



M Ű E G Y E T E M 1 7 8 2

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Massár Lóránt Mátyás

PÓKSZERŰ, JÁRÓ ROBOT FEJLESZTÉSE

KONZULENS

Nagy Ákos

BUDAPEST, 2020

Tartalomjegyzék

| | |
|---|-----------|
| Összefoglaló | 6 |
| Abstract..... | 7 |
| 1 Bevezetés | 8 |
| 1.1 A pókszerű robot..... | 8 |
| 1.2 Célkitűzések..... | 9 |
| 2 Hardver | 11 |
| 2.1 Integrált áramkörök, érzékelők | 11 |
| 2.1.1 STM32L476RET6 | 11 |
| 2.1.2 ESP-01 | 12 |
| 2.1.3 YDLIDAR X2..... | 13 |
| 2.1.4 BNO055 Adafruit | 14 |
| 2.2 NYÁK terv | 15 |
| 2.2.1 Fejlesztő környezet | 15 |
| 2.2.2 Eredeti nyomtatott áramkör bemutatása | 16 |
| 2.2.3 Nyomtatott áramkör módosítása..... | 16 |
| 3 Felhasznált szoftver technológiák..... | 19 |
| 3.1 ROS..... | 19 |
| 3.1.1 ROS működése | 19 |
| 3.1.2 Beágyazott környezet - ROSSerial | 20 |
| 3.1.3 3D szimulációs eszközök - RVIZ | 21 |
| 3.1.4 Térképező algoritmusok – hector_slam..... | 22 |
| 3.2 FreeRTOS | 24 |
| 3.2.1 Taszkok és ütemezés..... | 24 |
| 3.2.2 Taszkok közötti kommunikáció..... | 25 |
| 4 Beágyazott szoftver | 26 |
| 4.1 Áttekintés | 26 |
| 4.2 FreeRTOS | 27 |
| 4.2.1 Konfiguráció | 27 |
| 4.2.2 Taszkok..... | 28 |
| 4.2.3 Esemény kezelés | 32 |
| 4.2.4 Dinamikus memórafoglalás | 33 |

| | |
|--|-----------|
| 4.2.5 Task osztály leszármazottak diagramjai | 33 |
| 4.3 Wifi kommunikáció – ESP-01 vezérlése | 34 |
| 4.3.1 AT parancsok..... | 35 |
| 4.3.2 ESP-01 inicializálása | 35 |
| 4.3.3 Adat fogadás | 36 |
| 4.3.4 Adatküldés | 37 |
| 4.4 IMU szenzor – BNO055 vezérlése | 38 |
| 4.4.1 BNO055 interfésze | 39 |
| 4.4.2 BNO055 inicializálása | 41 |
| 4.4.3 Mérési adat lekérdezése | 42 |
| 4.4.4 Debug kérés | 43 |
| 4.5 LIDAR szenzor – YDLIDAR X2 vezérlése | 44 |
| 4.5.1 YDLIDAR könyvtár | 45 |
| 4.5.2 Mérési adat feldolgozása | 45 |
| 4.6 Mozgás vezérlése | 49 |
| 4.6.1 Mozgás elemek | 50 |
| 4.6.2 Mozgás kérése..... | 50 |
| 4.7 Beágyazott ROS csomópont | 51 |
| 4.7.1 A ROSNode fontosabb elemei..... | 51 |
| 4.7.2 ROS node inicializálása | 52 |
| 4.7.3 A ROS node futása | 53 |
| 5 Tesztek, eredmények | 54 |
| 5.1 Forrasztás, élesztés..... | 54 |
| 5.2 ROS eredmények | 55 |
| 5.2.1 ROS kapcsolat..... | 55 |
| 5.2.2 RVIZ | 56 |
| 5.2.3 Robot mozgatása..... | 58 |
| 5.2.4 Térképezés | 58 |
| 6 Összegzés, értékelés | 62 |
| 6.1 Továbbifejlesztési lehetőségek | 62 |
| Irodalomjegyzék..... | 63 |
| Forráskód | 63 |
| Fejlesztő eszközök | 63 |
| Adatlapok, kézikönyvek | 63 |

| | |
|----------------------|----|
| ROS..... | 64 |
| FreeRTOS | 64 |
| Egyéb források | 64 |

HALLGATÓI NYILATKOZAT

Alulírott **Massár Lóránt Mátyás**, szigorló hallgató kijelentem, hogy ezt a diplomatervet 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. 20.

.....
Massár Lóránt Mátyás

Összefoglaló

Napjainkban egyre növekvő ütemben fejlődnek és válnak mindennapi életünk részévé az okos eszközök, amelyek között már a mobil robotok is egyre jelentősebb részt képviselnek, gondoljunk csak a mindenki által ismert robotporszívókra, önjáró járművekre, raktárakban, vagy éppen kórházakban használt szállító robotokra.

Ezek a mozgásra képes eszközök gyakran komplex rendszerek, amelyeknek le kell küzdeniük számos, a szakterületen közismert problémát. Ahhoz, hogy a robot sikeresen el tudja végezni a számára elrendelt mozgást, a robotnak ismernie kell a saját pozícióját és a környezetének felépítését, például azt, hogy milyen akadályok vannak az útjában. Továbbá gyakori követelmény lehet az is, hogy a robotot valós időben szeretnénk vezérelni, így a robottal tudnunk kell üzemelés közben is kommunikálni.

Természetesen az előző bekezdésben bemutatott problémákon felül számtalan egyéb, akár robotok között eltérő probléma is felmerülhet, azonban a dolgozatom fókusza egy, a tanszéken található, négy lábbal rendelkező, pókszerű robot volt. Ezt a robotot mozgásra képes állapotban vettem át, azonban sem kommunikációs csatornával, sem szenzorokkal nem volt ellátva. Célom tehát az volt, hogy felszereljem a megfelelő érzékelőkkel, és beintegráljam egy olyan rendszerbe, amely biztosít számára térképezési és útvonaltervezési algoritmusokat, így felkészítve arra, hogy adaptív módon, az éppen aktuális tereptől függően legyen képes mozgási feladatokat végrehajtani.

Munkám során elkészítettem a robot új hardver tervét, amelyet később össze is szereltem, illetve leimplementáltam az új vezérlőszoftvert, ami nem csak az új hardvert támogatja, de kompatibilissé teszi a robotot a ROS nevű rendszerrel, amely segítségével különböző térképezési, útvonaltervező és egyéb mozgással kapcsolatot szolgáltatást tudunk biztosítani a robotnak.

Abstract

Nowadays smart technology is improving in an increasing pace, and simultaneously becoming a part of our lives, this including mobile robots as well, just think about the well-known robot vacuums, self-driving vehicles, or the transporting robots used in warehouses and hospitals.

These mobile devices are often rather complex, and they have to deal with numerous issues, that are well-known among professionals. For a robot to be able to complete a movement task, it has to know its own position and its environment, for example what obstacle can be found around it. It is often a requirement, that we want to control the robot in real time, therefore we must be capable to communicate with it during operation.

Of course the problems presented in the previous paragraph are only a small portion of the possible hardships, and we haven't even mentioned the problems that can vary between different robot builds, however the focus of my paper was a four legged, spiderlike robot, belonging to my department. At the time I started to work on this robot, it was capable to perform movement, but it had no sensors, neither any means of communication. My goal was therefore to upgrade this device with proper sensors, and integrate it into a system, that can provide mapping and route planning algorithms, making it capable of performing movement in various terrain, adapting to the current conditions.

As a result of my work I have created a new hardware design, which I later assembled as well. I have implemented a new software, which is not only handling the new hardware parts, but also makes the robot compatible with a system called ROS. This system can provide mapping and route planning services, among many other.

1 Bevezetés

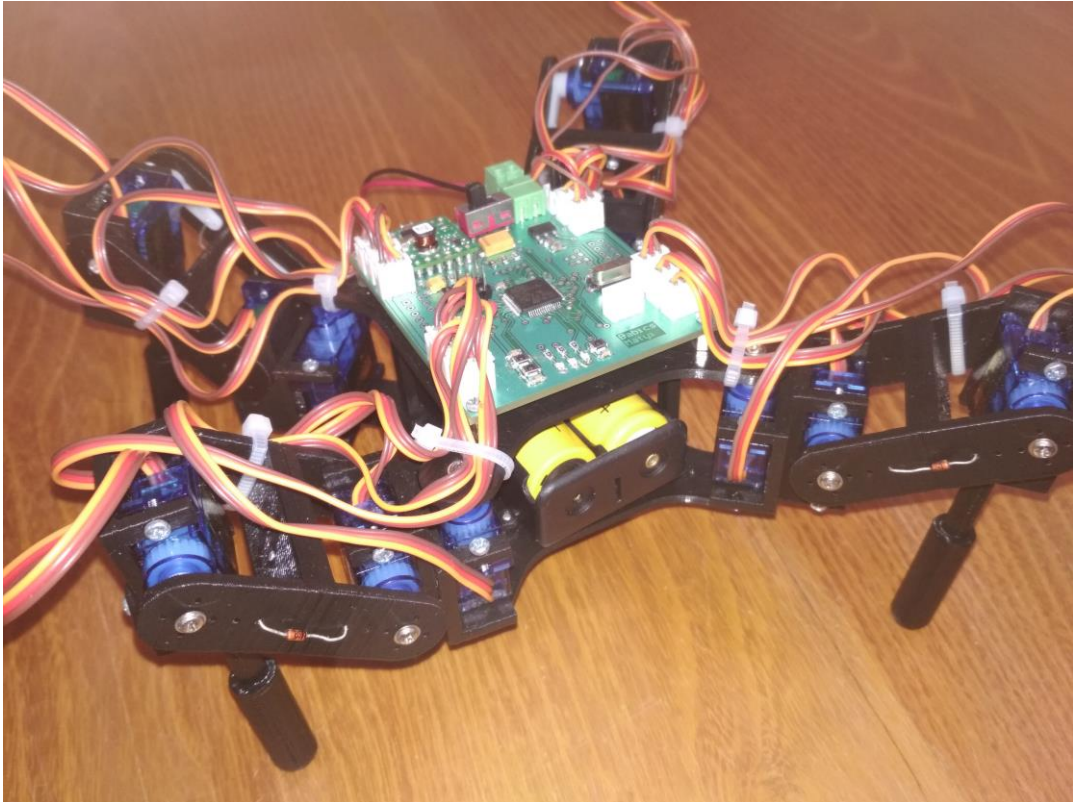
Manapság az okoseszközök világában élünk. Az okostelefonoktól az okosvárosokig, az elektronikai ipar minden szegletében megjelenik az a szemléletmód, hogy az "okos" eszköz értékesebb hagyományos társainál. De mitől lesz egy szerkentyű okos? A teljesség való törekvés nélkül elmondhatjuk, hogy az okoseszközökre általában jellemzőek a következő tulajdonságok:

