet, relatív páratartalom és légnyomás mérésére képes a beépített BME280-as szenzormodul segítségével. A mikrokontrolleren futó alkalmazás indulásakor megkísérli a WiFi hálózathoz való csatlakozást. Amennyiben ez sikeres, az applikáció kezdeményezi a szenzormodulnál a mérések elindítását. Miután a mérések eredményei elérhetővé válnak, azokat egy adatfolyam komponens az internetre történő továbbításhoz megfelelő formátumba alakítja. A felhőben történő adattároláshoz és megjelenítéshez a ThingSpeak internetes IoT platformot használtam. A ThingSpeak egy Matlab alapú nyílt platform, amely kifejezetten IoT alkalmazásokhoz készült és lehetővé teszi a különböző szenzormérések felhőben történő tárolását, illetve azok különböző eszközökön (pl. számítógép, okostelefon) történő megjelenítését. A különböző szenzorokhoz ún. ThingSpeak csatornákat lehet létrehozni, ahol minden csatorna 8 maximum adatmezővel rendelkezhet. Az adatmezők tartalmát a felhasználó határozhatja meg és tetszőleges adatok tárolására alkalmas. A platformra feltöltött adatok visszakereshetők és hozzájuk interaktív vizualizációk készíthetők. Ezek mellett lehetőség van az adatok online, felhőben történő analizálására Matlab segítségével. A szakdolgozatom során létrehozott csatorna olvasásra, megtekintésre publikusan elérhető

és négy adatmezővel rendelkezik. Ezek közül háromhoz, a három mért időjárási paraméterhez tartozik vizualizáció, a negyedik mező pedig az időjárás előrejelzés tárolását szolgálja. A ThingSpeak platformmal történő kommunikáció történhet egyaránt HTTP vagy MQTT protokoll segítségével. A megvalósított rendszer HTTP protokollt alkalmaz, mivel az adatcsere gyakorisága meglehetősen alacsony, és kezdetben az időjárás előrejelzés lekérdezése is közvetlenül a mikrokontroller által történt, utóbbi API pedig csak HTTP protokollt támogat. A mérések feltöltése a következő HTTP kérés segítségével történik:

```
POST https://api.thingspeak.com/update HTTP/1.1
Host: api.thingspeak.com
Content-Type: application/x-www-form-urlencoded
Content-Length: <automatikus>

api_key=5PVLUGAUWS0XXXXX&field1=<hőmérséklet>&field2=<nyomás>&field3=<páratartalom>
```

A megfelelő URL specifikációja után a mérési adatok és a hitelesítéshez használt API kulcs a kérés törzsében utaznak a szerver felé. A ThingSpeak támogatja a feltöltést GET kérések használatával is, azonban ebben az esetben az érzékeny adatnak minősülő hitelesítő kulcs is része az URL-nek, ezért kevésbé biztonságos.

A nyers mérési adatok felhőbe történő továbbítása mellett az eszköz képes az internet segítségével lekérdezni a következő néhány nap időjárás előrejelzését is, majd azokat a beépített kijelzőn megjeleníteni. A feladat specifikációjának megfelelően az eszköz a következő három nap átlaghőmérsékletét és az adott napon várható időjárási feltételeket reprezentáló szimbólumokat jelenít meg. Bár a kezdeti elképzeléseim szerint a pillanatnyi időjárási paraméterek közül az alkalmazás lekérdezi a felhőzet százalékos értékét és a körülményeket reprezentáló azonosítót is, ezeket végül nem jelenítettem meg. Az előrejelzéseket az OpenWeatherMap nyílt internetes platform alkalmazásával szerzi meg az applikáció, amely HTTP protokollt támogat. Az előrejelzés adatokat a következő GET kérés segítségével olvashatjuk ki (az olvashatóság kedvéért az URL-t több sorba tördeltem a paraméterek mentén):

```
GET https://api.openweathermap.org/data/2.5/onecall?
lat=<földrajzi szélesség>&
lon=<földrajzi hosszúság>&
exclude=alerts,hourly,minutely&
appid=<API kulcs>&
units=metric
HTTP/1.1
Content-Type: json
```

Az előrejelzés helyének földrajzi szélességének és hosszúságának megadása után specifikáljuk azokat az adatokat, amelyeket nem szeretnénk megkapni a szerver válaszában, esetemben ezek a figyelmeztetések, az óránkénti és percenkénti előrejelzés. A kizárt adatok után szerver válaszában a pillanatnyi időjárási értékek és a napi előrejelzések szerepelnek a következő két hétre vonatkozóan. A hitelesítéshez használt API kulcs és a preferált mértékegységek specifikációja után a kérésre válaszul a szerver elküldi a kívánt paramétereket a válasz törzsében JSON formátumban. Kezdetben megkíséreltem a platformtól közvetlenül a mikrokontroller által lekérdezni és feldolgozni az adatokat, azonban mivel az előrejelzés rengeteg számomra többnyire nem szükséges paramétert tartalmaz, hamar problémák adódtak a mikrokontrolleren elérhető szabad RAM memória kor-

