Szegedi Tudományegyetem
Informatikai Tanszékcsoport

Szenzorok illesztése STM32-es mikrovezérlőhöz sportteljesítmény mérése céljából

Szakdolgozat

Készítette:

Lovászi Zsuzsanna
mérnökinformatikus
BSc szakos hallgató

Témavezető:

Dr. Mingesz Róbert
egyetemi adjunktus

Társ-témavezető:

Tönköly Andor
PhD hallgató

Szeged
2023

Feladatkiírás

A modern informatika korában egyre elterjedtebb a mérés mint alkalmazás és fogalom szükségessége, ipari és kutatási területen is számos olyan alkalmazás merül fel, melyek környezeti paraméterek monitorozását igénylik, kiegészítve természetesen szükséges döntéshozatallal. Egy ilyen felhasználási terület a különböző sporttevékenységek közben felmerülő fizikai mennyiségek mérése, a sportolók tevékenységnek monitorozása, ebből megfelelő, akár hosszútávú következtetések levonása, segítségével pedig összehasonlító elemzések végzése, ezekkel elősegítve a sportoló fejlődését, és csökkentve sérülésének kockázatát. Ahhoz, hogy az említett teljesítményt vizsgálni tudjuk digitális tartományban szenzorok szükségesek, melyeknek napjainkra széles tárháza elérhető, mégis akadnak egyes feldolgozó egységek, többnyire mikrovezérlők, melyekkel a szenzorok széles skálájának használata nehézkes, lévén, hogy a feldolgozó egység és az eszköz közötti szoftverréteg nem áll rendelkezésre, amennyiben pedig igen, annak minősége nem garantált.

A hallgató feladata kiválasztani azokat a szenzorokat, amelyek által érzékelt mennyiségek szükségesek és hasznosak sportteljesítmény mérése céljából, elkészíteni a szükséges szoftver elemeket, melyekkel a kommunikáció megvalósítható a szenzorokkal, egy alkalmazás tervezése mely autonóm módon képes méréseket elvégezni, illetve próbamérések végzése, mely az elkészített alkalmazás alkalmasságát bizonyítják.

Tartalmi összefoglaló

A téma megnevezése

Szenzorok illesztése STM32-es mikrovezérlőhöz sportteljesítmény mérése céljából

A megadott feladat megfogalmazása

A feladat egy olyan eszköz prototípusának megtervezése és kivitelezése, mely képes mérni egy sportoló mozgását edzés közben. A mért adatok feldolgozásával számszerűsíthető az elvégzett edzésmunka, melynek köszönhetően képet kaphatunk a sportoló valós teljesítményéről. Ennek megjelenítéséhez egy grafikus felhasználói felülettel rendelkező alkalmazás létrehozása is szükséges.

A megoldási mód

Először is meg kellett határozni, hogy melyek azok a mozgások, amelyek szenzorokkal mérhetőek és hasznos információval szolgálnak a sportteljesítmény elemzésben. A legalkalmasabb szenzorok és mikrovezérlő kiválasztása után következett a szenzor könyvtárak megírása és tesztelése. Majd a már működő eszköz által mért adatok feldolgozására készítettem egy szoftvert, mellyel a legfontosabb információk megjeleníthetőek felhasználó számára.

Alkalmazott eszközök, módszerek

Az eszköz központi egysége egy NUCLEO-L412KB mikrovezérlő panel, ehhez van hozzákapcsolva egy NEO-7m GPS modul, egy MPU6050 inerciális mérőegység, egy HMC5883L magnetométer és egy SD kártya író-olvasó modul. A mikrovezérlő programozását STM32CubeIDE fejlesztői környezetben végeztem C nyelven, a méréseket feldolgozó és megjelenítő asztali alkalmazást PyCharm programban készítettem python nyelven.

Elért eredmények

Az elkészült eszköz alkalmas pozíció- és mozgásállapot-változás mérésére. Az adatok egy SD kártyán kerülnek tárolásra, melyek feldolgozása és megjelenítése egy bővíthető asztali alkalmazásban történik. A GPS-es helyzetkövetés éles környezetben, edzés közben került tesztelésre, míg a giroszkóp, gyorsulásmérő és magnetométer tesztelése az eszköz prototípus mivolta miatt korlátozott mozgástartományban történt. Ezen tesztek alapján funkcióját betölti az eszköz, a mért adatok a valóságnak megfelelnek, feldolgozásra alkalmasak.

Kulcsszavak

STM32, szenzor illesztés, GPS, IMU, teljesítménymérés, mozgáskövetés

Tartalomjegyzék

Bevezetés

Az utóbbi évtizedekben az élsportban az emberi teljesítőképesség határát feszegetjük, nüánszokon múlik egy-egy siker vagy kudarc. A régebben használt papír, toll, stopperóra, emberi megfigyelésen alapuló feljegyzéseket felváltották az egyre precízebb, automatizált mérőeszközök, az általuk rögzített hatalmas mennyiségű adat feldolgozására pedig egyre több és több módszer, algoritmus áll rendelkezésre, amelyek olyan részletes elemzést nyújtanak, olyan információkkal szolgálnak, melyek a győzelemhez szükséges különbséget jelenthetik.

A labdarúgás berkein belül nagyjából egy bő évtizede kezdett elterjedni az a gondolkodás, hogy nem csak a hagyományos edzési, taktikai módszerek vezetnek eredményre, hanem bizony az adatok gyűjtése és elemzése hatalmas előnyt adhat a riválisi csapatokkal szemben. Egyre többféle taktikai, technikai megmozdulást kezdtek számszerűsíteni, melyeknek statisztikai elemzése rávilágított számos téves berögződésre. Ezenfelül egyre nagyobb hangsúlyt kapott az egyén fizikai mutatóinak mérése és kiértékelése, melynek köszönhetően nem csak az egyes játékosok teljesítményei lettek objektívan megítélhetők, összehasonlíthatók, de hosszú távon nyomon követhetővé vált a fejlődésük, illetve a sérülések megelőzése is egyre hatékonyabb lett.

Számos játékoskövető eszköz létezik, azonban ezeknek túlnyomó többsége csak csapatok számára érhető el és magas áraik sem kedvezőek. Szerettem volna létrehozni egy olyan eszközt, melynek segítségével nyomon követhetem teljesítményemet edzéseken, mindezt minél költséghatékonyabban.

1. Technológiai és irodalmi áttekintés

1.1. Adatok a labdarúgásban

Ahogy azt már említettem, a labdarúgásban is egyre többféle adat áll rendelkezésre az elemzők számára [1]. A legalapvetőbb az az általános statisztika, melyet egy átlag foci szurkoló lát a tv-ben. Jó 30 éve csak a legnyilvánvalóbb adatokat írták fel, a gólok és a büntető lapok számát. Ma már magasan képzett elemzők követik élőben a meccseket, minden egyes megmozdulást feljegyeznek, mint többek között a passzok száma, de ezt is lebontva aszerint, hogy milyen irányú (hátra, oldalra, kulcspassz, indítás, keresztlabda stb.), párharcok, különböző típusú lövések, beadások száma, de fejlett algoritmusok már képesek például a gólok számának előrejelzésére is. Az említett statisztikai mutatókon túl számos olyan adat érhető el, mely laikus szemnek feleslegesnek tűnhetnek, azonban a sportadatelemzők számára mindegyik értékes, elemzéseikkel olyan információkkal tudnak szolgálni az edzői stábnak, melyek a győzelmet jelenthetik a következő meccsen, de segítséget nyújthatnak ígéretes fiatal tehetségek és a csapathoz leginkább illő rutinos játékosok felfedezésében is.

A taktikai elemeken túl fontos információval szolgálnak a játékosok fizikai mutatói. Ezeket két csoportra oszthatjuk [2]: az egyik a külső terhelés, ide tartozik például az összes megtett távolság, átlagsebesség, maximális sebesség, magas intenzitású futások és sprintek száma és ezekkel megtett távolság, irányváltások, ugrások száma. A másik a belső terhelés, mely ahogy a neve is sugallja, a testen belül zajló, fiziológiai folyamatokról ad információt, mint például az átlag pulzus, maximális pulzus, pulzus zónákban töltött idő, szívfrekvencia variabilitás, légzési ráta, véroxigénszint, vázizom oxigenizáció. A kétféle terhelést együtt mérve képet kaphatunk például arról, hogy egy-egy sprint vagy lövés elvégzéséhez mekkora erőkifejtés, energiafelhasználás szükséges, mekkora a relatív intenzitás. Hosszútávon nyomon követhető a sportoló regenerációs képessége, terhelhetősége, mely nagy segítség a megfelelő edzésintenzitás megtervezésében és a sérülések megelőzésében. Mindkét - külső és belső - terhelés mérésére számtalan eszköz létezik, vannak specifikusan csak egy célra tervezett termékek, mint például a pulzusmérő mellkaspántok, pulzoximéterek vagy GPS nyomkövetők, de vannak komplex eszközök, amelyek egyszerre mérnek különféle adatokat, így komplex képet adva a sportoló mozgásáról. Ilyenek például a sportórák, ám azok olyan funkciókkal is rendelkeznek, melyek labdarúgás szempontjából szükségtelenek, illetve a viselésük nem

megengedett mérkőzéseken, ezenfelül mérési frekvenciájuk sem megfelelő. De számos olyan céleszköz létezik már, mely specifikusan az edzés és mérkőzés közben nyújtott teljesítmény mérésére szolgál. Többségük az alábbi adatok gyűjtésére alkalmasak: műhold alapú vagy lokális helyzetmeghatározás, sebesség, pulzus, gyorsulások, irányváltások. Ezeket valós időben vagy utólag az eszközhöz tartozó szoftver automatikusan kielemzi, részletes statisztikákat, ábrákat, hőtérképet hoz létre és jelenít meg. A felhasználó így azonnal információhoz jut fentebb említett külső terhelési és a pulzushoz tartozó mutatókról, valamint gyártótól függően egyéb extra adatok is generálódnak, például az összes adat alapján kalkulált összterhelés (player load [3]) vagy értékelés (pro score [4]).

Mivel az először leírt, technikai és taktikai elemeket leíró adatok gyűjtése főként emberi megfigyelésen és részben digitális képfeldolgozáson alapul, ezért a terhelés típusú adatok mérésére fókuszáltam szakdolgozatom elkészítése során, azon belül is a külső terhelésre. Ugyan léteznek olyan szenzorok [5], amelyek cipőre vagy vádlihoz rögzítve viszonylag pontosan mérik a technikai elemek, mint a passzok, lövések, labdaérintések, irányváltások számát, azonban egy ilyen eszköz kivitelezése jóval bonyolultabb lenne apró méretigénye és érzékenysége miatt. Így választásom egy olyan szenzorrendszer prototípusának megtervezésére és megvalósítására esett, mellyel a külső terhelést tudom mérni.

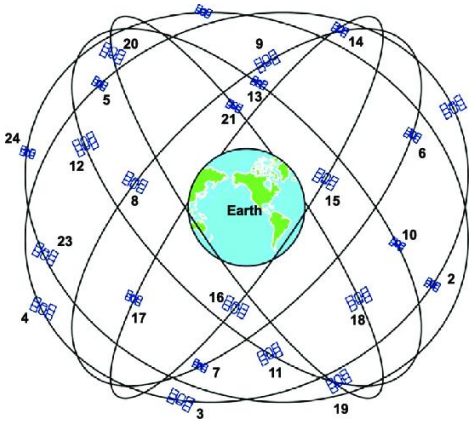
1.2. Mérési elvek

Mivel a külső terhelés mérése a cél, szükség van olyan szenzorokra, amelyek helyzetmeghatározásra és mozgásállapot-változás érzékelésére alkalmasak. Működési elveik megértése elengedhetetlen a használatukhoz.