- Nagyfokú automatizáltságra való törekvés
- Magasszintű, felhasználóbarát interfész, mely elfedi a belső működést
- Stabil kommunikációs csatorna
- Az eszköz képes saját és környezete állapotáról adatot gyűjteni, jellemzően szenzorok segítségével

Diplomamunkám során egy másik hallgató, Babics Mátyás, által készített robottal dolgoztam, és feladatom ezen robot fejlesztése, vagy akár mondhatnánk azt is, hogy "okosítása" volt.

1.1 A pókszerű robot

A robot, aminek fejlesztésén dolgoztam, önmagában is egy kész projectnek tekinthető. Szerkezetét tekintve leginkább a pókszerű jelző tudja leírni, négy lábbal rendelkezik, ezek mind egy központi törzs elemhez csatlakoznak, a törzs elem egy-egy sarkánál. A lábak 3 csuklóból állnak így biztosítva a megfelelő mennyiségű szabadságfokot ahhoz, hogy a láb bármilyen pozíciót fel tudjon venni.



1. ábra: A Babics Mátyás által készített pókszerű robot

A robothoz készült egy program is, amely tartalmazta a lábak mozgásának vezérlését. Ez a program C++ nyelven íródott az STMicroelectronics gyártó LL könyvtárát felhasználva, mely a hardver absztrakciós réteget biztosítja.

Összefoglalva, azon a ponton, amikor átvettem a robot fejlesztését, a robot képes volt előre haladó és forgó mozgásra, illetve ezek összefűzésével egy négyzet alakú pálya bejárására.

1.2 Célkitűzések

A célom az volt, hogy a robotot sikerüljön úgy tovább fejleszteni, hogy képes legyen egy PC-s eszköztől parancsokat fogadni, illetve az állapotáról információt küldeni. Mindezt egy olyan interfészen tegye meg, hogy a felhasználó a robot belső működésének részletes ismerete nélkül is tudja azt használni. Ahhoz, hogy ezeket a célokat meg tudjam valósítani a következő követelményeket fogalmaztam meg:

- A robot legyen képes egy megbízható csatornán, legalább egy asztali számítógéppel kommunikálni.

- A robot rendelkezzen legalább egy olyan szenzorral, aminek segítségével képes a környezetében levő objektumokról, akadályokról információt gyűjteni.
- A robot rendelkezzen legalább egy olyan szenzorral, amin keresztül képes saját pozíciójáról adat szerezni.
- A robot legyen beintegrálható ROS alapú rendszerbe.

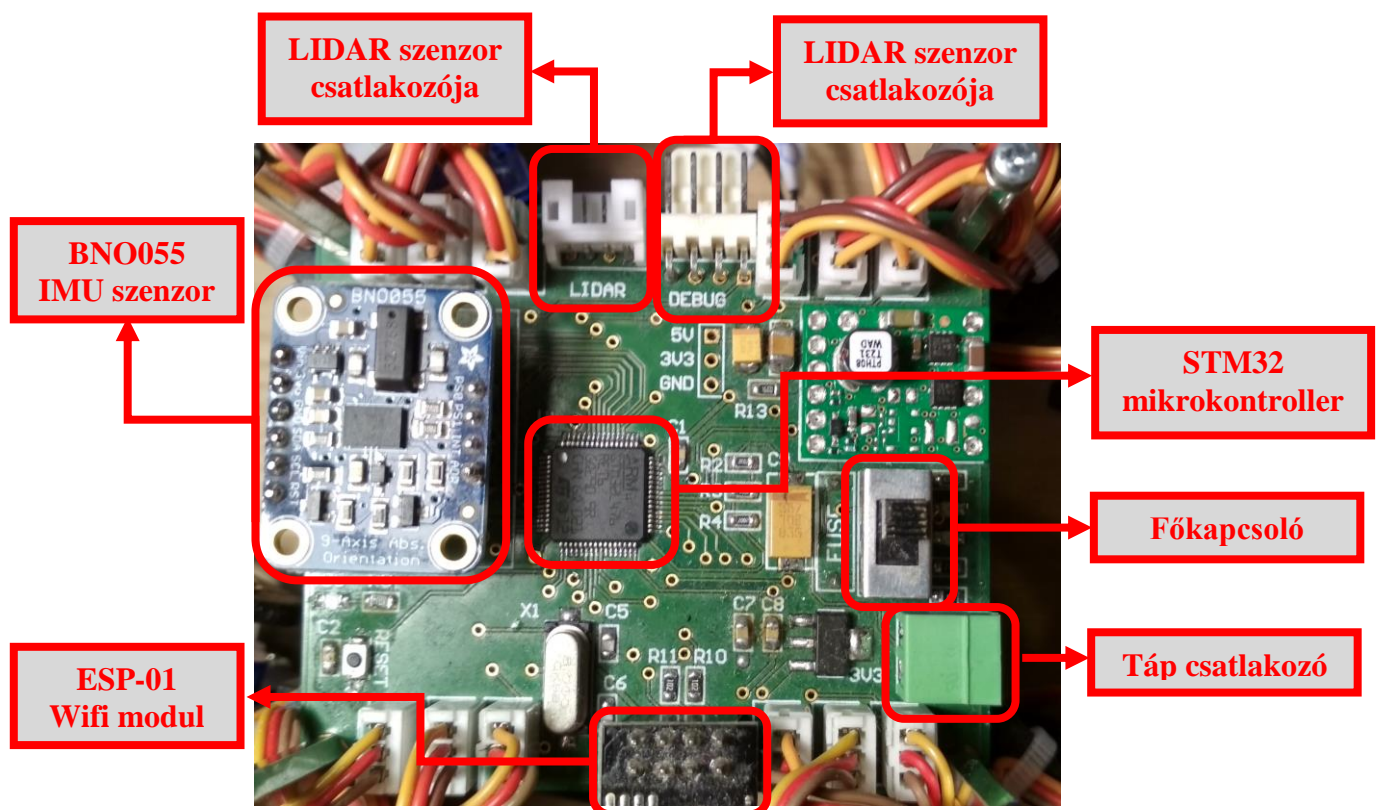
A megfogalmazott célok eléréséhez meg kellett vizsgálnom az elérhető hardveres megoldásokat, és kielemezni azt, hogy miként lehet kibővíteni a meglévő elektronikát úgy, hogy a rendszerbe belekerülhessenek az új eszközök. Az új hardvert össze kellett forrasztanom, majd felélesztenem. A hardveres részeken túl el kellett készítenem a beágyazott vezérlő szoftvert is. Ehhez meglévő technológiák megismerésére és saját kód implementálásra is szükségem volt.

2 Hardver

Ez a fejezet a robot hardverének bemutatását szolgálja, röviden ismerteti azokat az eszközöket, amelyeket a szoftver fog majd vezélni.

2.1 Integrált áramkörök, érzékelők

A robot elektronikáját egy, a robot tetejére erősített, nyomtatott áramköri lemez tartalmazza.



2. ábra: A robot nyomtatott áramköri lemeze

2.1.1 STM32L476RET6

Az STM32L476RET6 típusú mikrokontroller a robot központi vezérlőegysége, így a következőkért felelős:

- A robot applikációjának futtatása.
- A szervomotorok vezérlése belső időzítők által generált PWM jel segítségével.

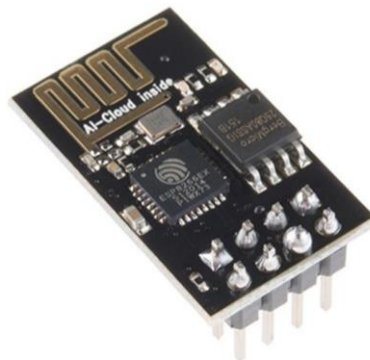
- Az ESP-01-es wifi chip-el való kommunikáció folytatása az erre kijelölt UART vonalon. Továbbá szoftver szinten az ESP-01 felinicializálása és a vezeték nélküli kommunikáció kezelése olyan alacsony szinten, hogy ez elrejtve maradjon a felsőbb rétegek előtt (például ROS).
- A szenzorok számára megfelelő kommunikációs vonalak biztosítása, illetve a szenzor adatok feldolgozása.
- ROS node futtatása, és a csomópont által használt topic-ok biztosítása.