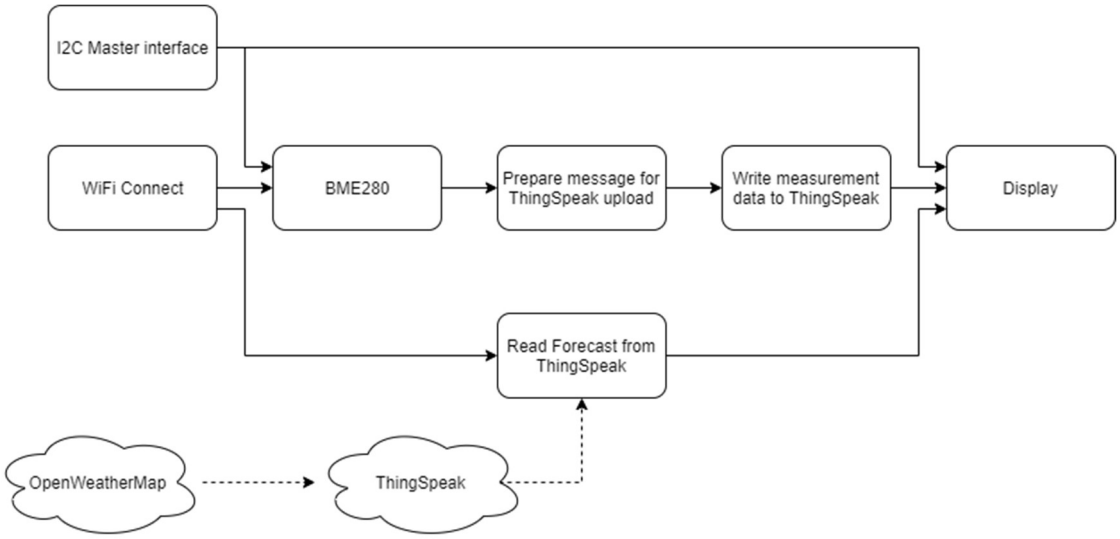
látaival. A megoldás a már korábban említett módon a feladatok felhőbe történő átmozgatása jelentette. A ThingSpeak lehetőségeit kihasználva képesek vagyunk periodikusan egy felhőben futó Matlab kód segítségével analizálni a feltöltött adatokat. Ebben a Matlab kódban viszont lehetőségünk van tetszőleges egyéb műveletek elvégzésére is, például az időjárás előrejelzésre használt API lekérdezésére is. A választott megoldás tehát, hogy a felhőben periodikusan (kb. 10 percenként) futó Matlab kód végzi az OpenWeatherMap platform lekérdezését, a szerver válaszaként kapott JSON objektumot feldolgozza, kibányássza belőle a számomra érdekes paramétereket, majd az eredetinél sokkal kisebb JSON formátumú leírást feltölti a negyedik (nem megjelenített) ThingSpeak adatmezőbe. A mikrokontroller ezek után kizárólag csak a ThingSpeak szerverrel kommunikál, az előrejelzés adatokat pedig erről a rejtett adatmezőből szerzi meg. Ezzel a megoldással már lehetőség van a kisméretű, csak a számomra szükséges adatokat tartalmazó JSON leírás feldolgozására és megjelenítésére a mikrokontrolleren.

A felhőben futó Matlab kód lényegesebb részei alább láthatók, az URL összeállítása után megtörténik a szerver lekérdezése, majd a szükséges adatok kibányászása, végül azok feltöltése a ThingSpeak-re:

```
% Specifying content type as JSON for the response
options = weboptions('ContentType','json');

% Assembling the request URL for the forecast API
url = [ 'https://api.openweathermap.org/data/2.5/onecall?', ...
        'lat=',latitude, ...'&lon=', longitude,'&exclude=',exclude, ...
        '&appid=',appid,'&units=', units ];

% Reading current weather and forecast data from the API
data = webread(url, options);

% Extracting current weather conditions
parsed.now.id = data.current.weather.id;
parsed.now.clouds = data.current.clouds;

% Extracting daily forecast data for the next 3 days
for i = 1:3

    % Extracting weekday ID for displaying abbreviation on the MCU
    parsed.forecast(i).day = weekday(datetime(data.daily{i+1,1}.dt, ...
                                'ConvertFrom', 'posixtime')) - 1;

    % Extracting weather condition ID
    parsed.forecast(i).id = data.daily{i+1,1}.weather.id;

    % Extracting forecasted day temperature
    parsed.forecast(i).temperature.day = data.daily{i+1,1}.temp.day;

end

% ThingSpeak Channel ID
channel = 1168081;

% ThingSpeak channel API-key for writing
writeKey = '5PVLUGAUWS0XXXXX';

%Converting structured data to JSON format
jsonData = jsonencode(parsed);

%Writing JSON formatted data to ThingSpeak channel
thingSpeakWrite(channel, {jsonData}, 'Fields', 4, 'WriteKey', writeKey);
```

A teljes hálózati kommunikációt és a méréseket megvalósító adatfolyamot a 3.5.2. ábra mutatja, a folytonos nyilak a mikrokontrolleren található folyamkomponensek közötti