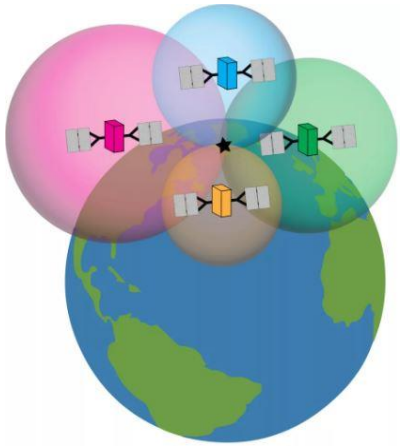
1.2.1. Helymeghatározás

A legkézenfekvőbb mód helyzetmeghatározásra a Global Positioning System [6] használata. A GPS egy 32 műholdból álló rendszer. 6 közepes távolságú pályán, 20200 km magasan mindig 24 aktív műhold kering, így a Föld egy pontján mindig legalább 6 látható, ez látható az 1.1 ábrán. A műholdak folyamatosan sugározzák a helyzetüket és az üzenetküldés pillanatának nagyon pontos, naponta többször korrigált atomórák szerinti idejét. Mivel ismert az üzenet hullámhossza, a benne szereplő idő és az üzenet vevőhöz való beérkezésének ideje közti különbségből - amíg a levegőben terjedt - kiszámítható a műhold és a vevő távolsága. A vevő

egyszerre többi műhold jelét is veszi, de legalább 4-re (3 távolság, 1 óra bizonytalanság miatt) szüksége van a pontos helyzetmeghatározáshoz: mindegyik távolság egy-egy gömb sugarát határozza meg az 1.2 ábrán látható módon, ezek metszéspontja a pozíció.



1.1 ábra *6 pályán 24 műhold* [7]



1.2 ábra *pozíció meghatározása* [8]

A műholdak két kódot továbbítanak, egy rendkívül pontos (cm nagyságrendű) jelet és egy kisebb pontosságút (kb. 2 m). Azonban előbbi titkosított, kizárólag katonai használatra érhető el, így civil célú felhasználásra meg kell elégednünk az utóbbival. Maga az üzenet a NMEA (National Marine Electronics Association) [9] vagyis a Nemzeti Tengerészeti Elektronikai Szövetség által kifejlesztett szabványnak felel meg, melyet tengerészeti elektronikai eszközök egymás közötti és egyéb számítógépekkel való kommunikációjára hoztak létre. Egy-egy NMEA mondat teljesen független a többitől. Mindig $ karakterrel kezdődik, és az úgynevezett checksum-mal, az üzenet hosszával végződik, legfeljebb 80 karakter hosszú, az elemeket vessző választja el. Részletes felépítése az 1.3 ábrán látható.

```
$GPRMC,164219.00,A,4613.74811,N,02005.26839,E,3.480,,250923,,,A*73
```

1. üzenetet küldő eszköz típus, GP=GPS
2. üzenet tartalma, RMC: pozíció és UTC adatok
3. UTC idő (hhmmss.ss)
4. státusz, A=érvényes adat
5. szélesség
6. észak (N) vagy dél (S)
7. hosszúság
8. kelet (E) vagy nyugat (W)
9. sebesség (csomóban)
10. dátum (ddmmyy)
11. üzenet hossza (checksum_A)

1.3 ábra *példa egy NMEA mondatra*

1.2.2. Mozgásállapot-változás mérése

Egy test mozgásállapot-változásának mérésére alkalmas eszközök az inerciális mérőegységek, röviden IMU (Inertial Measurement Unit) [10]. Nélkülözhetetlenek pilóta nélküli járművek irányításához, de mobil telefonokban, okosórákban, kontrollerekben, mozgáskövető rendszerekben is megtalálhatók. Gyakran alkalmazzák navigációs rendszerekben GPS kiegészítőjeként: ha gyenge a jel, vagy elveszik, például alagutakban, egy IMU segítségével továbbra is nyomon követhető és kiszámítható az eszköz pozíciója. Alapvetően egy ilyen modulba két szenzor van integrálva: a haladó mozgást érzékeli gyorsulásmérővel és a forgó mozgást giroszkóppal 3-3 tengely mentén, de gyakran tartalmaznak a giroszkóp esetleges pontatlanságainak javítása érdekében magnetométert is.

1.3. Felhasznált szenzorok

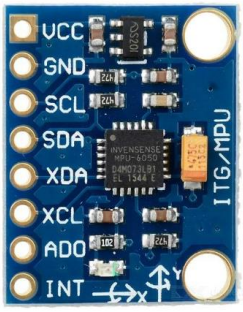
1.3.1. GPS modul

A helymeghatározáshoz egy NEO-7m [11] modult használtam (1.4 ábra). Ez egy kompakt, kis méretű, viszonylag olcsón beszerezhető GNSS (Global Navigation Satellite System) modul, ami azt jelenti, hogy több regionális műhold rendszerhez is képes kapcsolódni (GPS, GLONASS, QZSS, Galileo). Pontossága 2,5 m, 1-10 Hz közötti frekvenciával tudja frissíteni a pozícióra vonatkozó információkat, képes UART-on, SPI-on, I2C-n és USB-n keresztül is kommunikálni. A panel egy LED-et is tartalmaz, mely folyamatosan világít, ha a modul nem képes megtalálni a pozíciót, ha ez sikerült, villogni kezd.

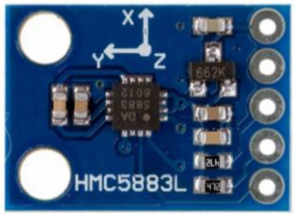
1.4 ábra *NEO-7m* [12]

1.3.2. Inerciális mérőegységek

Először egy MPU9250-es szenzort [13] választottam a mozgásváltozások érzékelésére. A nevében a 9-es azt jelöli, hogy 9 szabadságfokú: tartalmaz egy 3 tengelyű gyorsulásmérőt, egy 3 tengelyű giroszkópot és egy 3 tengelyű magnetométert. A modul 9 darab 16 bites analóg/digitális konvertert tartalmaz a gyorsulásmérő, a giroszkóp és a magnetométer x, y és z tengelyein mért adatok konvertálására. A gyorsulásmérő ±2g, ±4g, ±8g és ±16g, a giroszkóp ±250°/s, ±500°/s, ±1000°/s és ±2000°/s, a magnetométer ±4800µT tartományban képes mérni, ez konfigurálható. I2C és SPI protokollal is használható. Hosszas tesztelés után azonban másik modult kellett választanom, mivel az általam kipróbált összes MPU9250-es panel hibás volt. Teszteltem Arduino Unoval, STM32F407VGT6 development board-dal, kipróbáltam I2C-vel, SPI-jal, monitoroztam PicoScope-on a ki- és bemenő jeleket, számtalan internetes fórumot végig böngésztem, melyekből az derült ki, hogy az MPU9250-as modulok jelentős része hibásan kerül forgalomba. Így nem bíztam a szerencsében, hogy találok-e működőt, hanem inkább áttértem egy MPU6500-as modul (1.5 ábra) használatára. Ez annyiban tér el az MPU9250-estől, hogy nem tartalmaz magnetométert, csak gyorsulásmérőt és giroszkópot. Viszont ezek teljesen ugyanolyanok, ugyanazokkal a regisztercímekkel rendelkeznek, így ugyanazzal a kóddal működtethető, amelyet az MPU9250-eshez készítettem. Hátránya, hogy egy különálló magnetométerre is szükség van. A végleges verzióba egy HMC5883L típusú magnetométer [14] (1.6 ábra) került. Ez 3 tengely mentén méri a mágneses teret és egy 12 bites ADC-nek köszönhetően 1-2°-ok pontosságú adatokkal szolgál. Kizárólag I2C-n keresztül kommunikál.



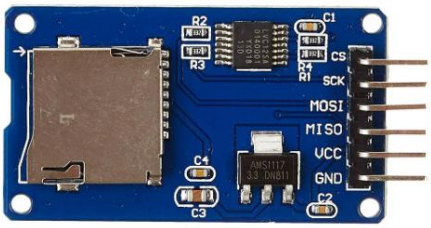
1.5 ábra *MPU6500* [15]



1.6 ábra *HMC5883L* [16]

1.4. Adattárolás

A szenzorok által mért adatok tárolásának számos módja van. A lehetőségek közé tartozik a vezetékes vagy vezeték nélküli, valós idejű vagy utólagos továbbítás. Elegendőnek tartottam, ha utólag dolgozom fel az adatokat, nem szükséges azonnal, valamilyen vezeték nélküli hálózaton keresztül továbbítani azokat. Így a legegyszerűbb és legolcsóbb megoldást, az SD kártyára mentést választottam. Ehhez szükség van egy SD kártya írására és olvasására képes modulra [17] (1.7 ábra).



1.7 ábra *SD kártya modul* [18]

1.4.1. FatFs

Az SD kártyán való adattároláshoz létre kell hozni egy fájlrendszert. Erre alkalmas a FAT [19] (file allocation table), ami egy szinte minden operációs rendszerrel kompatibilis fájlrendszertípus. Ezt implementálja a FatFs [20], amely egy kifejezetten beágyazott rendszerekhez fejlesztett fájlkezelést megvalósító függvény könyvtár. Teljesen platform független, minden C nyelv futtatására képes rendszeren alkalmazható. Nem függ az adathordozó típusától sem, mivel csak egy köztes réteget (media access interface) biztosít a szoftver és a hardver között. Ennek viszont az is a következménye, hogy magát az eszközzel való kommunikációt külön implementálni kell. Ezért az SD kártya használatához szükség van egy library-re [21], amely a FatFs SPI-jal történő adattovábbítását valósítja meg.

1.4.2. JSON

A GPS adatok tárolási formája a NMEA szabvány miatt adott, viszont a többi szenzor által mért adatok valamilyen struktúrába rendezése szükségszerű az adatfeldolgozás sikerének érdekében. Erre egy alkalmas megoldás a JSON formátum [22]. A JSON (JavaScript Object Notation) egy számítógépek közti adatcserére megalkotott szabvány, mely eredetileg JavaScirpt nyelvhez készült, de ma már széles körben elfogadott, platformfüggetlen formátum. Tulajdonképpen egy

adatszerkezetet valósít meg, melynek kétféle felépítése lehet: kulcs-érték párokból álló halmaz vagy értékek rendezett listája. Tárolhat számot, egyszerű szöveget (string-et), logikai értéket, tömböt vagy objektumot. Egyszerűen előállítható és feldolgozható, lehetővé teszi különböző nyelvű környezetek közti adatcserét, ezért egy rendkívül elterjedt adatformátum. A mikrovezérlőkön is használt C nyelvben a cJSON függvény könyvtár [23] valósítja meg.

1.5. Mikrovezérlők

A mikrovezérlők olyan kisméretű célszámítógépek. melyek esetében egyetlen lapkára vannak integrálva a komponensek, többek között a processzor, memória, adatbuszok, kommunikációs áramkörök, analóg és digitális perifériák. Kis méretük, alacsony fogyasztásuk, nagy sebességük és megbízhatóságuk miatt széles körben elterjedtek: járművekben, háztartási gépekben, okos eszközökben, szinte minden intelligens rendszerben megtalálhatóak.

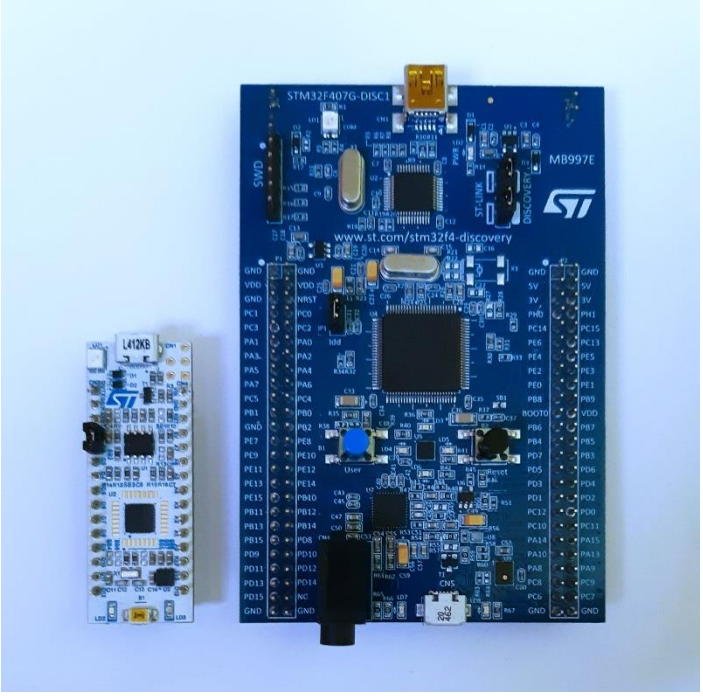
1.5.1. Felhasznált mikrovezérlők

A kiválasztott szenzorok vezérléséhez egy olyan mikrovezérlőt kellett találni, amely rendelkezik a megfelelő mennyiségű és típusú kommunikációs interfésszel, implementálható rajta a FatFs csomag és fontos a kis méret is.