A szervomotorok vezérlését már a Babics Mátyás által készített hardver és szoftver részek is biztosították, így ezek részletezésére nem fogok kitérni a következőkben.

2.1.2 ESP-01

A kommunikációs csatorna kiválasztásánál a döntésem a wifi (vezeték nélküli hálózati protokoll) alapú kommunikációra esett. A wifi kommunikáció előnyei közé tartozik, hogy vezeték nélküli kapcsolatról van szó, ami egy mozgó robotnál praktikus, hiszen nincsenek kábelek, amik korlátoznák a mozgást. A wifi továbbá egy jól standardizált, és széleskörben alkalmazott protokoll, amit manapság már minden PC-s eszköz támogat, így a kommunikáció másik oldala miatt nem kell aggódnunk. Harmadik előny, hogy az IoT (Internet of Things – ”okos technológia”) elterjedésével egyre több olcsó, de megbízható beágyazott eszköz található a piacon.

Az ESP-01 az Espressif gyártó terméke, ami egy ESP8266-os chippel és antennával rendelkező board. Feladata, hogy egyszerű vezeték nélküli, TCP/IP protokoll alapú kapcsolatot biztosítson saját hálózat létrehozásával. A kártya UART vonalon kommunikál az ESP8266-os modullal.



3. ábra: ESP-01

Az ESP-01 önmagában is alkalmas lenne vezérlő programok futtatására, ugyanis a chip lehetőséget ad saját készítésű szoftver telepítésére, azonban a robot koncepciójában mindössze TCP üzenetek küldéséért és fogadásáért felelős. A robot teljes szoftvere egyébként is összetettebb annál, minthogy az ESP-01 futtatni tudja. Az üzenetek feldolgozását a ST mikrovezérlő végzi el.

2.1.3 YDLIDAR X2

A LIDAR (Light Detection and Ranging – lézer alapú távérzékelés) technológia a radarokhoz hasonlóan időmérésre alapuló távolság meghatározásra használt módszer [13]. A különbség mindössze annyi a két eljárás között, hogy míg a radar rádióhullámokat, addig a LIDAR lézerimpulzusokat használ.

A LIDAR szenzorok egyik gyakori alkalmazása objektumok, akadályok észlelése, és ezek összegyűjtésével a környezet feltérképezése. A robot pontosan erre a célra lett ellátva egy YDLIDAR X2 gyártmányú LIDAR érzékelővel.



4. ábra: YDLIDAR X2

YDLIDAR X2 egy 360°-os forgásra képes, kétdimenziós síkban működő LIDAR szenzor [5]. Az eszköz kis méretét, alacsony fogyasztását, és maximálisan 8 méter körüli hatótávolságát tekintve elsősorban kisebb, beágyazott rendszerekre lett tervezve. A LIDAR 5V-os tápfeszültségről működtethető, amely kompatibilis a robot számára elérhető feszültség szintekkel.

A LIDAR a táp és föld vonalakon kívül egy bemenettel és egy kimenettel rendelkezik. Az eszköz az általa mért adatokat a kimeneti vonalon (Tx) biztosítja, soros aszinkron protokollon keresztül. A bemeneti vonal pedig egy 3.3V-os szinten lévő

PWM jelet vár, amivel a LIDAR-t forgató motor sebesség lehet állítani a kitöltési tényező módosításával. Ez utóbbi azonban elhanyagolható, ebben az esetben a motor a maximális forgási sebességgel fog működni.

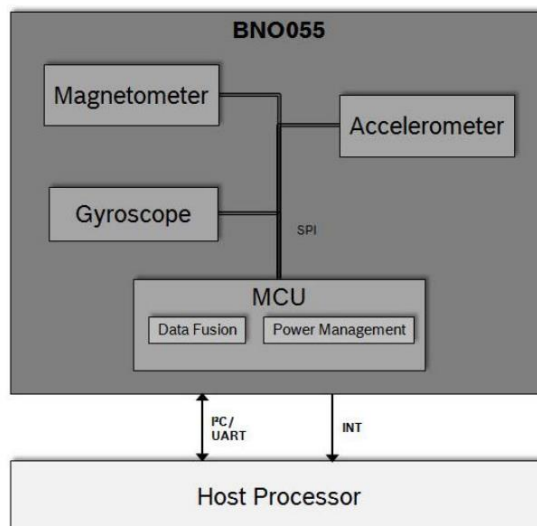
2.1.4 BNO055 Adafruit

Az IMU (Inertial Measurement Unit) egy olyan érzékelő fajta, amely több, általában kettő vagy három, szenzort tartalmaz egy csomagban, és a feladata az, hogy információt biztosítson a szenzor térbeli elhelyezkedéséről és mozgásáról [15].

A Bosch BNO055 érzékelője, amely a robotra került, egy kilenc szabadságfokú (Degree of Freedom, vagy DOF) IMU [6]. Ez azt jelenti, hogy az szenzor tartalmaz (lásd 5. ábra):

- gyorsulás mérőt – 3 szabadságfok x, y, z irányban történő gyorsulásra
- giroszkópot – 3 szabadságfok x, y, z tengely körüli forgásra
- magnetoszkópot – 3 szabadságfok a térben való abszolút orientáció meghatározására.

Figure 1: system architecture

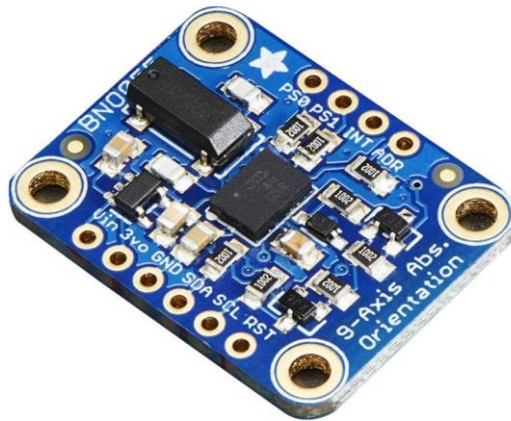


5. ábra: Bosch BNO055 Architektúráka [6]

Az egyetlen probléma ezzel az érzékelővel a tokozása volt, ugyanis a gyártó az eszköz kompaktására törekedve LGA tokozást alkalmazott. Ez nem jelentette gondot egy automatizált gyártósoron készült NYÁK esetében a fejlett forrasztási technológiáknak

köszönhetően, azonban a robot esetében az alkatrészek kézzel kerülnek felforrasztásra, ezért egy másik megoldást kellett találni.

Szerencsére elérhető ennek a szenzornak számos olyan változata is, amely egy board-al együtt kapható. Az egyik ilyen kártya az Adafruit gyártó terméke, amely a 6. ábrán látható.



6. ábra: Adafruit BNO055

A fejlesztőkártya UART vonalon biztosítja a kommunikációt a szenzor és a mikrokontroller között. Ráadásul mindezt az 5. ábrának megfelelően úgy teszi, hogy nem a nyers mérési eredményeket adja vissza, hanem egy már részben feldolgozott és összegzett eredményt bocsát a rendelkezésünkre, ez jelentősen megkönnyíti az adatok kezelését a mikrovezérlő szintjén. Ha ilyen módon kérjük le a mérési eredményeket, akkor úgynevezett fúziós adatokat kapunk. A fúziós adat kétféle formában kérhető: Euler szögek, vagy kvaternió.

2.2 NYÁK terv

A diplomamunkám során új funkcionalitásokkal és alkatrészekkel egészítettem ki a robotot, ami természetesen azt is jelentette, hogy egészen a nyomtatott áramkör szintjéig vissza kellett nyúlnom.

2.2.1 Fejlesztő környezet

A NYÁK tervét Altium Designer segítségével készítettem el [2]. Az Altium Designer egy CAD (Computer-aided design) rendszer, amely számos eszközzel támogatja a nyomtatott áramkör tervezés minden lépését, az áramkörtervezéstől egészen

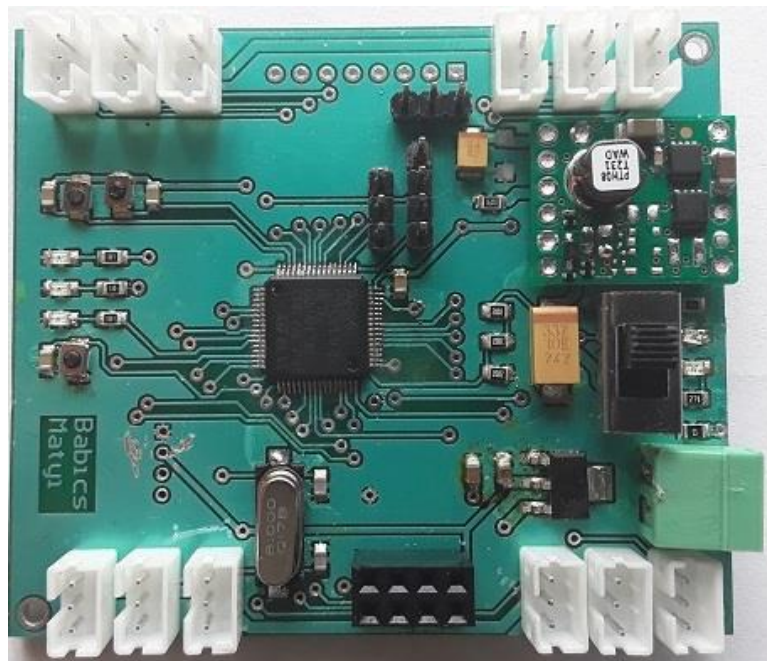
az alkatrészek fizikai elhelyezéséig. A nyomtatott áramkör áttervezésénél Babics Mátyás design-jából indultam ki, és azt bővítettem ki a szükségleteknek megfelelően.