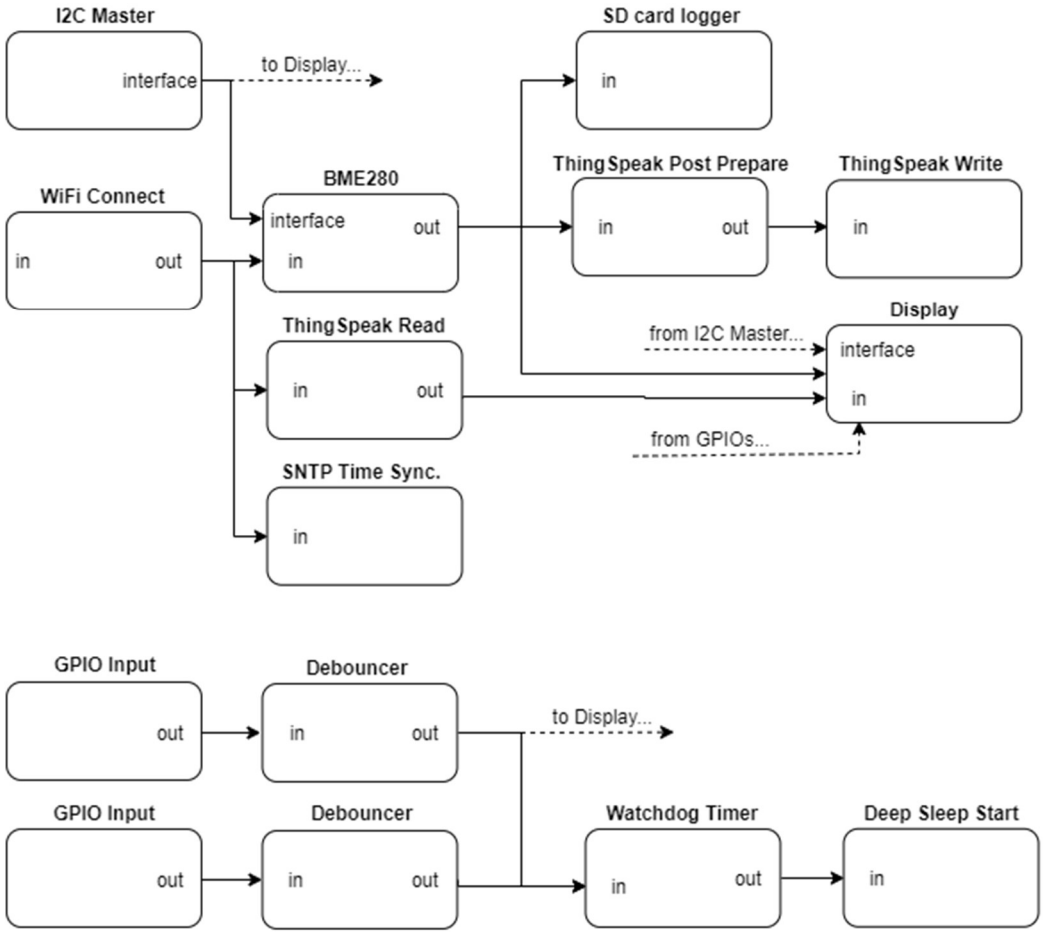
kapcsolatokat jelzik, míg a szaggatott vonalak az alkalmazáson kívül fennálló adatáramlást jelzik. Az ábrán kizárólag a méréseket és az előrejelzést érintő komponenseket tüntettem fel. Az alkalmazás indulásakor az I2C interfészt megvalósító komponenst csatlakoztatjuk a szenzorhoz és a kijelzőhöz egyaránt, mivel mindkettő ugyanazt a buszt használja. A WiFi hálózathoz történő sikeres csatlakozás eseménye kezdeményezi a szenzor méréseinek elindítását, az eredmények megjelenése pedig a felhőbe történő feltöltés folyamatát. Végül a mért értékek a folyam ezen ágán áthaladva eljutnak a kijelzőhöz, amely a pillanatnyi állapotától függően esetlegesen megjeleníti azokat. A hálózati csatlakozás ezzel párhuzamosan egy másik folyamágon elindítja az előrejelzés adatok lekérdezését a ThingSpeak szerverről, amely a háttérben eközben periodikusan kommunikációt folytat az OpenWeatherMap szerverével. Fontos kihangsúlyozni, hogy az egyes folyamágak között implicit párhuzamosítási lehetőség van, a mikrokontroller operációs rendszerének ütemezője képes konkurensen futtatni az egyes folyamágakat anélkül is, hogy ezt explicit jelezni kellene az alkalmazáskódban.



3.5.2. Ábra – Mérések és előrejelzés adatok kezelése adatfolyam diagramon

3.6. Összefoglalás

A korábbi fejezetekben már több alkalommal szerepeltettem az elkészített alkalmazás kódjának, adatfolyamos leírásának egyes részleteit, a továbbiakban ismertetem a teljes adatfolyam struktúrát és összegzem a tapasztalatokat. A teljes adatfolyam diagram a 3.6.1. ábrán szerepel. A figyelmes olvasó felfedezheti a hasonlóságot az 1.2.2. ábrán szereplő funkcionális rendszertervvel, melynek egyes elemei és adatútjai visszaköszönnek a megvalósított adatfolyamban.



3.6.1. Ábra – A teljes mérési és hálózati adatfolyam

Az alkalmazás viselkedése felbontható egy statikus, minden futáskor végrehajtandó részre, és egy dinamikus, a felhasználói interakcióktól függő részre. A statikus rész a 3.6.1. Ábra felső részén látható.

A mikrokontroller ébredésekor automatikusan inicializálásra kerül az I2C interfész az *I2C_Master* komponensben, melynek kimenetén megjelenik az interfész eléréséhez szükséges memóriacím. Ez a cím az adatkapcsolatokon keresztül eljut a szenzorhoz és a kijelzőhöz, melyek a címet elmentik, majd ezen keresztül használják az I2C buszt a későbbiekben. Az I2C kommunikációs absztrakciókról bővebben a 3.4 fejezetben volt szó.

Ugyancsak a mikrokontroller ébredésekor történik meg az automatikus csatlakozás a WiFi-hez, melynek sikerességéről a kimenetén megjelenő státuszüzenet tájékoztat. A *WiFi_Connect* komponens működését egy manuális, ún. kezdeti üzenet kezdeményezi, melyet az adatfolyam komponensek létrehozása közben küld az inicializáló kód. Amennyiben a csatlakozás sikeres, a kimeneten logikai igaz értéket tartalmazó üzenet jelenik meg, ellenkező esetben nem történik semmi, így nem indulnak el azok a folyamatok sem, amelyek a hálózati csatlakozás hiányában feleslegesek lennének. A sikeres csatlakozás három ágon indítja el a további műveletvégzést. Egyrészt kezdeményezi a szenzorméréseket (*BME280* komponens), másrészt az időjárás előrejelzés lekérdezését (*Thingspeak_Read* komponens) és továbbítását a megjelenítőhöz (*Display* komponens), illetve a dátum és idő szinkronizációját (*SNTP* komponens).

A *BME280* komponens valósítja meg a szenzoros mérések kezelését. Inicializációja során blokkoló módon várakozik a kommunikációhoz használt I2C interfész címének megérkezésére, majd pedig a sikeres WiFi csatlakozásra. Amennyiben mindkét előfeltétel teljesült, elvégzi a hőmérséklet, nyomás és páratartalom mérését, majd az eredményeket egy közös üzenetben továbbítja a kimenetére kapcsolt komponensek felé. Egyrészt a mért értékek eljutnak a *Display* komponenshez az OLED kijelzőn történő megjelenítéshez, másrészt pedig az *SD_Logger* komponenshez, ami az üzenet megérkezésekor elvégzi a beépített SD kártya csatlakoztatását (FAT fájlrendszeren keresztül), és a mérési adatokat a hozzájuk tartozó időbélyeggel együtt elmenti a kártyán található CSV fájlba.