Az egyszerűbb és gyorsabb fejlesztés érdekében gyakran integrálják a mikrovezérlő chipeket egy úgy nevezett fejlesztői panelre, mely egy olyan integrált áramkör, ami többek között tartalmaz tápellátást és órajelet biztosító áramköröket, GPIO csatlakozókat és programozói interfészt az egyszerű fejlesztés és tesztelés érdekében.

Elsőkörben a GPS modul teszteléséhez egy STM32F407VG-es mikrovezérlőt tartalmazó, STM32F407G-DISC1 [24] fejlesztői panelt alkalmaztam, mivel korábban már használtam ezt a típust. Az STM32F407VG egy 32-bites Arm Cortex-M4 RISC mikroprocesszoron alapuló, akár 168 MHz órajel frekvenciára is képes chip. A szakdolgozat szempontjából fontos tulajdonsága, hogy támogatja az UART, SPI és I2C kommunikációkat is. A fejlesztői panel számos programozást és tesztelést segítő komponenst tartalmaz, melyek közül kiemelendő a szakdolgozatban is használt debuggolást biztosító ST-Link, a beépített nyomógomb és LED-ek. A végleges verzióhoz egy lényegesen kisebb, egy STM32L412KB mikrovezérlőt tartalmazó NUCLEO-L412KB [25] fejlesztői panelt használtam. Ez szintén 32-bites Arm Cortex-M4 RISC mikroprocesszor alapú, nagyon alacsony fogyasztásra optimalizált, maximális órajel frekvenciája 80 MHz. A panel részét képezi az ST-Link és egy LED, viszont

nyomógomb nem áll rendelkezésre. Az 1.8 ábrán jól látható a bal oldali NUCLEO-L412KB és a jobb oldali STMF407G-DISC1 közötti méretbeli különbség.



1.8 ábra *NUCLEO-L412KB és STM32F407G-DISC1 fejlesztői panelek*

1.5.2. STM32CubeIDE

A mikrovezérlő programozásához az STM32CubeIDE-t [26] használtam, mely egy STM32-es mikrovezérlők programozásához készített fejlesztői környezet. Számos előnye közé tartozik például az egyszerű debuggolási lehetőség, git verziókezelő csatlakoztatása és a grafikus konfigurációs felület, melynek segítségével néhány kattintással elvégezhető az mikrovezérlő konfigurálása, majd automatikusan le is generálja az ezeket megvalósító kódot, létrehozza a projekt fájlrendszerét.

1.5.3. HAL függvény könyvtár

Mikrovezérlők programozásához létezik egy nagyon hasznos könyvtár, a Hardware Abstraction Layer, röviden HAL. Ez egy köztes szoftverréteg a hardver és a felhasználó között, célja, hogy leegyszerűsítse és lerövidítse hardver programozását, akár a hardver mélyreható ismerete nélkül, egy egyszerű, általános függvényhívás is használható a különböző perifériák kezelésére, nincs szükség bonyolult, regiszter szintű programozásra.

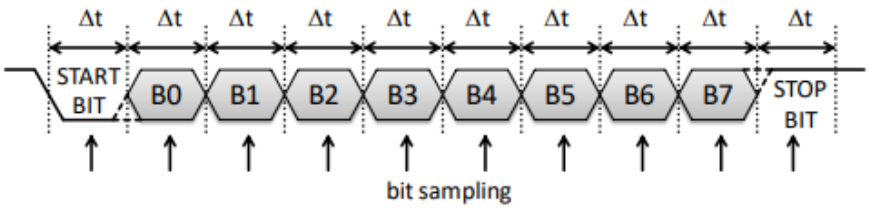
A HAL könyvtár részét képezik a mikrovezérlő perifériáinak kezelésére szolgáló driverek, így a kommunikációs interfészek használatához szükségesek is. Ezek tartalmaznak egy inicializációs és egy handle változókból álló adatstruktúrát. STM32CubeIDE használata esetén ezek értékeit az IDE automatikusan inicializálja a grafikus felületen megadott beállításoknak megfelelően. Ezen adatok alapján az interfész inicializálása is megtörténik, így csak a kommunikációt megvalósító, adat küldést és fogadást kezelő függvények meghívása a feladat, mely történhet polling vagy interrupt módban. Interrupt módban lehetőség van implementálni egy callback függvényt, melyet az adatáramlás befejeztével hív meg az interrupt kezelő rutin, ezzel megszakítva a főprogram futását. Az itt szereplő kód lefut, amely általában egy flag értékének megváltoztatása, majd folytatódik főprogram.

1.6. Kommunikáció

Az eszköz alkotóelemei között – mikrovezérlő, GPS modul, IMU modulok, SD kártya - biztosítani kell az adatok áramlását. Ez többféle kommunikációs csatornán zajlik, melyek vezérléséért a mikrovezérlő felel [27], [28].

1.6.1. UART

Az UART (universal asynchronous receiver/transmitter), ahogy a nevében is szerepel, aszinkron küld és fogad adatokat, nincsen szinkronizáló órajel a két kommunikáló eszköz között. Az adatátvitel kezdetét egy start bittel jelzi a küldő fél, majd sorban elküldi a biteket úgy, hogy mindegyik Δt ideig van a vezetéken. A fogadó fél a start bit detektálása után Δt időközönként hajt végre mintavételezést, egészen egy stop bit érzékéséig, ahogy az 1.9 ábrán is látható. Ezért fontos, hogy a két fél azonos időzítéssel rendelkezzen. Ezt az átviteli sebességgel vagy más néven baud rate-tel adhatjuk meg, ami nem más, mint az említett Δt reciproka. Az összekapcsolt két eszköz „egyenrangú”, mindkettő tud küldeni és fogadni is adatot, az egyik eszköz továbbításra használt kimenetét (TX) a másik fogadásra használt kivezetésével (RX) kell összekötni. Simplex módban az egyik eszköz csak továbbít, a másik csak fogad, half-duplex esetben mindkettő továbbít és fogad is, de nem egyidejűleg, full-duplex esetben egyszerre megvalósulhat a továbbítás és a fogadás is.



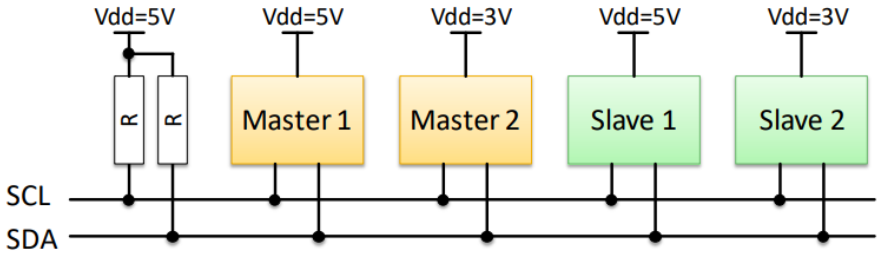
1.9 ábra *UART adatküldés idődiagramja* [27]

1.6.2. SPI

Az SPI (serial peripheral interface) alapvetően integrált áramkörök összekapcsolására alkalmas. 4 vezeték szükséges a sikeres adattovábbításhoz: az SCK az órajelet biztosítja ahhoz, hogy az eszközök tökéletes szinkronban legyenek, a MOSI (master in slave out) a master-ből (a mikrovezérlő) a slave-hez (például egy szenzor) kimenő adat továbbítására, a MISO (master in slave out) a master-hez beérkező adatok továbbítására szolgál. Lehetőség van egy master-hez több slave csatlakoztatására is, ezért van szükség a negyedik, SS (slave select) vezetékre, mellyel kiválasztható a megfelelő slave.

1.6.3. I2C

Az I2C (inter-integrated circuit) hasonlóan az SPI-hoz szinkron adattovábbítást valósít meg, viszont csak két vezetéken: SCL a megfelelő órajelet biztosítja, az SDA vonalon pedig a kétirányú adatátvitel zajlik. I2C használata esetén egy adatbuszon több master és több slave is osztozhat az 1.10 ábrán látható módon.



1.10 ábra *Több eszközt összekötő buszrendszer* [27]

1.6.4. DMA

A DMA (direct memory access vagy közvetlen memória hozzáférés) [29] egy nagysebességű buszrendszer, mely memória területeket és perifériákat köt össze. A processzor beavatkozása

nélkül mehet végbe az adatok mozgatása, így az közben más hasznos feladatokat tud végezni. Gyakran alkalmazzák ezt a megoldást nagy mennyiségű adat nagy sebességű kezelése érdekében. Szakdolgozatomban a GPS modul és a mikrovezérlő közötti UART kommunikációhoz használtam DMA-t azzal a céllal, hogy a folyamatosan érkező NMEA üzenetek olvasása közben a processzor más feladatokat lásson el, mint például a memóriakártyára írást vagy a többi szenzor kezelését.

2. Implementáció

2.1. Mikrovezérlő program

2.1.1. Szenzor driverek

A szenzorok működtetéséhez szükséges kódokat külön könyvtárakba írtam, így a főprogram minimális változtatása mellett tetszés szerint változtatható a felhasznált szenzorok száma. Tulajdonképpen az alkalmazástól független drivereket hoztam létre, amelyek más projektekben is könnyen integrálhatók, újra felhasználhatók.

Ugyan létezik számos működő driver az általam is használt szenzorokhoz, főleg az MPU6050-hez, de ezek olyan funkciókat is megvalósítanak, melyekre ebben a projektben nincs szükség. Ezenkívül ezeknek a drivereknek a forrása nem minden esetben megbízható, a tesztelésük gyakran hiányos, a kód pedig kommentek és clean code alkalmazásának hiánya miatt nehezen értelmezhető, javítani az esetleges hibákat, amiért nem az elvárásoknak megfelelően működik, nehezebb feladat lenne, mint írni egy teljesen új függvény könyvtárat.