2.2.2 Eredeti nyomtatott áramkör bemutatása

Az kiindulásként felhasznált nyomtatott áramkör már rendelkezett több olyan alkatrésszel is, amelyeket valamilyen formában megtartottam az új NYÁK-on is.

A tápforrás biztosításáért felelős elemek érintetlenül maradtak, ezek a tápcsatlakozó, a főkapcsoló illetve a feszültség szintek előállításáért felelős IC-k – egy 5V-os PTH08T231WAD DC-DC konverter, és egy 3.3V-os TS1117BCW LDO.

A lemez négy sarkán található hármas csatlakozó csoportok a szervomotorok számára lettek kialakítva, ezek szintén léteztek már az eredeti modellben is. A három csatlakozó mindegyike a robot lábának egy-egy csuklójához tartozik.



7. ábra: Babics Mátyás által készített NYÁK

2.2.3 Nyomtatott áramkör módosítása

A módosítás során felkerült a két új szenzor csatlakozási pontja, a LIDAR számára a csatlakozó, a BNO055 fejlesztőkártyának pedig megfelelő furatok a felforrasztáshoz. Több csatlakozó viszont eltávolításra került, egyrészt mert nem voltak használva, másrészt mert kellett a hely az előbb említett alkatrészeknek.

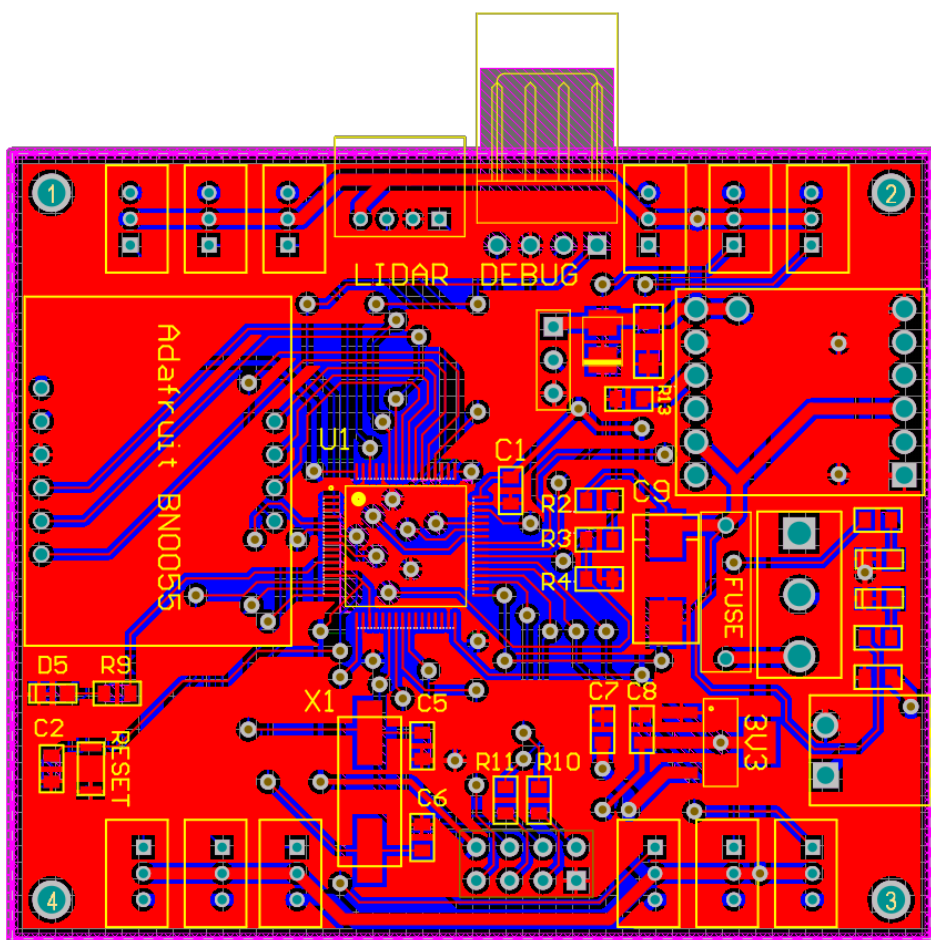
Megváltozott az ST chip programozására használt csatlakozó helye is, amely DEBUG felirattal a NYÁK felső szélére került. Ez a módosítás azért volt fontos, mert a LIDAR a nyomtatott áramkör fölött helyezkedik el, így a csatlakozó az eredeti pozíciójában, az áramkör közepén, nehezen lenne elérhető.

Újítként felkerült egy olvadóbiztosíték is FUSE felirattal. Ennek a szerepe természetesen a túláram védelem, amely komoly jelentőséggel bír, mert a tápként használt Lipo akkumulátorok elegendően nagy áram leadására képesek ahhoz, hogy komoly kárt tegyenek az áramkörben.

Az ST mikrovezérlő már az eredeti NYÁK-on is szerepelt, és bár a helye nem változott (továbbra is a lemez közepén található), kimeneteit az újításoknak megfelelően újra kellett huzaloznom.

Az ESP-01 chipnek kialakított 4x2-es csatlakozó is létezett már az eredeti NYÁK-on, azonban az új tervben meg lett fordítva. Erre azért volt szükség, mert az ESP-01 úgy van kialakítva, hogy a lábak a kártya egyik végén találhatóak, így az eredeti megoldásban a kártya a nyomtatott áramkör felett helyezkedett el. Az új terv szerint azonban a NYÁK fölé fog kerülni még egy LIDAR szenzor is, ezért az ESP-01 inkább meg lett fordítva, így az antennája nem a nyomtatott áramkör belseje felé néz, hanem lelóg a lemeztől az ellenkező irányban.

A 8. ábra mutatja az elkészült tervet felülnézetből. A nyomtatott áramkör egyoldalú, vagyis csak az egyik oldalára kerültek alkatrészek.



8. ábra: Nyomtatott áramkör terv

3 Felhasznált szoftver technológiák

A robot szoftverében kihasználtam, hogy léteznek könnyen elérhető, ingyenes és jól kiforrott technológiák, könyvtárak, és egyéb megoldások, amelyekkel kiváltható, hogy kézzel kelljen újra leimplementálnom egy programot, amit egyébként fejlesztők ezrei már széleskörben használnak. Ez a fejezet ismerteti a robotban használt technológiákat.

3.1 ROS

A Robot Operating System (ROS) egy olyan keretrendszer, amelyet robotokra terveztek [8]. Számos eszközzel és könyvtárral van felszerelve, amelyek egy egységesen kezelhető interfészt nyújtanak eltérő robotok és platformok között.

A ROS szervezet azonban ennél messzemenőbb tervekkel is rendelkezik. Nemcsak támogatást szeretne nyújtani saját eszközeivel, hanem egy olyan közösséget formálni, amely az egész világra kiterjedően képes megosztani tapasztalatait és eredményeit egymással. Emiatt számtalan könyvtár elérhető az interneten, ezek a legkülönbözőbb eszközöket tartalmazzák.

A robot fejlesztése során főbb célom volt, hogy a robotot beintegráljam a ROS rendszerbe a megfelelő szoftver részek elkészítésével. Ehhez a ROS alapkönyvtárain kívül három kiemelendő könyvtárat használtam: ROSSerial, RVIZ és hector_slam.

3.1.1 ROS működése

A ROS egy egyszerű küldő-feliratkozó (publisher-subscriber) mintát alkalmaz. A rendszer csomópontokból (node) áll, ezek tulajdonképpen egy programnak, vagy programrésznek felelnek meg, és általában egy funkcionalitást valósítanak meg. A csomópontok csatornákon (topic) tudnak egymással kommunikálni, ez a küldő-feliratkozó minta alapja. A csatornák rögzített adattípussal rendelkeznek és a csomópontokkal való kapcsolatot a háttérben futó úgynevezett "mag" (roscore) folyamat hozza létre, a csomópontban megadott információk alapján.

3.1.2 Beágyazott környezet - ROSSerial

A ROS rendszer használata arra a feltételezésre épült, hogy képesek vagyunk ROS csomópontot futtatni a roboton, amely valamilyen módon képes kapcsolatot teremteni a PC-n futó egyéb csomópontokkal.

A ROSSerial egy protokoll, ami a standard ROS üzenetek csomagolójaként szolgál, lehetővé téve, hogy a ROS rendszer beágyazott rendszereket is magába foglaljon [9]. A ROSSerial segítségével ROS üzeneteket tudunk átküldeni soros porton, vagy hálózati socket-eken.

Tehát a ROSSerial alkalmas arra, hogy megteremtse az interfészt a ROS csomópont és az alacsonyabb szintű beágyazott kód között.

3.1.2.1 ROS csomópont beintegrálása

A ROS könyvtárak összefordítására során a ROS alapkönyvtárak legenerálják a ROS működéséhez szükséges kódrészleteket. A ROSSerial pedig olyan kódrészleteket generál, amelyeket beágyazott eszközön tudunk futtatni. Ha jól raktuk össze a ROS projektet, akkor fordítás után egy olyan kódhalmazt kapunk, amit bele tudunk illeszteni a roboton futó szoftverbe. Az alapvető ROS kódokat egyfajta driver-nek lehet tekinteni.