A fentieken kívül a mérési eredmények továbbításra kerülnek a *Thingspeak_PostPrepare* komponens felé, amely olyan formátumba alakítja az eredményeket tartalmazó üzenetet, hogy az kompatibilis legyen az internetre történő feltöltésre, melyet a *Thingspeak_Write* komponens végez el, HTTP kapcsolaton keresztül. A *Display* komponensben a megjelenítendő adatok perzisztens tárolásra kerülnek a mikrokontroller RTC RAM memóriájában, így a komponens alvó állapotban is képes azok megjegyzésére. A következő induláskor tehát nem kell megvárnia a kijelzéssel az új adatok megérkezését, addig megjelenítheti az előző futásból származó adatokat. A szenzorillesztő és a megjelenítő komponensekről bővebben a 3.4 fejezetben volt szó, míg az internetes kommunikációt megvalósító komponenseket a 3.5.4 fejezetben részleteztem.

Az adatfolyam gráf eddig ismertetett része minden futáskor végrehajtásra kerül. A diagram dinamikus része felelős a felhasználóval történő interakciók lebonyolításáért, melynek működése a 3.6.1. Ábra alsó részén látható. A nyomtatott áramköri panel három nyomógombot tartalmaz, ebből egy közvetlenül a mikrokontroller újraindításáért felelős. A másik kettő szabad felhasználású, a konkrét alkalmazásban a kijelző egyes lapjainak léptetésére használtam fel őket. A két darab *GPIO_Input* komponens felelős a gombnyomások érzékelésért. Az események detektálásra megszakítások segítségével történik, melyek a gombok kimenetének lefutó élére következnek be, nyugalmi állapotban azokat egy-egy felhúzó ellenállás magas szinten tartja. A gombnyomások lefutó éleinek hatására egy logikai igaz érték jelenik meg a megfelelő komponensek kimenetén.

Mivel a hardverbe épített nyomógombok viszonylag nagyméretűek, ezért hajlamosak a pergésre, melyet a melléjük tervezett kondenzátorok valamelyest csillapítanak, de időnként előfordulnak többszörösen érzékelt lenyomások. Ezek kiküszöbölésére egy szoftveres pergésmentesítő komponenst (*Debouncer*) alkalmaztam, amely specifikálható periódusidővel képes a többszörös lenyomások szűrésére. A konkrét esetben a gombok méretét figyelembe véve ez a pergésmentesítési intervallum 50ms, amely még nem zavaróan lassú, de hatékony működést eredményez. A pergésmentesített nyomógomb bemenetek továbbításra kerülnek a kijelzőt kezelő komponens (*Display*) felé is, így az képes a megjelenített adatok közötti váltásra. Emellett a nyomógombok kimenete újraindítja az inaktivitás figyeléséért felelős időzítőt (*Watchdog* komponens) is.

Az alkalmazás indulásakor automatikusan inicializálásra kerül egy szoftveres időzítő a *Watchdog* komponens belsejében, amely 10 másodpercnyi inaktivitás után egy logikai

igaz értéket produkál a kimenetén, ellenkező esetben pedig nem történik semmi. Az időzítő újraindítását a nyomógomboktól érkező impulzusok végzik. Amennyiben 10 másodpercig nem érkezik ilyen jelzés, a kimeneten generálódó érték kezdeményezi a deep-sleep üzemmódot, melynek megvalósításáért a *DeepSleep* komponens felelős. Ebben a komponensben konfigurálom az alvási módból való felébredési lehetőségeket (10 perc után időzítővel, vagy előtte a nyomógombok hatására), majd elvégzem a mikrokontroller alacsony fogyasztású módba léptetését.

A korábbi fejezetekben összefoglaltam a programozási nyelv és mikrokontrolleres keretrendszer kiválasztásának legfontosabb tudnivalóit, ismertettem a különböző beágyazott szoftveres rendszerarchitektúrákat, bemutattam az adatfolyam architektúrát és annak egy egyszerűbb megvalósítását. Ezen felül tárgyaltam a beágyazott eszközök illesztőprogramjainak egyes alkalmazási mintáit és a különböző hálózati kommunikációs megoldásokat IoT rendszerekben. Végül bemutattam a szakdolgozatom keretében elkészített IoT eszköz szoftverének adatfolyam alapú működésének teljes struktúráját.

Köszönetnyilvánítás

Szeretném köszönetemet kifejezni Lernyei Marcellnek a hardveres fejezet előzetes bírálatáért és az elkészült nyomtatott áramköri terv ellenőrzése során nyújtott hasznos tanácsaiért. Továbbá köszönöm Naszály Gábornak a nyomtatott áramkör legyártásában és gyártás előtti ellenőrzésében nyújtott támogatását és rendkívül részletes észrevételeit. Köszönöm szüleimnek, barátaimnak a végtelen türelmüket, amit a szakdolgozatom elkészítése közben irántam tanúsítottak. Köszönöm Révay Reginának a rengeteg támogatást és az alkalmazásban felhasznált grafikák elkészítését.

4. Hivatkozások

[1] Wikipedia, „Dolgok Internete,” [Online]. Available: https://hu.wikipedia.org/wiki/Dolgok_internetje.

[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari és M. Ayyash, „Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications,” IEEE, IEEE Communications Surveys & Tutorials, 2015.

[3] KiCad, „Getting Started,” [Online]. Available: https://kicad-pcb.org/help/getting-started/.

[4] G. Tevesz és Z. Szabó, „Mikrokontroller alapú rendszerek - elektronikus jegyzet”.

[5] I. V. Evdokimov, A. R. J. Alalwan, R. Y. Tsarev, T. N. Yamskikh, O. A. Tsareva és A. N. Pupkov, „A cost estimation approach for IoT projects,” *Journal of Physics: Conference Series,* 2019.

[6] K. Venkatesh, S. Hemaswathi, R. Balakrishnan és P. Rajkumar, „IoT Based Home Automation Using Raspberry Pi,” *Journal of Advanced Research in Dynamical and Control Systems,* 2018.