Alapvetően mindhárom - NEO-7M, MPU6050, HMC5883L - driver két fájlból áll: egy .h header fájlból és egy ugyanazt a nevet viselő .c source fájlból. Az előbbi tartalmazza a szükséges konstansokat, struktúrákat és függvény deklarációkat, tulajdonképpen egy interfészről van szó. A source fájl tartalmazza a függvények megvalósítását. Mindegyikben szerepel egy inicializáló függvény, melyben a szenzor működéséhez szükséges konfigurációs lépések találhatók és egy, a mért adatok kiolvasására alkalmas függvényt.

mpu6050.h és ***mpu6050.c***

Az *mpu6050.h* header fájlban található az összes konstans, amire az MPU6050 modul működtetéséhez szükség van: *MPU6050_ADDRESS* a modul címe, *USER_CONTROL, ACCEL_CONFIG, GYRO_CONFIG, PWR_MGMT* az inicializáláshoz szükséges regiszter címek, *ACCEL_XOUT_H* a gyorsulásmérő által az x tengely mentén mért gyorsulás felső bájtját tároló regiszter címe. Az *A_G_DATASIZE* a beolvasni kívánt bájtok száma, az adatok olvasásánál van szerepe. Ezenkívül van még 4-4 konstans a gyorsulásmérő és a giroszkóp mérési tartományának beállításához, illetve 4-4 az adott tartományhoz tartozó szám, amellyel leosztva a beolvasott digitális értéket, megkapjuk a konkrét gyorsulást/szögsebességet. Utóbbi

tulajdonképpen az érzékenységet határozza meg. A kényelmesebb kezelhetőség és átláthatóság kedvéért globális változók helyett egy struktúrát definiáltam: létrehoztam a 2.1 ábrán látható *MPU6050_t* típust, melyet további három alstruktúrára osztottam aszerint, hogy milyen szerepet látnak el az adott alstruktúrába tartozó változók. Az első csoport a *rawData*, amely a szenzor által x, y és z tengely mentén mért gyorsulásnak és szögsebességnek megfelelő 16 bites egész számot tartalmazza, a *sensorData* a már konvertált, valóságnak megfelelő értékeket tartalmazza, a *config* a mérési tartomány és az érzékenység értékét tartalmazzák.

```
typedef struct
{
    struct
    {
        int16_t ax, ay, az, gx, gy, gz;
    } rawData;

    struct
    {
        float ax, ay, az, gx, gy, gz;
    } sensorData;

    struct
    {
        uint8_t aRange, gRange;
        uint16_t aMaxRange, gMaxRange;
    } config;
} MPU6050_t;
```

2.1 ábra *MPU6050_t struktúra.*

Az *mpu6050.c* source fájlban található a header fájlban deklarált öt függvény megvalósítása. Az első az *MPU6050_Init()* inicializáló függvény, melyben először alaphelyzetbe, mérésre kész állapotba lesz állítva a szenzor, beállításra kerül a kívánt mérési tartomány, majd az *MPU6050_SetGyroRange()* és *MPU6050_SetAccRange()* függvényekkel az *mpu->config.gMaxRange* és *mpu->config.aMaxRange* változók is meghatározásra kerülnek. A konfigurációs adatok küldése a *HAL_I2C_Mem_Write()* függvény meghívásával lehetséges, mely paraméterben várja a felhasznált I2C csatornához tartozó handle struktúra címét, a regiszter címét, amelyet szeretnénk elérni, a beírni kívánt számot, ennek méretét (1) és egy időkorlátot, (*HAL_MAX_Delay*), melynek az a szerepe, hogy sikertelen adatküldés esetén ennyi ideig kísérelje meg ismételten a küldést a master. A mért adatok beolvasásáért az *MPU6050_GetRawData()* felel. A szenzor 14 bájton tárolja az adatokat, melyek 16 bitesek, így minden tengelyhez - x, y, z irányú gyorsulás és x, y, z irányú szögsebesség - egy felső és egy alsó bájt tartozik. Ez 12 bájt, a maradék kettőben a modul lapjának hőmérséklete van. Az *ACCEL_XOUT_H* című (0x3B) regisztertől kezdődően egymást követő című regiszterekben vannak az adatok, így elegendő egy *HAL_I2C_Mem_Read()* függvényhívást végrehajtani,

melynek paraméterben meg kell adni a handle struktúra címét, a slave, vagyis a szenzor címét eggyel balra shiftelve, az adatokat tároló regiszterek közül az első címét (*ACCEL_XOUT_H*) , ennek méretét (1), a beolvasott adatok tárolására szolgáló tömböt (*acc_gyro*), hogy hány bájtot (14) szeretnék kiolvasni és végül egy időkorlátot (*HAL_MAX_Delay*). Ezt követően a felső és alsó bájtok egy darab, 16 bites, előjeles változóba kerülnek shiftelés segítségével a 2.2 ábrán látható módon.

```
void MPU6050_GetRawData(I2C_HandleTypeDef *I2Cx, MPU6050_t *mpu)
{
    uint8_t address=MPU6050_ADDRESS << 1;
    int8_t acc_gyro[A_G_DATA_SIZE];
    //read data
    HAL_I2C_Mem_Read(I2Cx, address, ACCEL_XOUT_H, 1, acc_gyro, A_G_DATA_SIZE, HAL_MAX_DELAY);
    //convert high and low byte into a 16 bit variable
    mpu->rawData.ax = (int16_t)acc_gyro[0] << 8 | acc_gyro[1];
    mpu->rawData.ay = (int16_t)acc_gyro[2] << 8 | acc_gyro[3];
    mpu->rawData.az = (int16_t)acc_gyro[4] << 8 | acc_gyro[5];
    mpu->rawData.gx = (int16_t)acc_gyro[8] << 8 | acc_gyro[9];
    mpu->rawData.gy = (int16_t)acc_gyro[10] << 8 | acc_gyro[11];
    mpu->rawData.gz = (int16_t)acc_gyro[12] << 8 | acc_gyro[13];
}
```

2.2 ábra *MPU6050 által mért adatok kiolvasása*

Az *MPU6050_GetSensorData()* függvény a beolvasott adatok valós gyorsulás és szögsebesség értékké való konverzióját végzi, a *mpu->sensorData* alstruktúrába menti ezeket. Ez végül nem lett felhasználva a szakdolgozatomban, kizárólag a nyers adatok kerülnek mentésre, ugyanakkor mivel egy újra felhasználható driverről van szó, más alkalmazásokban hasznos lehet ennek megléte.

hmc5883l.h és ***hmc5883l.c***

A magnetométer modul drivere nagyon hasonló felépítésű, mint az előbb bemutatott IMU driver. A *hmc5883l.h* fájlban definiált konstansok közül a *HMC_WRITE* és *HMC_READ* az eszköz címe írás, illetve olvasás esetén, a *HMC_CONFIG_A*, *HMC_CONFIG_B* és *HMC_MODE* konfigurációhoz szükséges regiszter címek, a *HMC_GAIN*_x konstansok a szenzor érzékenységét meghatározó számok: magasabb “gain” kisebb változást képes érzékelni, ugyanakkor a mérési tartomány kisebb lesz. Ugyanúgy, mint az MPU6050-nél, itt is tengelyenként (x, y, z) 2 darab - alsó és felső - bájton van tárolva a mért érték, egymást követő című regiszterekben, így elegendő az első regiszter címét és az olvasni kívánt bájtok számát ismerni: *HMC_X_DATA_H* és *BUFFER_SIZE*. A HMC5883L modulnak három féle üzemmódja lehet: egyszeri mérés, várakozó állapot és folyamatos mérés. Utóbbi esetén állítható a mérési frekvencia, a *HMC_SAMPLE_RATE_x* konstansok által definiált értékekre. Itt is egy struktúrát hoztam létre az adatok tárolására, a *HMC5883L_t* adattípust, amely a három tengely

mentén mért adatokat 16 bites integerekben tárolja: *x_data, y_data, z_data*, és 8 bites integerekben a szenzor beállításait: *sample_rate, gain, mode*. A *hmc5883l.c* source fájlban az *mpu6050.c*-hez hasonlóan van egy inicializáló függvény, a *HMC5883L_Init(),* ebben kerül beállításra a kívánt mérési mód, érzékenység, folytonos mérés esetén a frekvencia. A *HMC5883L_ReadData()* felel a szenzor által mért adatok kiolvasásáért és ugyanúgy mint az *MPU6050_GetRawData()* függvénynél, itt is konvertálásra kerül a két-két 8 bites érték egy-egy 16 bites változóvá (*hmc.x_data, hmc.y_data, hmc.z_data*). Az I2C kommunikáció megvalósítása megegyezik az mpu6050 drivernél leírtakkal: a *HAL_I2C_Mem_Write()* és a *HAL_I2C_Mem_Read()* függvények meghívásával indítható el az adatáramlás, a megfelelő paraméterek megadása mellett.

neo7m_gps.h és neo7m_gps.c

A NEO-7m GPS modulhoz készített driver felépítését tekintve szintén hasonlít az előzőekhez: a header fájlban definiáltam a konstansokat, létrehoztam egy *GPS_t* típusú struktúrát, mely a mérési frekvenciát (*measurement_rate*), a (később leírt) felesleges NMEA üzenetek engedélyezését vagy tiltását megadó változót (*recommended_min_info*), a beérkező üzenetet aktuális karakterét (*gps_data*) és egy új karakter érkezését jelző változót (*rx_cplt*) tartalmaz. Több dologban viszont eltér a korábban látott sémától. Egyrészt a modul UART-ot használ kommunikációhoz, ráadásul ezt DMA-val valósítottam meg, így ennél a korábban látott polling mód helyett interrupt módban történik a mikrovezérlő és a modul közötti kommunikáció. Másik nagy különbség a konfigurációs üzenetek formája: nem csak egy-egy bájtot kell küldeni, hanem UBX formátumnak megfelelő üzeneteket kell megadni a modulnak. Egy ilyen üzenet alapvetően az alábbi részekből áll:

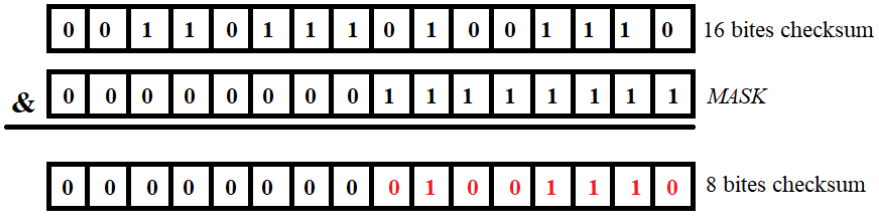
header: 2 bájt; class: 1 bájt; id: 1 bájt; hossz: 2 bájt; payload; ck_a: 1 bájt; ck_b: 1 bájt

A header a NEO-7m modul esetén mindig 0x5B és 0x62, a class és az id azt határozza meg, hogy mit szeretnénk beállítani, a hossz az payload hossza, a payload tartalmazza a beállítandó értékeket. A ck_a és a ck_b az úgynevezett checksum, melyet minden üzenethez ki kell számolni az 2.3 ábrán látható algoritmus segítségével.

```
void Calc_checksum(uint8_t *message, uint8_t arraysize)
{
    uint8_t checksum_A=0u;
    uint8_t checksum_B=0u;
    uint8_t index;
    for(index=2; index<arraysize-2;index++)
    {
        checksum_A=(checksum_A+message[index]) & MASK;
        checksum_B=(checksum_A+checksum_B) & MASK;
    }
    message[arraysize-2]=checksum_A;
    message[arraysize-1]=checksum_B;
}
```

2.3 ábra *Checksum kiszámítása*

A függvényben egy ciklus a header részt kivéve végighalad az üzenetet tároló tömbön és minden iterációban az aktuális bájttal növeli a *checksum_A* értékét, majd ezzel a számmal növeli a *checksum_B*-t. Mivel ezek az összegek hamar túllépik a 8 bites nagyságot, szükség van egy maszkolásra is. Ennek során A 2.4 ábrán látható módon egy bitenkénti *ÉS* művelet történik a 16 bites érték és a *MASK* (0xFF) konstans között, vagyis csak az alsó 8 bit marad meg, így megőrizve a 8 bites méretet.