A kód bemásolása után fontos megírni a ROSSerial lefele mutató interfészét. Ez mindössze annyit jelent, hogy megírjuk a 1. kódrészletben is szemléltetett read és write függvényeket.

```

int read()
{
    char a;

    if (esp_Wifi == nullptr)
        return -1;

    if (esp_Wifi->GetRos_inData().pop(a))
    {
        return a;
    }
    return -1;
}

void write(uint8_t* data, int length)
{
    for(uint16_t looper = 0; looper < length; looper++)
    {
        esp_Wifi->GetRos_outData().push((char&)data[looper]);
    }
}

```

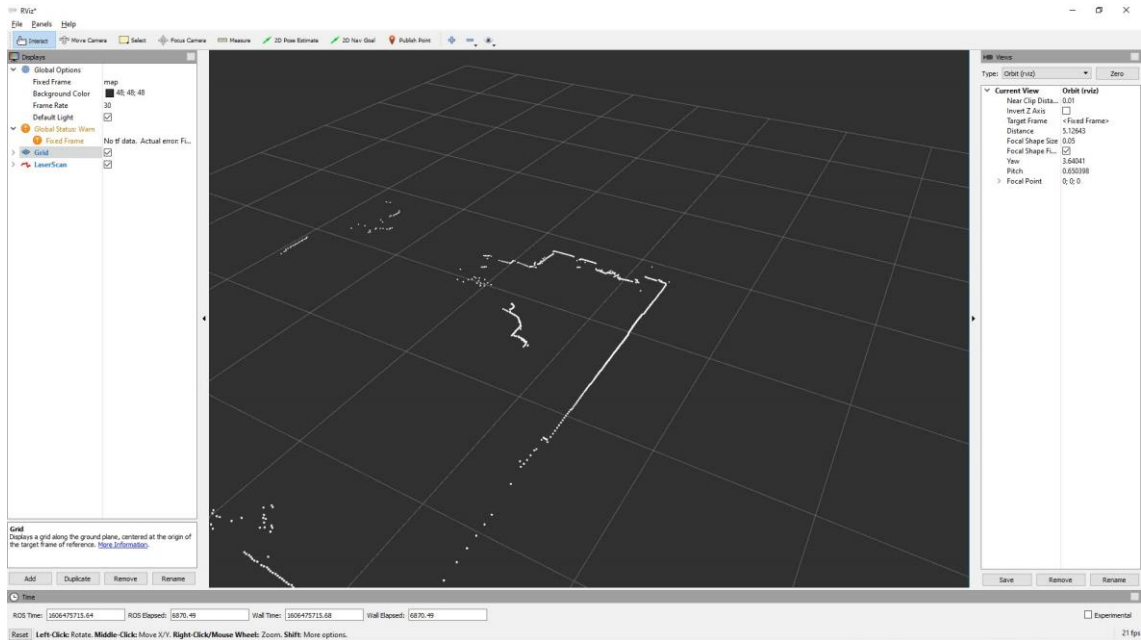
1. kódrészlet: ROSSerial interfésze

Ha biztosítottuk a kapcsolatot a két réteg között az előző bekezdésben leírt módon, akkor a ROS rendszert sikeren beintegráltuk. Ezután a ROS csomópontot tetszőlegesen leimplementálhatjuk, a számunkra elvárt működés szerint.

3.1.3 3D szimulációs eszközök - RVIZ

A ROS egyik leginkább közismert eszköze az RVIZ, amely nem más, mint egy 3D-s szimulációs eszköz [10]. A RVIZ tekinthető a ROS egy alapkönyvtárának, hiszen a ROS telepítésekor automatikusan az RVIZ is települ, hacsak szándékosan máshogy nem rendelkezünk.

Az RVIZ ugyanúgy működik, mint bármelyik másik ROS csomópont, elindítása után értesül az elérhető csatornákról, és ha megfelelő típusú adat érkezik egy adott csatornán, akkor azt képes megjeleníteni is (lásd: 9. ábra).



9. ábra: LIDAR scan RVIZ virtuális megjelenítőben ábrázolva

3.1.4 Térképező algoritmusok – hector_slam

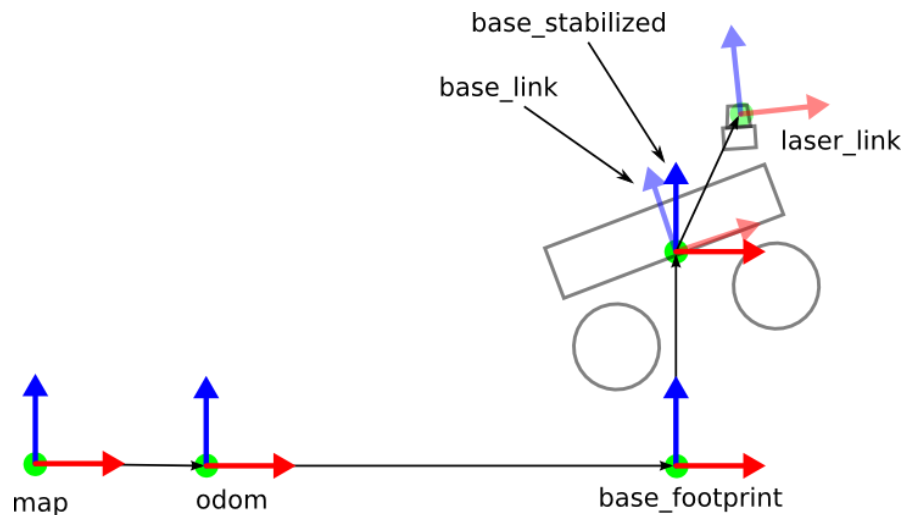
A mobil robotok világában a környezetről való információgyűjtés egy gyakran felbukkanó kérdés. Az egyik elterjedt megközelítés a térképkészítés. Az elnevezés magáért beszél, a cél, hogy a robot a környezetében lévő akadályokat és a szabad, mozgásra alkalmas területeket egymástól megkülönböztessük. Az így elkészített térkép segítségével már képesek vagyunk navigálni. A térképkészítés problémájára többféle algoritmus is létezik, ezek közül az egyik gyakorta előforduló megoldás a SLAM.

3.1.4.1 SLAM – Simultaneous Localization And Mapping

A SLAM, vagyis az egyidejű lokalizáció és térképezés, egy olyan algoritmus, amely segítségével ismeretlen környezetben vagyunk képesek térképet készíteni, miközben az ágensünk pozícióját is nyomon követjük [13].

3.1.4.2 hector_slam

A SLAM algoritmus használatára, elterjedtsége miatt, jó néhány ROS könyvtár létezik, ezek közül az egyik a hector_slam [11]. A hector_slam a beérkező LIDAR jelek alapján készíti a térképet, ezt opcionálisan kiegészíthetjük egyéb pozíció adatokkal is, például egy IMU segítségével.



10. ábra: hector_slam koordináta rendszerek

A 10. ábrán láthatóak a hector_slam által használt koordináta rendszerek. Ezekkel a koordináta rendszerekkel tudjuk leírni a robot fizikai felépítését, például a base_link jelenti a robot középpontját. Ezeket a koordináta rendszereket tetszőlegesen összevonhatjuk, ha nem akarjuk megkülönböztetni őket. Például a base_stabilized koordináta rendszerre csak akkor van szükség, ha a feltételezzük, hogy a robot képes a térkép síkjával nem megegyező síkban is forogni (roll/pitch – hajlás/bólintás). A LIDAR szenzor koordináta rendszerét a laser_link adja meg, ezt egy statikus linkkel kapcsoljuk a robot középponti eleméhez, a base_link-hez. Tehát a laser_link együtt fog mozogni a base_link-el, úgy mintha a szenzor a robot vázának meghatározott részére lenne erősítve.

A hector_slam algoritmus eredményét meg is tudjuk jeleníteni, például RVIZ segítségével. A 11. ábrán látható a hector_slam működése, a kép bal alsó sarkában látható, hogy az eszközt kézben viszik körbe egy felépített labirintusban, az pedig mérések segítségével meghatározza, hogy hol vannak a labirintus falai, illetve, hogy maga az eszköz hol található az útvesztőn belül.



11. ábra: hector_slam RoboCup 2011 [17]

3.2 FreeRTOS

A FreeRTOS egy ingyenesen felhasználható beágyazott rendszerekre tervezett operációs rendszer, amely számos széleskörűen használt, operációs rendszerekre jellemző szolgáltatást biztosít [12]. A felhasználó tetszés szerint tudja bekonfigurálni, hogy ezen szolgáltatások közül melyeket szeretné használni a szoftverében. A következő fejezetekben egy rövid összefoglalás található, a FreeRTOS legfontosabb elemeit, és a robotban alkalmazott megoldásokat pedig a 4.2 fejezetben mutatom be részletesebben.