[7] A. Maier, A. Sharp és Y. Vagapov, „Comparative Analysis and Practical Implementation of the ESP32 Microcontroller Module for the Internet of Things,” 2017.

[8] J. Horstmann, M. Ramsbeck és S. Bosse, „Analog Sensor Signal Processing and Analog-to-Digital Conversion: Technology and Applications,” in *Material-Integrated Intelligent Systems - Technology and Applications*, 2017, pp. 257-280.

[9] G. Orosz, „Beágyazott és Ambiens Rendszerek - előadásdiák”.

[10] ElectronicsTutorials, „Thermistors,” [Online]. Available: https://www.electronics-tutorials.ws/io/thermistors.html.

[11] B. Kiss és G. Kovács, „Ipari irányítástechnika - előadásdiák”.

[12] M. Tomasz, „Use Analog Techniques To Measure Capacitance In Capacitive Sensors,” ElectronicDesign, 13 11 2012. [Online]. Available: https://www.electronicdesign.com/technologies/analog/article/21796004/use-analog-techniques-to-measure-capacitance-in-capacitive-sensors.

[13] Battery University, „BU-808: How to Prolong Lithium-based Batteries,” 17 09 2020. [Online]. Available: https://batteryuniversity.com/learn/article/how_to_prolong_lithium_based_batteries.

[14] Espressif Systems, „ESP32 DevKitC V4,” 06 12 2017. [Online]. Available: https://dl.espressif.com/dl/schematics/esp32_devkitc_v4-sch.pdf.

[15] R. Cerda, „Pierce-Gate Crystal Oscillator, an introduction,” 03 2008. [Online]. Available: https://www.crystek.com/documents/appnotes/pierce-gateintroduction.pdf.

[16] P. Wilson, The Circuit Designer's Companion 4th Edition, Newnes, 2017.

[17] J. Z. Lee W. Ritchey, Right the First Time: a Practical Handbook on High Speed Pcb and System Design I., Speeding Edge, 2003.

[18] H. Zumbahlen, „Staying Well Grounded,” Analog Devices, [Online]. Available: https://www.analog.com/en/analog-dialogue/articles/staying-well-grounded.html#.

[19] National Instruments, „SPICE Simulation Fundamentals,” National Instruments, 16 05 2019. [Online]. Available: https://www.ni.com/hu-hu/innovations/white-papers/06/spice-simulation-fundamentals.html.

[20] AspenCore, „2019 Embedded Markets Study (2019),” 2019. [Online]. Available: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf.

[21] J. Beningo, „5 Reasons to start using C++ over C,” 15 02 2018. [Online]. Available: https://www.beningo.com/5-reasons-to-start-using-c-over-c/.

[22] Espressif Systems, „Get Started,” Espressif Systems, [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html.

[23] B. John T., „Processes,” University of Illinois at Chicago, [Online]. Available: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html.

[24] „NodeRed,” [Online]. Available: https://nodered.org/.

[25] J. P. Morrison, Flow Based Programming: A New Approach to Application Development, Van Nostrand Reinhold, 1994.

[26] Circuit Basics, „Basics of UART communication,” [Online]. Available: https://www.circuitbasics.com/basics-uart-communication/.

[27] Sparkfun Electronics, „I2C,” [Online]. Available: https://learn.sparkfun.com/tutorials/i2c/all.

[28] Sparkfun Electronics, „Serial Peripheral Interface (SPI),” Sparkfun Electronics, [Online]. Available: https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all.

[29] Wikipedia, „Finite impulse response,” [Online]. Available: https://en.wikipedia.org/wiki/Finite_impulse_response#/media/File:FIR_Filter.svg.

[30] MathWorks, „FIR Filter Design,” [Online]. Available: https://www.mathworks.com/help/signal/ug/fir-filter-design.html.

[31] Wikipedia, „Infinite impulse response,” [Online]. Available: https://en.wikipedia.org/wiki/Infinite_impulse_response#/media/File:IIR-filter.png.

[32] MathWorks, „IIR Filter Design,” [Online]. Available: https://www.mathworks.com/help/signal/ug/iir-filter-design.html.

[33] eletokosan.hu, „Mi az a Zigbee?,” eletokosan.hu, 21 11 2018. [Online]. Available: https://eletokosan.hu/mi-az-a-zigbee/.

[34] Bravosmart, „Mi az a Z-Wave?,” Bravosmart, 09 09 2018. [Online]. Available: https://bravosmart.hu/mi-az-a-z-wave/.

[35] C. Wang, „HTTP vs. MQTT: A tale of two IoT protocols,” 26 11 2018. [Online]. Available: https://cloud.google.com/blog/products/iot-devices/http-vs-mqtt-a-tale-of-two-iot-protocols.

[36] Espressif Systems, „Get Started,” [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/_images/what-you-need1.png.