2.4 ábra *16 bites szám maszkolása*

A GPS modul többféle típusú NMEA üzenetet küld, mint például a látható műholdak száma ($GPGSV kezdetű), pozíció dátum nélkül ($GPGLL), mód és aktív műholdak száma ($GPGSA), GPS fix adat ($GPGGA), sebesség ($GPVGT) és GPS fix adat dátummal és mérés idejével ($GPRMC). A mozgás útvonalának követéséhez a legutóbbi elegendő információval szolgál, így a többi üzenet küldését érdemes letiltani, csökkentve ezzel a tárhelyigényt. Az ezek letiltásához szükséges UBX üzeneteket a *cfg_msg* konstans tömbben találhatóak. Az inicializáló függvény *measurement_rate* paraméterével adható meg, hogy milyen időközönként mérje az eszköz a pozíciót. Ez az érték a header fájlban megadott *MEASUREMENT_RATE_***x***MS* konstansok valamelyike lehet. Mivel **x**=1000 és **x**=500 esetén ez egy 16 bites szám, ezért itt is szükség van a korábban látott maszkolásra: egy felső és egy alsó bájtra bontást követően bekerülnek a mérési ráta konfigurációjához szükséges UBX üzenetet tároló *cfg_rate* tömb megfelelő pozíciójába. Ezután a Calc_checksum() függvényben

kiszámításra kerülnek a checksum értékek és *cfg_rate* tömb utolsó két helyére kerülnek, így már teljes lesz az UBX üzenet. Először ez lesz továbbítva a modulnak a *HAL_UART_Transmit_DMA_IT()* függvény segítségével, melynek paraméterben kell megadni a használt UART csatornához tartozó handle struktúra címét, az üzenetet és annak hosszát. Ha a továbbítás sikeres, lefut a *HAL_UART_TxCpltCallback()* függvény, melyben *tx_cplt* értéke 1 lesz. Ennek hatására - amennyiben a paraméterben kapott *recom_min_info* értéke 1 - a *cfg_msg* tömb első eleme továbbításra kerül, illetve a *tx_cplt*-t ismét 0 lesz. Így halad végig a tömböm, amint sikeresen befejeződött az előző küldése, megkezdődik a következő küldése. Új adat fogadásának elindítása a *HAL_UART_Receive_DMA_IT()* függvénnyel történik, paramétere a handle struktúra, a *gps_data* változó, amely a beolvasott adatot fogja tárolni és ennek mérete, vagyis 1. Új adat érkezésekor lefut a *HAL_UART_RxCpltCallback()* függvény, melyben az *rx_cplt* változó értéke 1 lesz, ennek a főprogramban lesz szerepe.

2.1.2. Főprogram

A főprogram alapját az IDE által generált *main.c* fájl alkotja. Ez a váz tartalmazza a kommunikációs interfészek használatához szükséges globális handle változókat (*hi2c1, hi2c3, hspi1, huart2, hdma_usart2_rx, hdma_usart2_tx*), a felhasznált perifériák (GPIO, TIM6, I2C1, I2C3, SPI1, UART2, DMA) és a HAL inicializáló függvényét és az órajel konfigurációt. Ezenfelül lehetőség van saját hibakezelés és tesztelés implementálására a fájl végén, direkt erre a célra generált kódrészben. Minden más, vagyis a *USER CODE BEGIN* és *USER CODE END* kommentek közötti részek saját kódok. A további változókat a 2.1 táblázat tartalmazza.

típus	név	funkció
FATFS	fs	FATFS struktúra a fájlkezeléshez
FIL	gps_file	GPS adatokat tároló fájl
FIL	imu_file	IMU adatokat tároló fájl
FRESULT	fres	fájlkezelési függvények visszatérési értéke
MPU6050_t	mpu	MPU6050 struktúra
HMC5883L_t	hmc	HMC5883L struktúra
uint8_t[*NAME_SIZE*]	file_name_gps	GPS adatokat tároló fájl neve
uint8_t[*NAME_SIZE*]	file_name_imu	IMU adatokat tároló fájl neve
uint8_t	file_name_index	fájlnevek utolsó karaktere, értéke: 1-9
uint8_t	session_end	mérés végét jelző flag
uint8_t	time_elapsed	IMU mérési frekvenciájának lejártát jelzi
uint8_t	time_counter	TIM6 reset (100ms) esetén inkrementálódik
uint8_t	imu_sampling_rate	imu_sampling_rate*time_counter=imu mérési időköz

2.1 táblázat *főprogramban használt globális változók*

A program elején az inicializáció történik. Első a HAL és a perifériák alapbeállítása, hiszen ezek nélkül a további komponensek nem is működnének. Majd a kívánt mérési tartomány, mód és mérési frekvencia megadása után a szenzorok "init" függvényei is lefutnak. Ezt követően létre kell hozni a két fájlt, amelyekbe a szenzorok adatai fognak kerülni. Hogy ne kelljen minden egyes mérés végeztével átmásolni ezeket a számítógépre, a fájlok neveinek végén van egy szám is, melynek értéke 1-9 lehet, ezzel lehetőség van egyszerre 9 mérés tárolására a memóriakártyán. A megfelelő név megtalálása a *Find_file_name()* függvényben (2.5 ábra) történik, melyben minden hívásra nő a *file_name_index* értéke, az *snprintf()* függvény segítségével a „gps***n***.txt" és az „imu***n***.txt" stringek 3. pozíciójába, az ***n*** helyére kerül. Az így kapott fájlnevekkel történik az *f_open()* fájlt megnyitó függvény meghívása. Amennyiben ez sikertelen, mert már létezik ilyen nevű, akkor a függvény *FR_EXIST* értékkel tér vissza és újra meghívódik, míg meg nem találja a megfelelő számot. Abban az esetben, ha elfogytak a lehetséges nevek vagy már csak kevesebb, mint két lehetőség van, világítani kezd a mikrovezérlőbe beépített LED.

```
FRESULT Find_file_name()
{
   if(file_name_index>7)
   {
      HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3, SET);
   }
   file_name_index++;
   if(file_name_index<10)
   {
       snprintf(file_name_gps, NAME_SIZE, "gps%d.txt", file_name_index);
       snprintf(file_name_imu, NAME_SIZE, "imu%d.txt", file_name_index);
       f_mount(&fs, "", 0);
       fres=f_open(&gps_file, file_name_gps, FA_CREATE_NEW | FA_WRITE);
       if(fres!=FR_OK) return fres;
       fres=f_open(&imu_file, file_name_imu, FA_CREATE_NEW | FA_WRITE);
       if(fres!=FR_OK) return fres;
   }
  return fres;
}
```

2.5 ábra *Fájlnevet kiválasztó függvény*

A megnyitást követően az imu*n*.txt fájlba kiíratásra kerülnek az MPU6050 és a HMC5883L szenzorok beállításait (a 2.6 ábrán látható JSON formátumban), hiszen az adatfeldolgozásnál fontos információ, hogy milyen tartomány, milyen érzékenység mellett történtek a mérések.

```
char* JSON_Header()
{
    cJSON *json = cJSON_CreateObject();
    cJSON *config=cJSON_CreateObject();
    cJSON_AddItemToObject(json, "config", config);
    cJSON_AddNumberToObject(config, "gyro_range", mpu.config.gMaxRange);
    cJSON_AddNumberToObject(config, "acc_range", mpu.config.aMaxRange);
    cJSON_AddNumberToObject(config, "sample_rate", imu_sampling_rate);
    char *json_str=cJSON_Print(json);
    cJSON_Delete(json);
    return json_str;
}
```

2.6 ábra *szenzorok beállításait JSON-re konvertáló függvény*

Ezután kezdődik a mérés, lefutnak a driverekben szereplő, az adatok olvasásáért felelős függvények (*MPU6050_GetRawData(), HM588L_ReadData(), GPS_Receive()*) és elindul a timer (*HAL_TIM_Base_Start_IT()*). A timer 100 milliszekundumonként generál megszakítást, ekkor lefut a *HAL_TIM_PeriodElapsedCallback()* függvény, melyben eggyel nő a *time_counter* változó értéke. Amikor ez megegyezik az *imu_sampling*_rate értékével, vagyis eltelt két mérés közötti idő, akkor a *time_counter* nulla lesz, a *time_elapsed* értéke pedig 1. Ekkor a fő ciklus belép az *if(time_elapsed){}* ágba, meghívja a mért adatokat JSON formátumba konvertáló *Convert_to_JSON()* függvényt, majd a kapott stringet kimenti az SD kártyára. Ezt követően felszabadítja a stringet tároló memória területet, újból elindítja a méréseket és törli a *time_elapsed* flag-et. Eközben a GPS modul folyamatosan küldi a NMEA mondatokat, minden karakter érkezésekor a driverben implementált *HAL_UART_RxCpltCallback()* függvény a

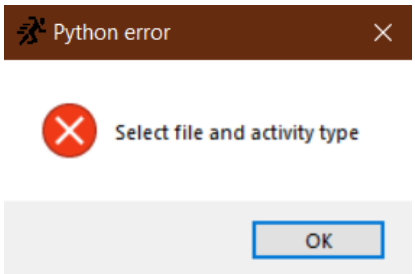
gps.rx_cplt változót 1-re állítja, ezzel jelezve a főprogramnak, hogy elérhető egy új karakter. Ekkor a fő program kimenti a memóriakártyára az aktuális karaktert, törli a *gps.rx_cplt* flag-et és folytatja az adatok fogadását. A harmadik változó, amit a fő ciklus figyel, az a mérés végét jelző, *session_end* változó. A felhasználó az edzés befejezését jelezve megnyomja a rendszerhez kapcsolt nyomógombot, amely egy megszakítást generál: lefut a *HAL_GPIO_EXTI_Callback()* függvény, melyben a *session_end* értéke 1 lesz. Ennek hatására a főciklusban megtörténik a fájlok lezárása.

2.2. Python program

A mérések eredményeinek megjelenítésére létrehoztam egy egyszerű, bővíthető asztali alkalmazást. A python nyelv mellett esett a választásom, mert némi adatfeldolgozásra is szükség van, ehhez pedig számos modul elérhető, melyekkel ez a folyamat gyorsan és egyszerűen megvalósítható. Az alábbi függvénykönyvtárakat használtam fel: *tkinter* a grafikus felhasználói felület létrehozásához, *pandas* a fájlból készült adattábla kezeléséhez, *datetime* idő és dátum kezeléshez, *numpy* különböző számolásokhoz, *geopy* a koordináták kezeléséhez, matplotlib grafikon létrehozáshoz, *tkintermapview* az útvonal megjelenítéséhez.

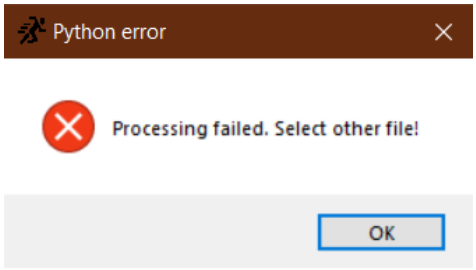
2.2.1. Felhasználói felület

Az alkalmazást elindítva először ki kell választanunk egy txt vagy csv kiterjesztésű, GPS modul által mért adatokat tartalmazó fájlt (*Select GPS log*), majd választhatunk tevékenység típust egy legördülő menüben (*Select activity type*): labdarúgást, futást vagy kerékpározást, ezután a *Show activity* gombra kattintva megjelennek az edzés adatai. Amennyiben nem választottunk fájlt vagy sportot akkor 2.7 ábrán látható hibaüzenetet tartalmazó ablak ugrik fel.



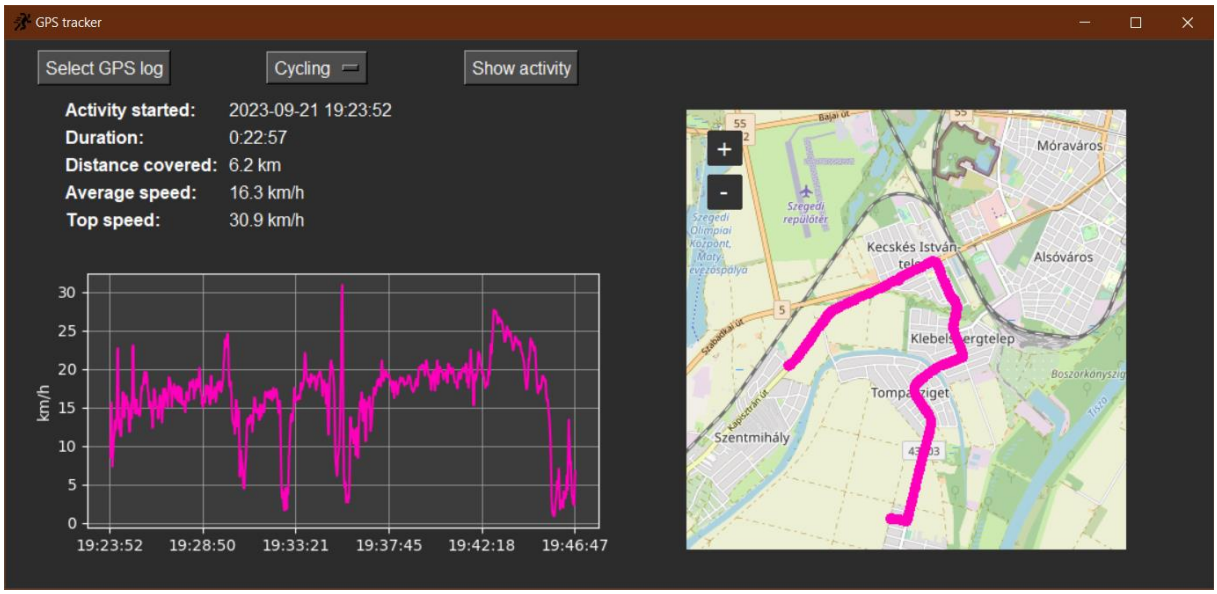
2.7 ábra *Hibaüzenet fájl vagy tevékenység típus kiválasztásának hiánya esetén*

Ha nem megfelelő a fájl formátuma, akkor a 2.8 ábrán látható figyelmeztető ablak ugrik fel.