3.2.1 Taszkok és ütemezés

A FreeRTOS, mint a legtöbb operációs rendszer, alapvető építő elemei a taszkok. Egy taszk általában egy jól egységbezárható feladatot lát el, ezen kívül az ütemezés szempontjából is fontos szerepet játszik, ugyanis az operációs rendszer mindig egyszerre csak egy taszkot enged futni (egy processzor mag esetén). Az, hogy mi alapján kap egy taszk futási jogot eltérő lehet az alkalmazott operációs rendszertől függően. A FreeRTOS preemptív ütemezőt és prioritásokkal ellátott taszkokat használ, ami azt jelenti, hogy adott időközönként (FreeRTOS tick) a rendszer megvizsgálja, hogy melyik az a legmagasabb prioritású taszk, amelyik futásra kész állapotban van, és annak fog futási jogot adni. Mivel az ütemező preemptív, ezért a kontextus váltás akár úgy is megtörténhet, hogy az éppen futó, de alacsonyabb prioritású taszk még nem végzett, vagyis egy magasabb prioritású taszk képes megszakítani egy alacsonyabb prioritású taszk futását. Így tudjuk biztosítani, hogy mindig az a taszk fog futni, amelyik a legfontosabb feladatot látja el.

3.2.2 Taszkok közötti kommunikáció

A FreeRTOS többféle megoldással is támogatja a taszkok közötti kommunikációt. Ezen megoldások közül csak kettőt emelnék ki: stream (adatfolyam) buffer és message (üzenet) buffer. A két szoftverelem sok tekintetben hasonló, ami abból ered, hogy a message buffer a stream buffer leszármazottja. Mindkét elem, mint a legtöbb taszkok közötti kommunikációt megvalósító elem, kezeli a közös erőforrás problémát, és a megfelelő interfésszel nem csak taszkok, hanem megszakítások is tudják használni őket.

A stream buffer megközelítése arra alapoz, hogy a benne tárolt adatot bájtfolynak tekinti. Ennek a megoldásnak az előnye, hogy egyszerű a küldés és a fogadás, mindössze egy pointert és egy méretet kell megadnunk argumentumként. A robot szoftverében stream buffert az UART vonalak kezelésénél alkalmaztam, hiszen az UART maga is bájtok soraként látja az elküldött/fogadott adatot, ráadásul a stream buffer véd a közös erőforrás probléma ellen is, és megszakításból is kezelhető.

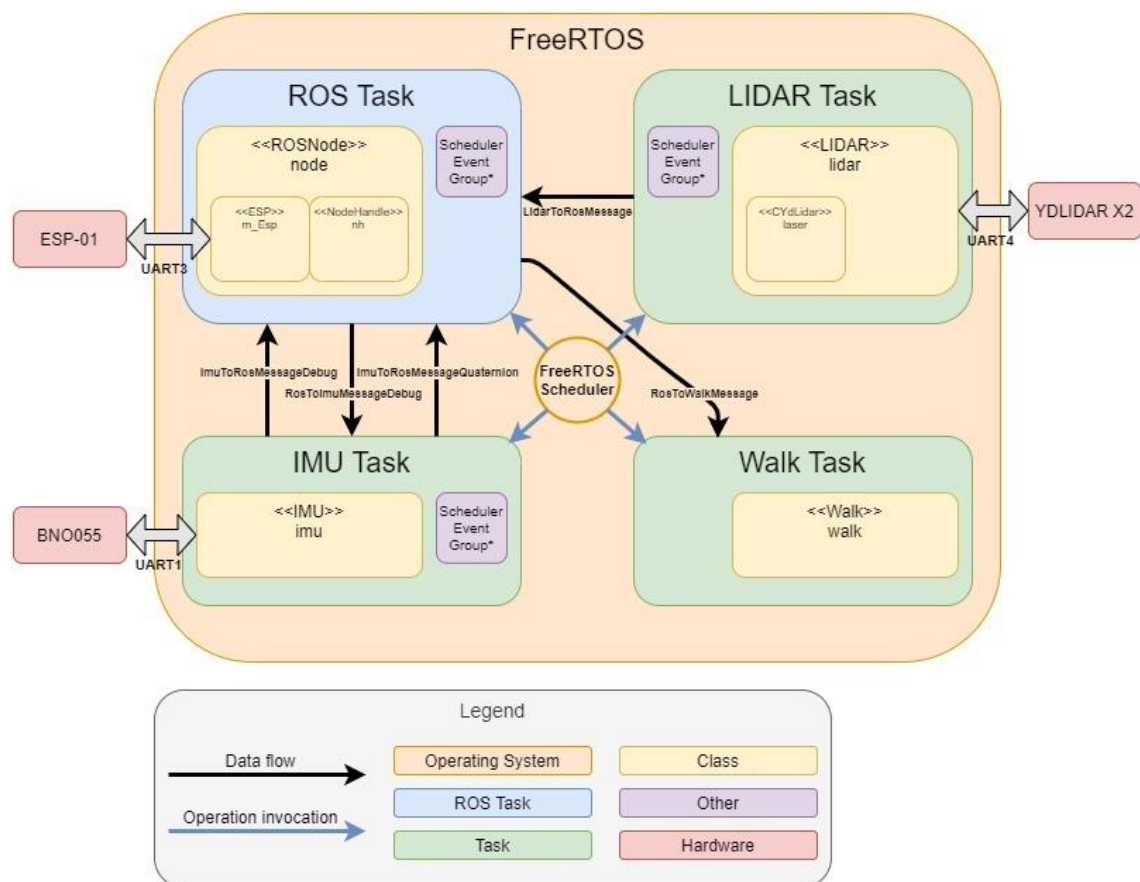
A message buffer, bár a stream bufferre alapoz, annál komplexebb adatstruktúrák kezelésére is alkalmas. A tároló nem bájtokat vár, hanem tetszőlegesen hosszú, akár egymástól eltérő hosszúságú adatokat. A kiolvasásnál az együtt beírt adatbájtokat egyszerre tudjuk kivenni. A robotban ezt a megoldást például a szenzor adatoknál tudtam kihasználni, ugyanis azok a szenzortól függő, de előre specifikált struktúrában érkeznek.

4 Beágyazott szoftver

Ez a fejezet a robot szoftverének felépítéséről, a kódban alkalmazott megoldások ismertetéséről szól.

4.1 Áttekintés

A szoftver objektumorientált szemléletmóddal íródott C++ nyelven, számos könyvtárral kiegészítve.



12. ábra Szoftver architektúra

A szoftver működését taszkok valósítják meg, amiket a FreeRTOS operációs rendszer ütemez. A taszkoknál alkalmazott implementációról a 4.2.2 fejezetben található információ. A rendszer öt taszkból áll, ezek a következők:

- ROS Task – Ez a taszk felelős a ROS csomópont futtatásáért, ami magába foglalja az ESP-01 chip kezelőfüggvények végrehajtását is.

Mivel ebben a taszkban kerül sor a wifi kommunikációra, ezért ez a taszk folyamatosan fut, alacsony prioritással.

- LIDAR Task – A LIDAR szenzor által küldött jelek feldolgozását és továbbítását hajtja végre. A szenzorkezelő taszkok magas prioritással futnak.
- IMU Task – Az IMU szenzortól lekérdezi az aktuális pozíció adatokat. A szenzorkezelő taszkok magas prioritással futnak.
- Walk Taks – A mozgásért felelős kódrészeket hajtja végre. A taszk közepes prioritással fut.
- Adc Task – Ez a taszk mindössze azért felelős, hogy ellenőrzi az akkumulátor feszültség szintjét, és villogtat egy LED-et ha kezd lemerülni a tápforrás.

4.2 FreeRTOS

Ez a fejezet mutatja be a robot szoftverében használt FreeRTOS operációsrendszerhez köthető részeket.

4.2.1 Konfiguráció

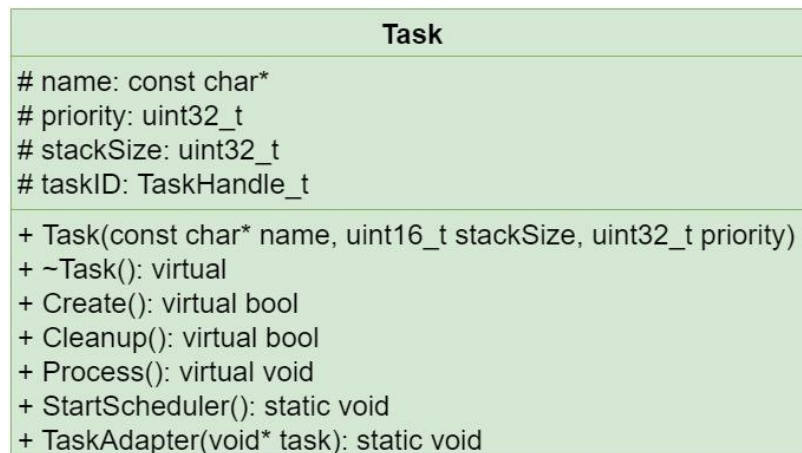
A FreeRTOS konfigurálást egy speciális header fájlban, a FreeRTOSConfig.h-ban tehetjük meg. Itt számos opciót adhatunk meg, ezek közül a legfontosabb elemeket emelném ki:

- A configTICK_RATE_HZ makró segítségével megadhatjuk az ütemező lefutási gyakoriságát – ez a jelenlegi konfigurációban 1 ms-re van állítva.
- Be tudjuk állítani, hogy hány prioritás szint létezzen – külön a taszkokra és külön a megszakításokra.
- Engedélyezni tudjuk, hogy bizonyos elemek, például a különböző tárolók, megszakításokból is tudjanak működni.
- Be tudjuk állítani, hogy szeretnénk taszk késleltetést (TaskDelay) használni.
- A configCHECK_FOR_STACK_OVERFLOW engedélyezi a stack memória túlcsoordulását ellenőrző diagnosztikát.

- A `configUSE_MALLOC_FAILED_HOOK` engedélyezi, hogy egy hook függvény fusson le, ha memória hiányában sikertelen a dinamikus memóriafoglalás.