5. Függelékek

5.1. Kapcsolási rajz és nyomtatott áramkör

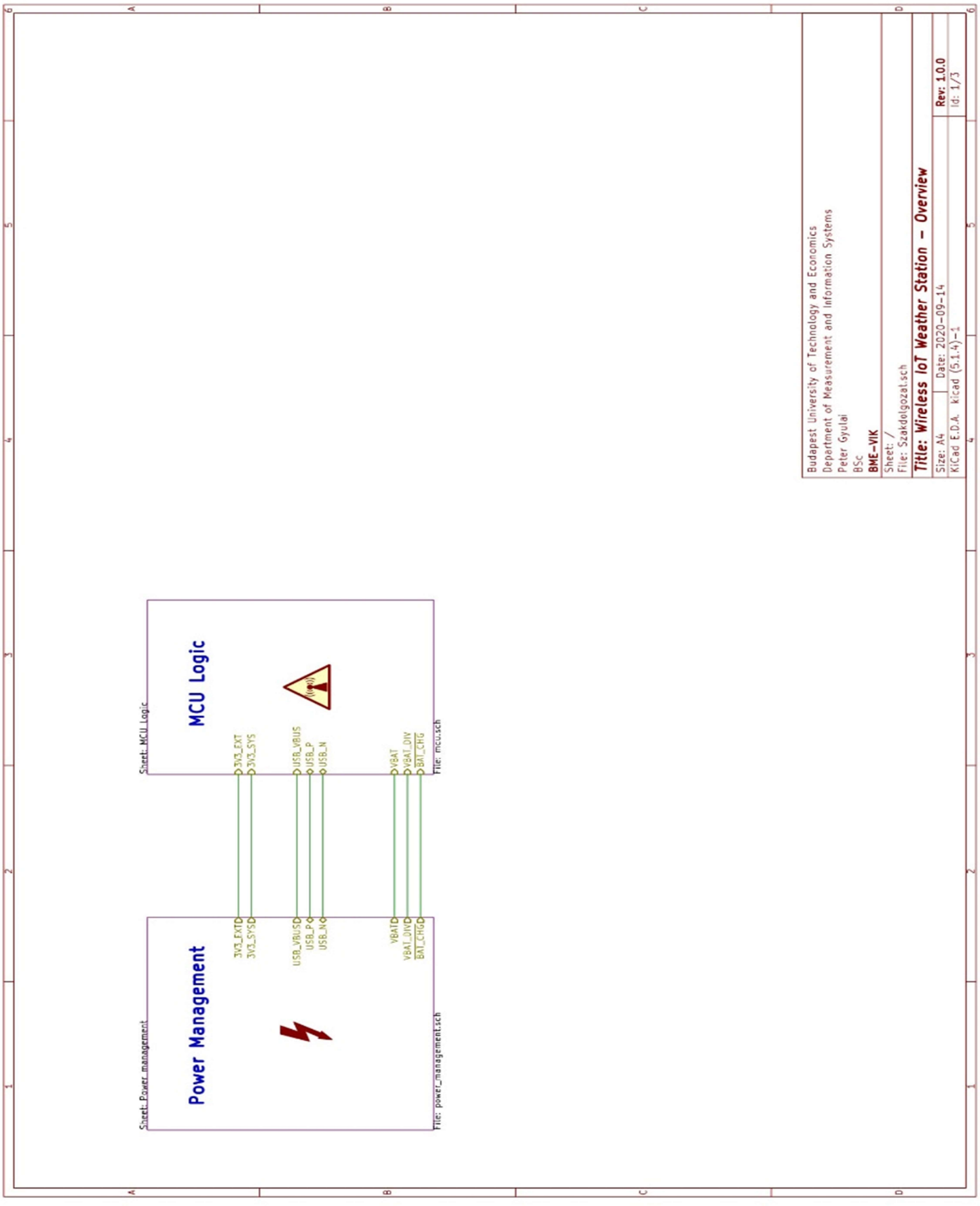
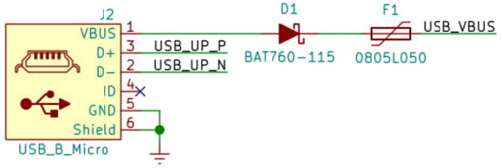


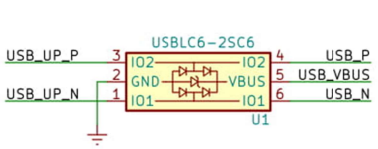
Figure 1.
Micro-USB Connection



Power and data is received through the micro-USB connector. The purpose of the D1 rectifier Schottky-diode is to prevent reverse current flow towards the USB bus in case of a faulty device. The F1 resettable polyfuse acts as a safeguard against overcurrent faults. For USB data line protection, see Figure 2.

D1 forward voltage: max. 550mV
D1 forward current: max. 1A
F1 hold current: min. 500mA
F1 tripping current: max. 1A

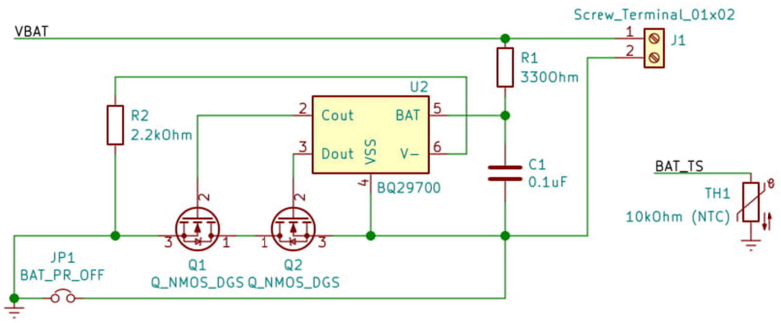
Figure 2.
USB Data line ESD protection



The USB data lines are protected against ESD fault events by a dedicated protection circuit. The device integrates four voltage clamping diodes and a Zener-diode for bus voltage stabilizing bus voltage. The data lines are used by a USB/UART bridge, see Figure 10 on the MCU Logic sheet.

U1 breakdown voltage: min. 6V

Figure 3.
Li-ion/Li-Po Battery protection with bypass jumper

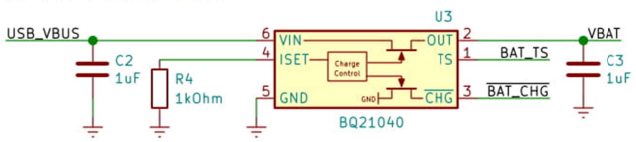


Li-ion and Li-Po batteries are extremely sensitive for overcharging, overdischarging and overcurrent events. Batteries with these types of chemistry must be protected against faults. This circuit protects the battery from overcurrent, overcharge and overdischarge by enabling or disabling the power FETs. Most commercial batteries have integrated protection on the pack, so this part of the circuit can be bypassed by shorting the BAT_PR_OFF jumper. The battery temperature can be monitored using the TH1 NTC-type thermistor.