2.8 ábra *Hibás formátumú fájlra figyelmeztető ablak*

Ha sikeres a fájl feldolgozása, megjelennek az edzés adatai: kezdési dátum és időpont, eltelt idő, megtett távolság, átlagsebesség, maximális sebesség. Alatta látható a sebesség változása egy grafikonon, az ablak jobb oldalán pedig egy térkép és rajta az útvonal található. A különbség a három választható tevékenység között az átlagsebesség mértékegysége. Labdarúgás esetén a legelterjedtebb megadási forma a percenként megtett méterek száma, futás esetén az egy kilométer megtételéhez szükséges idő, kerékpározásnál pedig a hagyományos egy óra alatt megtett kilométerek száma. Ha másik edzést szeretnénk megtekinteni, egyszerűen kiválasztunk egy másik fájlt, a típusát és a *Show activity* gombra kattintás után frissül az ablak, már az új adatokat mutatja a 2.9 ábrán látható módon.



2.9 ábra *Képernyőkép az alkalmazásról*

2.2.2. GPS adatok feldolgozása

Az adatok feldolgozása a *Show activity* gombra való kattintással indul el. Ekkor lefut a *show_activity()* függvény, melyben a kiválasztott fájlt betöltve létrejön egy pandas.DataFrame objektum, melynek a feldolgozása a *gpstracker.py* fájlban történik a *clear_df()* függvény által. Az adattábla minden sorában egy-egy NMEA mondat van, oszlopai pedig ennek szavai. Emlékeztetőül, a $GPRMC kezdetű sorok szolgálnak elégséges adattal a pozíció meghatározásához, ezek közül is csak azok tartalmaznak megbízható, helyes koordinátákat, melyeknek harmadik pozícióján - vagyis az adattábla kettes oszlopában - 'A' karakter szerepel. Minden más sor, ami nem felel meg ezen kritériumoknak, törlésre kerül az adattáblából. Az így kapott, 2.10 ábrán látható sorok tartalmilag megegyeznek 1.2.1 fejezetben található 1.3 ábrán látható példával.

	0	1	2	3	4	5	6	7	8	9	10	11	12
8735	$GPRMC	161915.00	A	4616.07333	N	02012.86923	E	13.865	40.95	200923	NaN	NaN	A*68
8741	$GPRMC	161917.00	A	4616.07944	N	02012.87445	E	14.053	34.74	200923	NaN	NaN	A*6A
8744	$GPRMC	161918.00	A	4616.08288	N	02012.87694	E	14.124	31.96	200923	NaN	NaN	A*67
8747	$GPRMC	161919.00	A	4616.08475	N	02012.88039	E	13.503	33.65	200923	NaN	NaN	A*64
8750	$GPRMC	161920.00	A	4616.08712	N	02012.88318	E	12.889	33.78	200923	NaN	NaN	A*6E

2.10 ábra *Helyes GPS adatokat tartalmazó dataframe részlete*

Nincs szükség az összes oszlopra, csak a 1-es, 3-as, 4-es, 5-ös ás 6-osra vagyis a mérés ideje, a szélesség, annak iránya, a hosszúság és annak irányát tartalmazó oszlopokra, illetve egy változóba kimentésre kerül a mérés kezdetének dátuma. Ha a szélesség déli vagy a hosszúság nyugati, akkor a koordináta kap egy negatív előjelet. Mindegyik oszlopban fix hosszúságú a karaktersorozat hossza, például az szélesség 10 karakter (4613.74811). Ha ettől eltérő, akkor *NaN* értéket kerül a helyére, majd ezeket az üres értékeket interpoláció segítségével a megelőző és a rákövetkező sorban szereplő értékek átlagával lesznek helyettesítve, azonban előtte megfelelő alakra kell hozni a koordinátákat, ugyanis azok számolásra alkalmatlan formában, stringként szerepelnek. Például a 4717.112671 szélesség az alábbiaknak felel meg:

- 47.28521118 fok
- 47 fok 17.112671 perc
- 47 fok 17 perc 6.76026 másodperc

Ezek közül az első opció megfelelő, a *convert_dms()* függvény alakítja ilyen formára a szélesség és hosszúság adatokat tartalmazó oszlopok értékeit. Az időt tartalmazó cellák

esetében is szükség van konverzióra, egyrészt stringről datetime típusúra kell konvertálni, valamint csak másodpercben is eltárolásra kerül az érték. Ezzel kész is van az adatok tisztítása, a már felesleges oszlopok eldobása után az adattábla *lat_dms* (szélesség fokban), *long_dms* (hosszúság fokban), *time* (mérés ideje), *time_in_sec* (mérési ideje másodpercben) 4 adatot tartalmazza. A 2.11 ábrán látható *calculate_distance()* függvényben kerül sor a koordináta párok által meghatározott pontok közötti távolság kiszámolására. Ehhez két új oszlop szükséges. Az első a *point*, amely az adott sorban szereplő szélességet és hosszúságot tartalmazó rendezett lista, a másik a *point_previous*, amely az előző sor *point* oszlop béli értéket tartalmazza. E kettő közötti távolság a *geopy* csomag *distance()* függvényével kerül kiszámításra. Mivel az első sorban csak egy pont van, nincs előző érték, ezért itt a távolságot nullára kell állítani.

```
def calculate_distance(df):
    df['point'] = df[['lat_d', 'long_d']].apply(tuple, axis=1)
    df['point_previous'] = df['point'].shift(1)
    df['distance'] = df.apply(
        lambda row: distance(row['point'], row['point_previous']).m, axis=1)
    df.at[0,'distance']=0
    df.drop(columns={'point', 'point_previous'}, axis=1, inplace=True)
    return df
```

2.11 ábra *calculate_distance() függvény*

A távolság és az idő ismeretében egyszerűen megkapható a sebesség az (1) képlettel,

$$v = \frac{s}{t} \tag{1}$$

ahol *v* a sebesség, *s* a megtett út, *t* az eltelt idő. Minden egyes adatpontra kiszámolható a 2.12 ábrán látható *calculate_speed()* függvénnyel a sebesség, kiegészítve egy mozgóátlagolással, a kiugró értékek simítása érdekében. A *speed* oszlop m/s-ban, a *speed_kmh* oszlop km/h-ban tárolja a sebességet. Ezzel már teljes a fájl feldolgozása, megjeleníthetőek az edzés alapvető adatai a felhasználói felületen.

```
def calculate_speed(df):
    df['speed'] = (df['distance'] / df['time_in_sec'].diff()).rolling(3).mean()
    df['speed_kmph'] = df['speed'] * 3.6
```

2.12 ábra *calculate_speed() függvény*

2.3. Eszközfejlesztés lépései

2.3.1. GPS modul tesztelése STM32F407 mikrovezérlővel

Az eszköz elkészítését a GPS modul tesztelésével kezdtem, mivel az általa nyújtott adatok szolgálnak a legfontosabb információkkal, a megtett távolsággal és különböző sebesség értékekkel. Ehhez egy STM32F407V-DISC1 fejlesztői panelt használtam, mivel ez állt rendelkezésemre és volt már tapasztalatom a programozásával. Nem készítettem még drivert, mindent a főprogramban implementáltam: az UART kommunikáció sima interrupt módban, DMA használata nélkül történt, a *HAL_UART_Transmit_IT()* és *HAL_UART_Receive_IT()* függvényekkel, melynek paraméterei megegyeznek a driverben használt DMA-t alkalmazó függvényekkel. A fájlkezelés annyiban különbözött a végleges verziótól, hogy a fájl nevének kiválasztásánál nem csak a szám lett inkrementálva, hanem létrehoztam egy tömböt fájlnevekkel és egy indexet növelve haladt végig egy for ciklus a tömbön, amíg olyan nevet nem talált, ami még nem szerepel a memóriakártyán. Hasonlóan a bemutatott, végleges verzióhoz, itt is a beépített LED világítása jelezte, ha már az utolsó lehetséges fájlnév került felhasználásra és a mérés végét szintén egy gomb megnyomása által generált megszakítás jelezte a programnak, annyi különbséggel, hogy a panelen beépített gombot használtam fel erre a célra, így a kapcsoláshoz nem kellett külön áramköri elemeket használni, csak a mikrovezérlőt, a NEO-7m és az SD kártya modult kellett összekötni. Az F4-es panel és az áramforrásként használt külső akkumulátor jelentősen befolyásolta a tesztelésre használt eszköz méretét, csak gyalogosan, kézben tartva tudtam mérni vele.

2.13 ábra *Az első mérés*

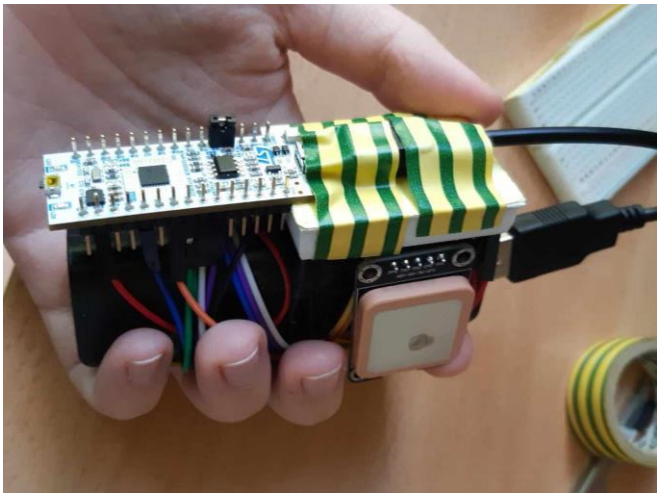
2.14 ábra *Első mérés eredménye*

Ahogy az első mérés alkalmával készült 2.13 ábrán is látszik, elég nagy méretű volt ez a verzió, azonban arra teljesen megfelelt, hogy megbizonyosodjak a GPS modul helyes működéséről. A

2.14 ábrán látható, hogy igen pontos eredményt adott, az adatlapban szereplő 2,5 méteres pontosságot tartotta.

2.3.2. GPS modul tesztelése STM32L412KB mikrovezérlővel

Ahhoz, hogy éles környezetben, edzés közben is tesztelhessem az GPS-t, lényegesen kisebb méretű kapcsolásra volt szükségem. Ennek érdekében döntöttem úgy, hogy lecserélem az F4-es mikrovezérlő panelt a korábban bemutatott, sokkal kisebb méretű L4-es NUCLEO boardra. Viszont jelentős méretbeli különbség azzal is járt, hogy ezen már nincs beépített nyomógomb, külön kell egyet az áramkörhöz kapcsolni, amihez egy kicsi próbapanelre is szükség van. Ennek ellenére, illetve egy kisebb méretű külső akkumulátor használatának köszönhetően mégis sikerült lefaragni a méretből, ahogy az a 2.15 ábrán látható, egy kisebb övtáskába beletéve már el tudtam vinni kerékpár és futó edzésre tesztelni.