4.2.2 Taszkok

A robot szoftverében használt taszkok mind egy-egy leszármazottjai egy `Task` nevű ősosztálynak. Ennek az osztálynak az a szerepe, hogy egy csomagolót biztosít a FreeRTOS-os taszkok számára.



13. ábra: Task osztálydiagramja

4.2.2.1 FreeRTOS taszkok létrehozása

A FreeRTOS taszkok létrehozásánál két függvény játszik kiemelkedő szerepet. Először létre kell hoznunk a taszk objektumot, erre szolgál a `xTaskCreate` függvény, ha létrehoztuk az összes taszkot, amit szerettünk volna, akkor el kell indítani az ütemezőt, ezt pedig a `vTaskStartScheduler` metódussal tehetjük meg. Amíg az ütemezőt nem indítottuk el addig nem fog egy taszk sem futni.

Az `xTaskCreate` függvény hat paramétert vár, ezek segítségével megadhatjuk a taszk nevét, a rendelkezésére álló stack memória méretét, a prioritását, illetve egy referenciát, ahova a `xTaskCreate` függvény beírja a lefutása során a taszk azonosítóját. A maradék kettő paraméter egyike egy függvény pointer, itt tudjuk megadni azt a metódust, ami az ütemező által majd mindig meghívásra kerül. Tehát itt azt a függvényt adjuk meg, ami a taszk tényleges feladatát hajtja végre, például egy szenzort kezelő taszk esetében ez a mérési eredmények lekérdezése lehet. Ezt a függvényt úgy kell elkészítenünk, hogy soha ne térjen vissza. A `xTaskCreate` függvény utolsó

argumentuma pedig arra használható, hogy tetszőleges paramétert tudunk átadni vele a taszknak.

Az `xTaskCreate` függvényt a `Task` osztály `Create` függvénye fogja meghívni. A név, a prioritás, a stack méret, illetve az azonosító számára a `Task` osztály biztosít tagváltozókat. Az `xTaskCreate` függvényt pointer argumentumának a `Task` osztály `TaskAdapter` statikus függvényét adjuk meg, ez fogja meghívni a `Task` osztály `Process` függvényét, ami implementálja a taszk feladatát.

A `TaskAdapter` statikus függvény, vagyis az összes `Task` típusú objektumra egy közös `TaskAdapter` függvény létezik. A `TaskAdapter` argumentumként vár egy `Task` objektumra mutató pointert, innen fogja tudni, hogy melyik objektum `Process` függvényét kell meghívnia. Emlékezzünk vissza, hogy az `xTaskCreate` metódusnak van egy argumentuma, amin keresztül tetszőleges paraméter átadható. Amikor a `Create` függvény meghívja az `xTaskCreate` függvényt, akkor paraméterként átadja neki a `this` mutatót, vagyis effektíven átadja a `Task` objektum mutatóját. Ezt a mutatót fogja megkapni a `TaskAdapter` is, és így tudja meghívni az adott taszk `Process` függvényét.

```
bool Task::Create()
{
    if (xTaskCreate(TaskAdapter, name, stackSize, this, priority,
&taskID) != pdPASS)
        return false;
    if (taskID == NULL)
        return false;
    return true;
}

void Task::TaskAdapter(void* task)
{
    Task* t = static_cast<Task*>(task);
    t->Process();
    t->Cleanup();
    configASSERT(!"Returned from task's Process!");
}
```

2. kódrészlet: Create és TaskAdapter függvények

Az ütemező elindítására a `Task` osztály `StartScheduler` függvényét használhatjuk. Ez a metódus szintén statikus, hiszen nem egy adott taszokra indítjuk el az ütemezést, hanem mindegyikre. A `StartScheduler` egyszerűen csak meghívja a `FreeRTOS vTaskStartScheduler` függvényét.

4.2.2.2 A Task osztály leszármazottjai

Az előző fejezet alapján láthatjuk, hogy a Task osztály kiváló megoldást biztosít a FreeRTOS taszkok létrehozására, azonban látható az is, hogy a Process függvényt nem tudjuk ezen a szinten implementálni, hiszen ez a kód minden taszkra eltérő kell legyen, márpedig Process függvényből csak egy darab van. Ezt a problémát leszármazottak létrehozásával tudjuk áthidalni.

A Task osztály leszármazottjaiban két függvényt tudunk módosítani. Egyrészt ki tudjuk egészíteni a konstruktort tetszőleges argumentummal, ami arra lesz jó, hogy a konstruktor segítségével képesek leszünk átadni minden egyéb FreeRTOS elemet: Event Group-okat, Stream/Message Buffer-eket. Másrészt tetszőleges módon tudjuk implementálni a Process függvényt, így az mindig az adott taszknak megfelelő feladatot látja el.

4.2.2.3 A Task osztály leszármazottjainak használata

A Task osztály leszármazottjait a következő módon lehet használni: Először hozzunk létre egy tetszőleges osztály példányt, ezek a robot szoftverében a megfelelő leszármazottra mutató globális pointerek. A Task példányok azért lettek pointerként definiálva, mert majd a heap memóriában fog helyet foglalni számukra a main függvény.

```

/* FreeRTOS task handlers */
task::TaskROS* taskROS;
task::TaskIMU* taskIMU;
task::TaskLIDAR* taskLIDAR;
task::TaskWalk* taskWalk;
task::TaskADC* taskADC;

int main()
{
...
    /* FreeRTOS task handlers */
    taskROS = new task::TaskROS(...);
    taskIMU = new task::TaskIMU (...);
    taskLIDAR = new task::TaskLIDAR(...);
    taskWalk = new task::TaskWalk("TaskWalk", 512, 3);
    taskADC = new task::TaskADC("TaskADC", 128, 3);

    /* Creating tasks */
    taskROS->Create();
    taskIMU->Create();
    taskLIDAR->Create();
    taskWalk->Create();
    taskADC->Create();

...
}

```

3. kódrészlet: Task példányok létrehozása

Következő lépésként példányosítsunk egy-egy objektumot a megfelelő globális Task leszármazott pointereknek a main függvényben a new operátor segítségével.

Megjegyzendő, hogy a FreeRTOS elemeket, az operációsrendszer működéséből eredően, dinamikusan foglaljuk, tehát a fordítási időben minden ilyen változó NULL pointer értékű. Ezért fontos, hogy a taszkok létrehozását csak akkor tegyük meg, ha a többi elem már érvényes értékkel létre lett hozva, így elkerülhető, hogy a taszkok konstruktorai NULL pointert kapjanak paraméterként. Emiatt a taszkokat a 3. kódrészletnek megfelelően dinamikus memórafoglalással hozzuk létre.

Ezután futtassuk a már érvényes osztálypéldányra mutató taszk pointerek Create függvényét. A Create függvény lefutása után a taszkok készen állnak a működésre, így már csak az ütemező elindítása van hátra, amit a StartScheduler statikus függvény meghívásával tehetünk meg.

4.2.3 Esemény kezelés

A FreeRTOS egyik szolgáltatása az Esemény Csoport, vagy angol nevén Event Group. Ezen szoftverelem segítségével egy taszk képes jelezni, hogy valami történt, ami fontos lehet egy másik taszk számára, például valamilyen feladat elvégzésre került.

A robot szoftverében mindössze egy darab Event Group létezik, ennek a neve SchedulerEventGroup. Az esemény csoportra mutató pointert mindegyik taszk a konstruktorában kapja meg. A rendelkezésre álló esemény bitek közül 6 van felhasználva:

`SCHEDULEREVENT_WIFI_BIT` – Ezt a bitet az ESP-01 chiphez tartozó UART vonal kezelő megszakítása állítja be, akkor, ha adat érkezett a soros vonalon, és a ROS node taszkja vizsgálja. Így tudjuk biztosítani, hogyha üzenet érkezik a wifi csatornán, akkor az feldolgozásra is kerül.

`SCHEDULEREVENT_LIDAR_BIT` – Ezt a bitet a LIDAR szenzor kezelését végző taszk állítja be, ha friss mérési adat áll rendelkezésre, és a ROS csomópont futtatását végző taszk vizsgálja, ha be van állítva a bit, akkor továbbítja a mérési adatokat.

`SCHEDULEREVENT_IMU_BIT` – Az előző bithez hasonló a funkcionalitása, annyi különbséggel, hogy az IMU szenzort kezelő taszk állítja be.

`SCHEDULEREVENT_WAIT_ROS_INIT` – Ezt a bitet a ROS taszk állítja be. Mivel a ROS csomópont felelős a wifi kommunikáció vezérléséért, ezért a szenzoroknak felesleges adatot küldeniük, amíg a ROS node nem hajtotta végre az inicializáló lépéseit, mert nem tudunk mit kezdeni az adatokkal. Ezért a szenzorok taszkjai a saját inicializáló lépéseik után nem kezdik el a mérési adatok előállítását, hanem várakoznak, amíg a ROS csomópont sikeresen elindul.

`SCHEDULEREVENT_IMU_CALIBRATION_BIT` – Ezt a bitet az IMU taszk állítja be, ha sikerült a BNO055 szenzor kalibrálása (lásd 4.4.2 fejezet), állapotát a ROS taszk vizsgálja, és egy üzenetben jelez, hogy a kalibráció sikeres.

`SCHEDULEREVENT_IMU_CALIBRFAIL_BIT` – Ezt a bitet szintén az IMU taszk állítja be. Az IMU szenzort sikertelen kalibráció esetén újra indítjuk (4.4.2 fejezet), ilyen esetben a ROS taszk tájékoztató üzenetet fog küldeni.