Fault detection thresholds (see datasheet):

U2 over-charge:	4.275V
U2 over-discharge:	2.8V
U2 charge overcurrent:	0.1V
U2 discharge overcurrent:	0.1V
U2 load short circuit:	0.5V

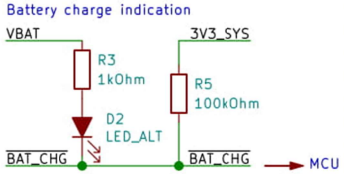
Figure 4.
Li-ion/Li-Po Battery charger



The battery is charged using a highly integrated Li-ion / Li-Po charger with programmable fast-charge current. The device is capable of operating from a USB port and is protected against overvoltage conditions. Battery temperature is monitored using the NTC thermistor from Figure 3. Charging status is indicated by an open-drain terminal pulling to ground while charging. Status is indicated by an LED and is monitored by the microcontroller.

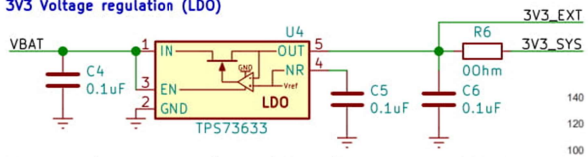
U3 fast-charge current: 540 mA

Battery charge indication

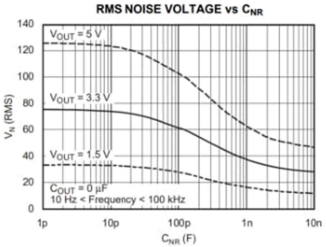


The battery charging status is indicated by lighting a yellow LED during charge. The microcontroller also monitors the charge status. The status indication line is pulled logic-high by a pull-up resistor during no-charge. When charging, the BAT_CHG line is held low by the charger.

Figure 5.
3V3 Voltage regulation (LDO)

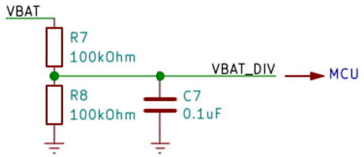


LDO output current: 500mA
LDO dropout voltage: 75mV



Battery voltages vary over time and the system needs a regulated, stable 3.3V power supply. Line voltage is regulated by a fixed 3.3V low-dropout regulator (LDO). Using an LDO provides great noise rejection which is further ensured by the input and output capacitors. The device also features a noise rejection pin to which a capacitor can be connected to minimize noise resulting from the bandgap reference of the LDO. Output current to the system flows through a 0 Ohm resistor, which can be replaced by a sense-resistor later for current monitoring. Externally connected PCBs can draw power from the 3V3_EXT line.

Figure 6.
Battery voltage monitoring



Battery voltage is monitored by the MCU by using a voltage divider. The divider circuit is necessary because the battery voltage usually exceeds the supply voltage of the MCU and would cause damage. To minimize leakage current high value resistors are used. The C7 capacitor is used to filter measurement noise and provide a charge buffer for the ADC of the microcontroller.

External sheet connections

USB_VBUS	USB_VBUS
USB_P	USB_P
USB_N	USB_N
VBAT	VBAT
VBAT_DIV	VBAT_DIV
BAT_CHG	BAT_CHG
3V3_SYS	3V3_SYS
3V3_EXT	3V3_EXT

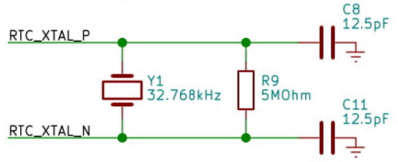
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Peter Gyulai
BSc
BME-VIK

Sheet: /Power management/
File: power_management.sch

Title: Wireless IoT Weather Station – Power management

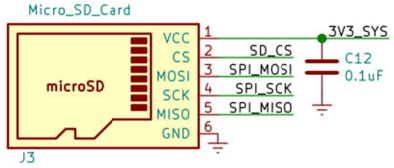
Size: A4	Date: 2020-09-15	**Rev: 1.0.0**
KiCad E.D.A. kicad (5.1.4)-1		Id: 2/3

Figure 7.
Real-time clock (RTC) crystal oscillator



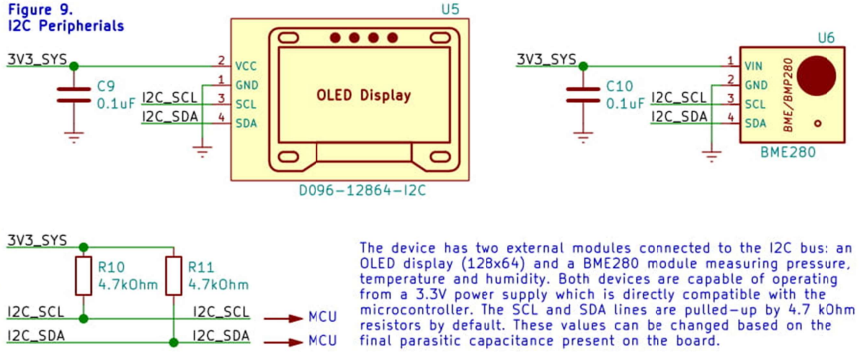
The microcontroller is capable of keeping precise time information during sleep-modes by using an external 32.768 kHz quartz crystal oscillator. The nominal load capacitance of the crystal is 12.5 pF. As this is not a standard capacitor value 10 pF capacitors can be used as a replacement. This results in a small error on the crystal accuracy, but can be compensated for by software. The parallel resistor is used to bias the crystal.