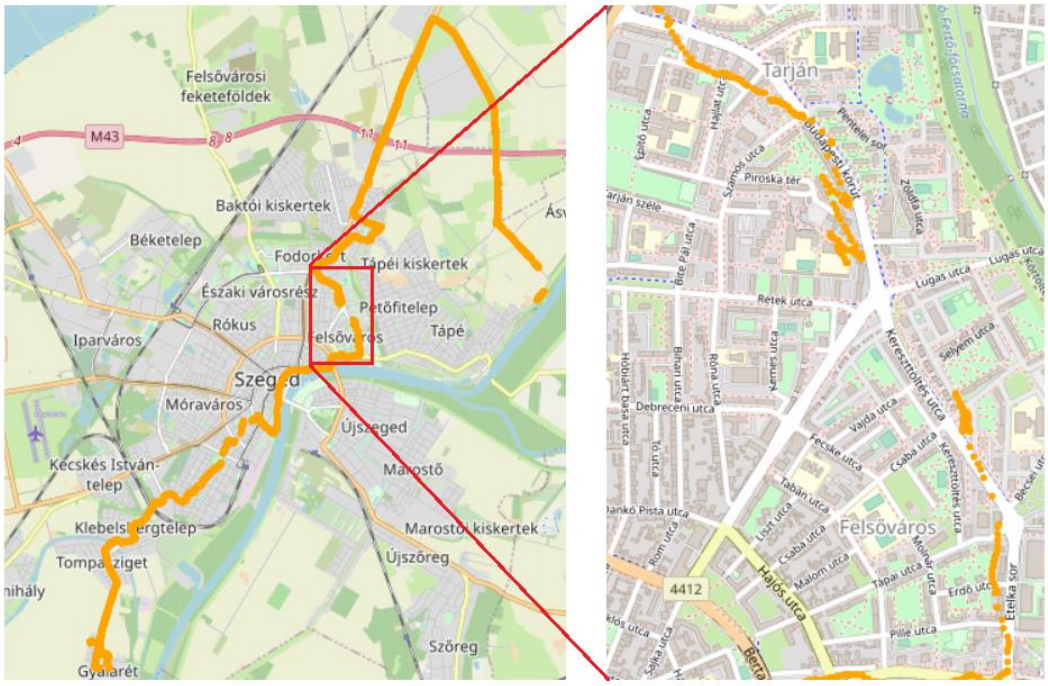
2.15 ábra *GPS modul teszteléséhez épített kapcsolás*

Ehhez a verzióhoz már elkészítettem a fent bemutatott *neo7m_gps* drivert, csak annyiban tér el ennek programja a végleges verziótól, hogy az *mpu6050* és a *hmc5883l* driverek még nincsenek implementálva.

Az 2.16 ábra bal oldalán látható, hogy itt már felmerültek problémák a mérés során, a legkritikusabb a jel megtalálásának ideje. A mérést a kép bal alsó részén látható, Gyálarét településen indítottam el, azonban az útvonalat jelző narancssárga vonal légvonalban nagyjából 11 km-rel arrébb jelenik meg. Ez időben körülbelül fél órát jelent, tehát nagyon lassan talált műholdat a modul. Korábban nem történt ilyen, viszont a későbbiekben egyszer megismétlődött: egy közel 40 perces futóedzés teljes ideje alatt nem sikerült jelet fogni a

modulnak. A két esetben a közös körülmény az volt, hogy a telefonom szintén az eszközt tartó tasakban, közvetlenül amellett volt, bekapcsolt GPS, Bluetooth és mobilinternet elérés mellett. Ebből arra következtetésre jutottam, hogy lehetséges, hogy a két eszköz között interferencia lép fel, a telefon zavarja a modult. Számos internetes fórum végigolvasása is megerősített benne, hogy több mint valószínű, hogy ez okozta a problémát.

Az 2.16 ábra jobb oldalán lévő, kinagyított részletben látható, hogy helyenként igen pontatlanul, akár 30 méteres hibával is mérhet a modul, többször el is veszíti a jelet. Ennek az oka nem más, mint hogy ezen a szakaszon 10 emeletes panelházak helyezkednek el szorosan az út mellett, amelyek akadályozzák a rádióhullámok zavartalan terjedését. Ez a probléma csak megerősített abban, hogy szükség van inerciális mérőegységek használatára is, melyekkel egyrészt korrigálhatóak a gyenge GPS jel miatti pontatlanságok, másrészt az olyan edzés során történő mozgásokat, mint a hirtelen irányváltások, gyorsulások, lassulások, felugrások, ütközések - melyek a mérni kívánt külső terhelést nagyban befolyásolják -, csak ezekkel lehet pontosan mérni.

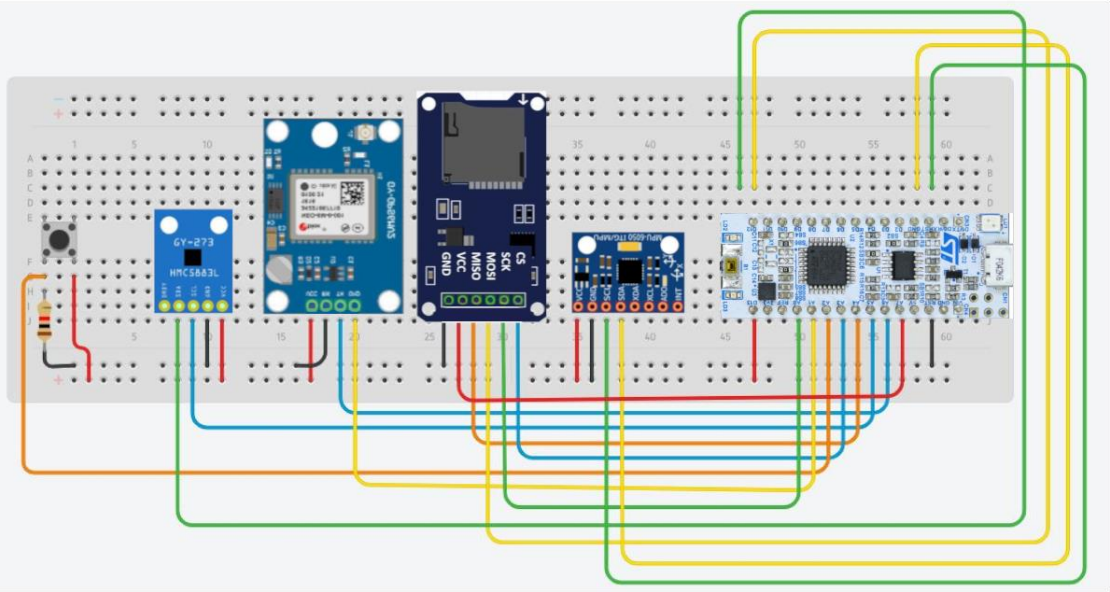


2.16 ábra *Egy mérés eredménye*

2.3.3. A végleges prototípus

A 2.1 és 2.2 fejezetekben bemutatott, végleges, MPU6050 és HMC5883L szenzorokat is tartalmazó áramkör esetében az illesztett hardverek mennyisége már jelentősen korlátozta a

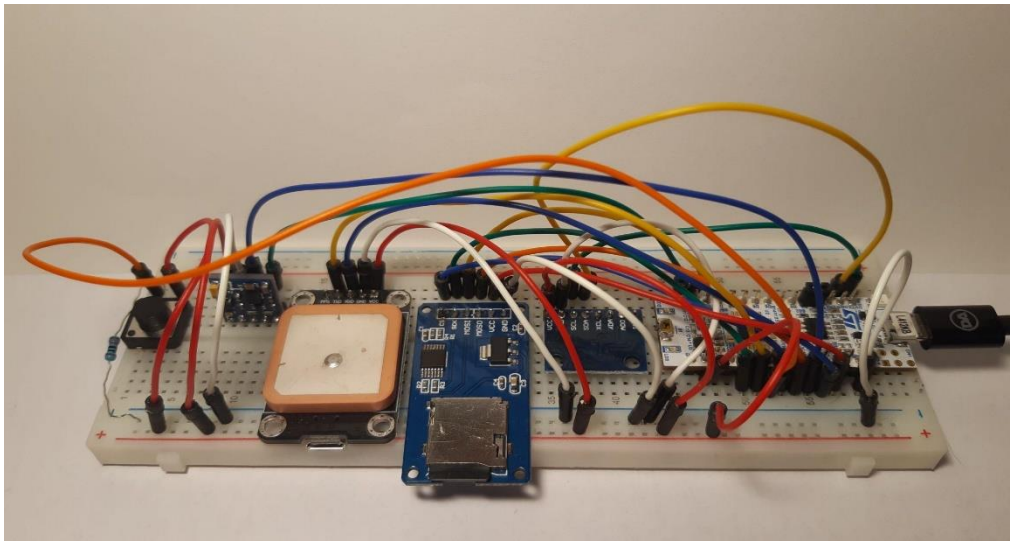
kapcsolás méretének minimalizálási lehetőségeit, így nem törekedtem a minél kisebb méretre, egyszerűen egy próbapanel és jumper kábelek felhasználásával kötöttem össze a komponenseket a 2.17 ábrán látható módon. Ennek azonban az a következménye, hogy az eszköz tesztelését csak korlátozott körülmények között volt lehetőség elvégezni. Ahhoz, hogy éles környezetben, edzés közben lehessen tesztelni, már áramkörtervezésre lenne szükség, de mivel csak prototípus tervezés és megvalósítás volt a feladatom, ezzel nem foglalkoztam. A helymeghatározást végző NEO-7m modul helyes működéséről már megbizonyosodtam az STM32F407V-DISC1 és NUCLEO-L412KB panelekkel való összekapcsolása során, mint ahogy azt a 2.3.2 és 2.3.1 fejezetekben kifejtettem. Az újonnan integrált szenzorokat a kapcsolás mérete és kialakítása miatt csak korlátozott körülmények között tudtam kis mértékben mozgatni és forgatni. Ezen tesztek alapján azt mondhatom, hogy a szenzorok megfelelő mérési adatokkal szolgálnak, adott elmozdulás esetén a becsült számérték-beli változást mutatják, alkalmasak az adatfeldolgozásra. Ha a későbbiekben sor kerül áramkör tervezésére és megvalósítására, akkor érdemes lehet majd kalibrálni a szenzorokat a tesztelést megelőzően.



2.17 ábra *Az eszköz kapcsolási rajza*

Az SD kártya 5 V bemeneti feszültséggel működik, minden más 3,3 V-tal. A mikrovezérlő panelen 5 és 3,3 V-os kivezetés is található, így feszültségosztással nem kellett foglalkozni. Az áramforrást az ST-Linken keresztül a számítógép vagy külső akkumulátor biztosítja. A nyomógomb használatához szükség van egy lehúzó ellenállás bekötéséhez a föld és a GPIO pin közé, ehhez egy 2,2 kΩ ellenállást használtam. A gomb lenyomásakor a pinen logikai magas állapot lesz, ez eredményezi a megszakítást a mikrovezérlő programjában, melynek hatására

megtörténik a mérési adatok mentésére szolgáló fájlok lezárása és az SD kártya leválasztása, vagyis véget ér a mérés. A kapcsolás megvalósítása, vagyis a végleges prototípus a 2.18 ábrán látható.



2.18 ábra *Az végleges prototípus*

3. Összefoglalás

Szakdolgozatom elkészítése során az informatika széles spektrumába nyertem betekintést, hiszen a témát adó eszköz és alkalmazás megvalósítása számos terület ismeretét igényelte: szükség volt elektronikai ismeretekre, hardverek működésének és programozásának ismeretére, ezek teszteléséhez különböző elektronikai műszerek használatára (például PicoScope, multiméter), szoftver tervezésre, fejlesztésre és tesztelésre, de még forrasztani is kellett néhány alkatrészt. Végig egy komplex rendszerben kellett gondolkodnom, az egyetemi éveim alatt tanultak széles körét alkalmaznom kellett.