Figure 8.
Micro-SD Card module



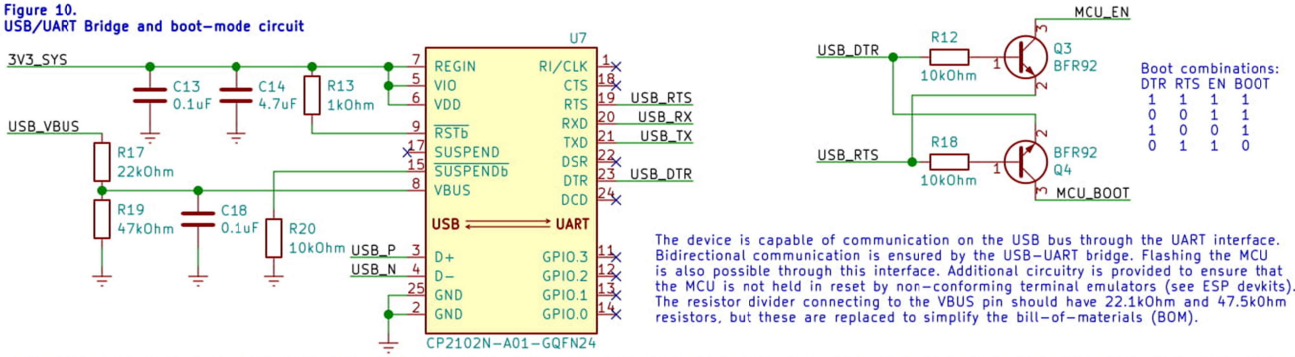
The device is capable of persistent data storage by using an external micro-SD card module. The module is connected to the board by a six-pin connector, which exposes the power and SPI connections. The card itself operates from 3.3V which is directly compatible with the microcontroller.

Figure 9.
I2C Peripherials



The device has two external modules connected to the I2C bus: an OLED display (128x64) and a BME280 module measuring pressure, temperature and humidity. Both devices are capable of operating from a 3.3V power supply which is directly compatible with the microcontroller. The SCL and SDA lines are pulled-up by 4.7 kOhm resistors by default. These values can be changed based on the final parasitic capacitance present on the board.

Figure 10.
USB/UART Bridge and boot-mode circuit

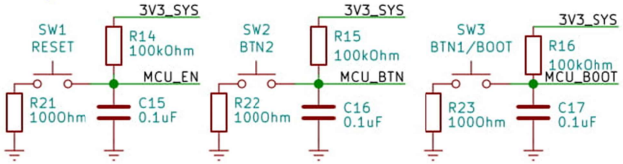


Boot combinations:

DTR	RTS	EN	BOOT
1	1	1	1
0	0	1	1
1	0	0	1
0	1	1	0

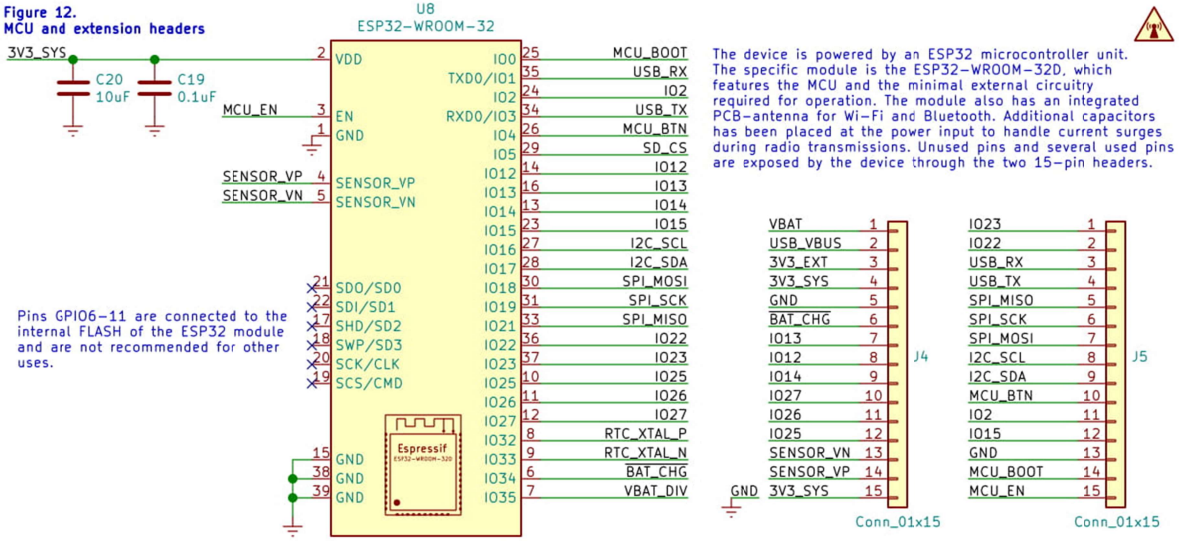
The device is capable of communication on the USB bus through the UART interface. Bidirectional communication is ensured by the USB-UART bridge. Flashing the MCU is also possible through this interface. Additional circuitry is provided to ensure that the MCU is not held in reset by non-conforming terminal emulators (see ESP devkits). The resistor divider connecting to the VBUS pin should have 22.1kOhm and 47.5kOhm resistors, but these are replaced to simplify the bill-of-materials (BOM).

Figure 11.
User push-buttons



The end-user can physically interact with the device using the three push-buttons provided on the PCB. Two of the buttons are for debug purposes, namely for resetting the device and selecting FLASH boot mode. The third button is a general purpose input for usage in the application software. All of the buttons are active-low and are pulled up by resistors when inactive. Pressing the buttons strongly pulls the input lines low. Debouncing is achieved by the parallel capacitors between ground and the input lines.

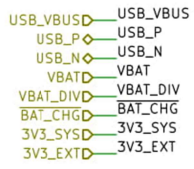
Figure 12.
MCU and extension headers



Pins GPIO6-11 are connected to the internal FLASH of the ESP32 module and are not recommended for other uses.

The device is powered by an ESP32 microcontroller unit. The specific module is the ESP32-WROOM-32D, which features the MCU and the minimal external circuitry required for operation. The module also has an integrated PCB-antenna for Wi-Fi and Bluetooth. Additional capacitors has been placed at the power input to handle current surges during radio transmissions. Unused pins and several used pins are exposed by the device through the two 15-pin headers.

External sheet connections



Budapest University of Technology and Economics
Department of Measurement and Information Systems
Peter Gyulai
BSc
BME-VIK

Sheet: /MCU Logic/
File: mcu.sch

Title: Wireless IoT Weather Station – MCU Logic

Size: A4	Date: 2020-09-15	**Rev: 1.0.0**
KiCad E.D.A. kicad (5.1.4)-1		Id: 3/3

5.2. Nyomtatott áramkör