Végeredményként sikerült megvalósítanom egy olyan prototípust, amely választható frekvenciával és méréstartományokkal képes mérni globális pozíciót, gyorsulást, elfordulást és irányt. Ezeket a mért adatokat egy memóriakártyán eltárolja. Egy gomb megnyomásával jelezhető a mérés vége és egy LED felkapcsolása jelzi a felhasználónak, ha ideje áthelyezni a mérési fájlokat a memóriakártyáról. Az ehhez készített, bővíthető alkalmazásban lehetőség van fájl és edzés típus kiválasztására. Hibás fájl esetén hibaüzenettel jelzi a felhasználónak, hogy válasszon másikat. Megjeleníti az edzés néhány alapvető adatát: kezdés dátumát és idejét, időtartamát, megtett távolságot, átlagos tempót és maximális sebességet. Egy grafikonon kirajzolja a sebesség változását az edzés folyamán és egy térképen nyomon követhető a megtett útvonal.

Rengeteg lehetőség van továbbfejlesztésre, úgy gondolom a legfontosabb az áramkör tervezés lenne, hogy az eszköz valóban hordható legyen. A leginkább kiaknázatlan terület az adatok feldolgozása: megoldandó feladat lehet a GPS adatok korrekciója a többi szenzorral, a korrigált adatok alapján sebesség zónák számolása és ábrázolása, az ezekben megtett távolság és idő kiszámítása, az IMU szenzorok által mért értékek alapján a gyorsulások, lassulások, irányváltások, ugrások, ütközések számának és mértékének kiszámítása, az összes adat alapján az edzés intenzitásának kiszámítása. Hosszú távon, nagy mennyiségű adat összegyűjtése után akár mesterséges intelligencia segítségével összetettebb elemzéseket is lehetne végezni az adatokon, amelyekkel például sérüléseket lehetne előre jelezni.

Irodalomjegyzék

[1] K. Simon és S. Stefan, *Soccernomics*.

[2] „Wearable Technology Sensors”. [Online]. Elérhető: https://www.coursera.org/learn/wearable-technologies/lecture/9kRN2/wearable-technology-sensors

[3] „What is player load?” [Online]. Elérhető: https://support.catapultsports.com/hc/en-us/articles/360000510795-What-is-Player-Load-

[4] „STATSports Apex Athlete Series”. [Online]. Elérhető: https://statsports.com/apex-athlete-series

[5] „Wearable Soccer (Football) Sensors That Track Shots, Passes And More”. [Online]. Elérhető: https://sportstechnologyblog.com/2022/06/01/wearable-soccer-football-sensors-that-track-shots-passes-and-more/

[6] „SZTE Távközlő hálózatok kurzus 2022-es 11. előadás jegyzete”.

[7] „Műhold pályákat ábrázoló kép”. [Online]. Elérhető: https://www.researchgate.net/figure/A-GNSS-consisting-of-a-constellation-of-24-satellites_fig1_280133049

[8] „Műhold jelek metszéspontja”. [Online]. Elérhető: https://www.earthscope.org/what-is/gps/

[9] „Majdnem haladó: Az NMEA és a GGA”. [Online]. Elérhető: https://gpstakarok2.wixsite.com/gpstakarok/nmea-s-gga

[10] „Inertial measurement unit”. [Online]. Elérhető: https://en.wikipedia.org/wiki/Inertial_measurement_unit

[11] „u-blox 7 Receiver Description”. [Online]. Elérhető: https://content.u-blox.com/sites/default/files/products/documents/u-blox7-V14_ReceiverDescriptionProtocolSpec_%28GPS.G7-SW-12001%29_Public.pdf

[12] „Neo-7m modul”. [Online]. Elérhető: https://i0.wp.com/www.rytronics.in/wp-content/uploads/2022/07/neo-7m-gps-module.jpg?fit=600%2C600&ssl=1

[13] „MPU-9250 Product Specification Revision 1.1”. [Online]. Elérhető: https://invensense.tdk.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf

[14] „3-Axis Digital Compass IC HMC5883L”. [Online]. Elérhető: https://cdn-shop.adafruit.com/datasheets/HMC5883L_3-Axis_Digital_Compass_IC.pdf

[15] „MPU6050 modul”. [Online]. Elérhető: https://cdn.webshopapp.com/shops/144750/files/289453791/mpu6050-6-axis-gyroscope-accelerometer-sensor.jpg

[16] „HMC5883L modul”. [Online]. Elérhető: https://components101.com/sites/default/files/components/HMC5883L-Magnetometer-Module.jpg

[17] „Micro SD Card Adapter Module”. [Online]. Elérhető: https://components101.com/modules/micro-sd-card-module-pinout-features-datasheet-alternatives

[18] „SD kártya modul”. [Online]. Elérhető: https://robu.in/wp-content/uploads/2016/03/Micro-SD-Card-Reader-Module-4.jpg

[19] „File Allocation Table”. [Online]. Elérhető: https://en.wikipedia.org/wiki/File_Allocation_Table

[20] „FatFS”. [Online]. Elérhető: http://elm-chan.org/fsw/ff/

[21] „STM32_SPI_SDCARD”. [Online]. Elérhető: https://github.com/eziya/STM32_SPI_SDCARD/tree/master/STM32F4_HAL_SPI_SDCARD
[22] „JSON”. [Online]. Elérhető: https://www.json.org/json-en.html
[23] „cJSON library”. [Online]. Elérhető: https://github.com/Orientsoft/BorgnixSDK-STM32/blob/master/rtt/bsp/stm32f40x/cjson/cJSON.c
[24] „STM32F407G-DISC1”. [Online]. Elérhető: https://www.st.com/en/evaluation-tools/stm32f4discovery.html
[25] „NUCLEO-L412KB”. [Online]. Elérhető: https://www.st.com/en/evaluation-tools/nucleo-l412kb.html
[26] „STM32CubeIDE”. [Online]. Elérhető: https://www.st.com/en/development-tools/stm32cubeide.html
[27] Z. Gingl és R. Z. Mingesz, „Laboratory practicals with the C8051Fxxx microcontroller family”.
[28] Z. Gingl, „SZTE Mikrovezérlők Alkalmazástechnikája kurzus, Mikrovezérlő alapok jegyzet”. [Online]. Elérhető: https://www.inf.u-szeged.hu/~gingl/hallgatoknak/mikrovezerlok/01_MikrovezerloAlapok.pdf
[29] „Getting started with DMA”. [Online]. Elérhető: https://wiki.st.com/stm32mcu/wiki/Getting_started_with_DMA

Nyilatkozat

Alulírott Lovászi Zsuzsanna mérnökinformatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Műszaki Informatikai Tanszékén készítettem, mérnökinformatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel. Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

2023. november 26.

Aláírás

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani Dr. Mingesz Róbertnek, hogy szakdolgozatom létrejöhetett. Örök hálával tartozom Tönköly Andornak, amiért felkarolt, mentorált, egyengette utamat a szakdolgozat, szakmai gyakorlat és a képzés utolsó évei során. Nélküle valószínűleg nem jutottam volna el idáig. Külön köszönet illeti, amiért biztosította a szakdolgozatomhoz szükséges hardvereket saját, személyes készletéből. Végezetül köszönöm a Trifeminátusnak a sok támogatást, hogy mind lelkileg, mind szakmailag teljesíthetőbbé tették az elmúlt évek akadályait.

Mellékletek

A prototípushoz felhasznált perifériák:

Kommunikáció:

- I2C1: standard mód (átviteli sebesség: 100 kHz)
- I2C3: standard mód (átviteli sebesség: 100kHz)
- SPI1: full-duplex master mód, 8 bites adatok, 2 Mbit/s átviteli sebesség
- USART2: aszinkron mód, interrupt mód DMA-val, 9600 bits/s baud rate

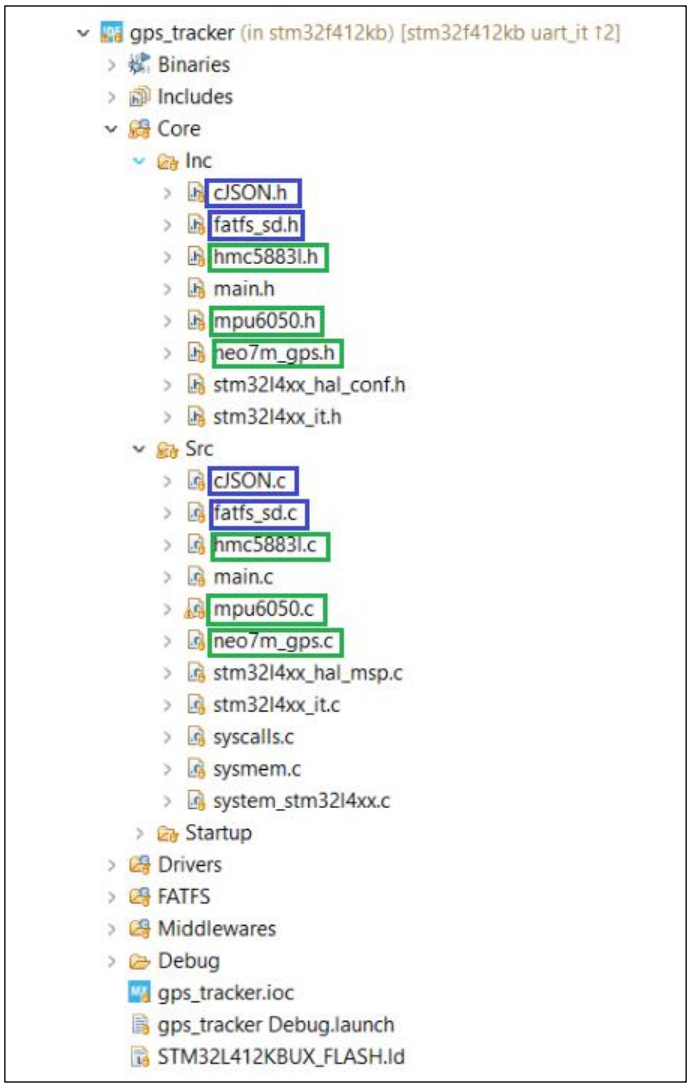
Timer:

- TIM6: IMU szenzorokkal való mérések időzítésére, interrupt mód, 10 Hz auto-reload frekvencia

Pinek:

- PA1: SPI1 SCK
- PA2: USART2 TX
- PA3: USART2 RX
- PA4: EXTI4, külső interrupt
- PA5: SPI1 SS
- PA6: SPI1 MISO
- PA7: I2C3 SCL
- PA9: I2C1 SCL
- PA10: I2C1 SDA
- PB3: MCU panelen beépített LED vezérléséhez szükséges
- PB4: I2C3 SDA
- PB5: SPI1 MOSI

1. melléklet *NUCLEO-L412KB konfigurációja*



***2. melléklet** A projekt könyvtárszerkezete*

A 2. mellékleten látható könyvtárszerkezet az STM32CubeIDE generálta, kivéve a kijelölt fájlokat, melyek utólag került beillesztésre. A kék színnel jelzettek az SD kártya írásához szükség, külső forrásból importált library-k, a zölddel jelzettek az általam megírt szenzor driverek.

```
while (1)
{
    if(gps.rx_cplt)
    {
        f_putc(gps.gps_data, &gps_file);
        gps.rx_cplt=0u;
        GPS_Receive(&huart2, &gps.gps_data,1);
    }
    if(time_elapsed)
    {
        char *json_str=Convert_to_JSON();
        f_puts(json_str, &imu_file);
        cJSON_free(json_str);
        MPU6050_GetRawData(&hi2c1, &mpu);
        HMC5883L_ReadData(&hi2c3, &hmc);
        time_elapsed=0;
    }
    if(session_end)
    {
        f_close(&gps_file);
        f_close(&imu_file);
        f_mount(NULL, "", 0);
        fres=FR_NOT_READY;
        session_end=0;
    }
  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
}
```

3. ***melléklet*** *Mikrovezérlő program főciklusa*