4.2.4 Dinamikus memóiafoglalás

A FreeRTOS támogatást nyújt arra, hogy az operációs rendszer objektumait dinamikusan hozzuk létre. Ezt a memóiafoglalást a FreeRTOS saját heap memóriájában tehetjük meg. A heap memória kezelésére több megoldás is létezik, ezek közül úgy tudunk választani, hogy a megfelelő számú heap_x.c fájl adjuk hozzá a projecthez. A FreeRTOS öt különböző heap kezelőt biztosít a felhasználónak, azonban a robot szoftverében ezek közül egyiket sem használtam.

A heap kezelésre Dave Nadler megoldását alkalmaztam, amelyet a DRNadler/FreeRTOS_helpers Github repository-ból tudtam letölteni. Ennek a megoldásnak az az előnye, hogy kezeli a C nyelv egyik alapértelmezett könyvtárában – newlib – található, dinamikus memóiafoglalásra használt malloc függvényét. Ha a C++ nyelvben használt new operátorral akarunk memóriát foglalni, márpedig úgy akarunk, akkor a malloc függvény fog futni a háttérben. Ha a FreeRTOS és a newlib függvényei nincsenek megfelelően, összehangoltan kezelve, akkor számos problémába futhatunk bele: például két külön heap jön létre, egy a FreeRTOS által kezelve, egy pedig a new operátor alatt, ezek a memóriák ráadásul könnyen össze is akadhatnak. Egy másik probléma, hogy a malloc önmagában nem szálbiztos, vagyis, ha a malloc futása közben történik kontextus váltás az memória korrupcióhoz vezethet. Szerencsére a szoftverben alkalmazott heap_useNewlib_ST heap kezelő ezeket a problémákat orvosolja.

A futás közben fellépő hibákra a FreeRTOS biztosít opcionális diagnosztikákat. Az egyik ilyen diagnosztika a vApplicationMallocFailedHook, ez a függvény akkor lesz meghívva, ha nincs elegendő memória a létrehozni kívánt objektumnak. A jelenlegi megoldásban ez a függvény nem csinál mást, csak egy üres végtelen ciklust futtat, ez debuggolásnál tud segítséget nyújtani.

Nem csak a heap memóriára létezik diagnosztika, hanem a stack-re is. Ez a vApplicationStackOverflowHook, és a stack memória túlsordulása esetén fut. Társához hasonlóan ez a függvény is csak egy végtelen ciklust futtat a jelenlegi megoldásban.

4.2.5 Task osztály leszármazottak diagramjai

| TaskROS |
|---|
| <ul style="list-style-type: none"> - node: ROSNode - SchedulerEventGroup: EventGroupHandle_t - ImuToRosMessageQuaternion: MessageBufferHandle_t - ImuToRosMessageDebug: MessageBufferHandle_t - RosToImuMessageDebug: MessageBufferHandle_t - LidarToRosMessage: MessageBufferHandle_t - RosToWalkMessage: MessageBufferHandle_t |
| + TaskROS(const char* name, uint16_t stackSize, uint32_t priority, EventGroupHandle_t& SchedulerEventGroup, MessageBufferHandle_t& ImuToRosMessageQuaternion, MessageBufferHandle_t& ImuToRosMessageDebug, MessageBufferHandle_t& RosToImuMessageDebug, MessageBufferHandle_t& LidarToRosMessage, MessageBufferHandle_t& RosToWalkMessage) + Process(): void |

14. ábra: TaskROS osztálydiagramja

| TaskLIDAR |
|--|
| <ul style="list-style-type: none"> - lidar: LIDAR - SchedulerEventGroup: EventGroupHandle_t - LidarToRosMessage: MessageBufferHandle_t |
| + TaskROS(const char* name, uint16_t stackSize, uint32_t priority, EventGroupHandle_t& SchedulerEventGroup, MessageBufferHandle_t& LidarToRosMessage) + Process(): void |

15. ábra: TaskLIDAR osztálydiagramja

| TaskIMU |
|--|
| <ul style="list-style-type: none"> - imu: IMU - SchedulerEventGroup: EventGroupHandle_t - ImuToRosMessageQuaternion: MessageBufferHandle_t - ImuToRosMessageDebug: MessageBufferHandle_t - RosToImuMessageDebug: MessageBufferHandle_t |
| + TaskROS(const char* name, uint16_t stackSize, uint32_t priority, EventGroupHandle_t& SchedulerEventGroup, MessageBufferHandle_t& ImuToRosMessageQuaternion, MessageBufferHandle_t& ImuToRosMessageDebug, MessageBufferHandle_t& RosToImuMessageDebug) + Process(): void |

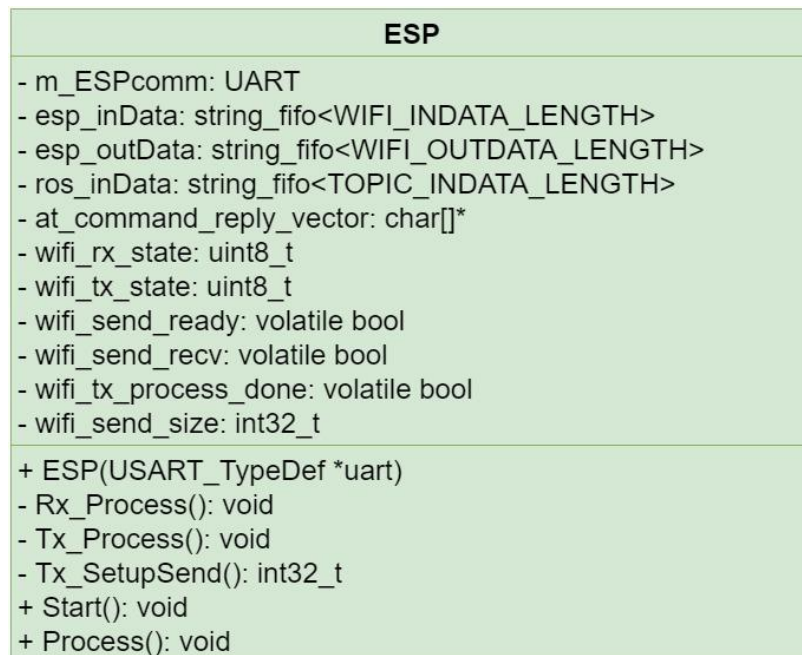
16. ábra: TaskIMU osztálydiagramja

| TaskWalk |
|--|
| <ul style="list-style-type: none"> - walk: Walk - walk_Command: WalkCommand - RosToWalkMessage: MessageBufferHandle_t |
| + TaskROS(const char* name, uint16_t stackSize, uint32_t priority, MessageBufferHandle_t& RosToWalkMessage) + Process(): void |

17. ábra: TaskWalk osztálydiagramja

4.3 Wifi kommunikáció – ESP-01 vezérlése

A robot képes TCP/IP protokollt használó wifi csatornán kommunikálni. Ezt egy külön hardver modul, egy ESP-01 típusú mikrovezérlő végzi, amely UART vonalon biztosít interfészt az ST mikrokontroller felé. Az ESP-01 chip kezeléséhez tartozó feladatokért egy külön osztály felel: ez az ESP osztály.



18. ábra: ESP részleges osztálydiagramja

Az ESP osztály főbb feladatai közé tartozik az ESP-01 chip felinicializálása a rendszer indulásakor, illetve, mivel egy kommunikációért felelős osztályról van szó, a fogadott illetve küldendő adatok kezelése.

4.3.1 AT parancsok

Az AT parancsok elsősorban modemek vezérlésre szolgálnak, azonban az ESP8266 chip (az ESP-01 board kontrollere) is definiál AT parancsokat, amik segítségével a wifi chip vezérelhető [7]. Ezek formátuma hasonlít a klasszikus AT parancsok felépítéséhez: A parancsok karakterláncok, jellemzően fix kezdő karakterekkel – "AT+", vagyis: attention – ezt követi a maga a parancs a megfelelő paraméterekkel, majd a záró karakterek: kocszi vissza és új sor.

A karakterláncokkal való műveletek támogatására készült egy string_fifo nevű tároló osztály. Ez az osztály annyiban tud többet egy FreeRTOS buffernél, amelyeket amúgy a szoftverben számos helyén alkalmaztam, hogy biztosít egy compare függvényt az interfészén, amely segítségével adott karakterlánc előfordulását kereshetjük a tárolóban.

4.3.2 ESP-01 inicializálása

Az ESP-01 induláskori felkonfigurálását az ESP osztály Start függvénye végzi. Az inicializálás lépései a következők:

1. Hard reset előállítása az ESP-01 részére a megfelelő pin földre húzásával, majd várakozás a dokumentációban meghatározott induláshoz szükséges ideig.
2. Az UART kommunikáció ellenőrzése. Ehhez az "AT" parancsot küldjük, amire az ESP-01 fix karakterlánccal válaszol, ha ez megérkezik, akkor megbízhatónak tekinthetjük a soros kapcsolatot.
3. Az "ATE0" paranccsal kikapcsoljuk, hogy az ESP-01 visszaküldje a fogadott parancsokat, mivel ezekre nincs szükségünk.
4. Beállítjuk az ESP-01 működési módját az "AT+CWMODE_CUR=2" paranccsal SoftAPI-ra, így tudjuk elérni, hogy az eszköz saját hálózatot hozzon létre.
5. Utolsó lépésként a TCP kapcsolat létrehozását kell biztosítani. A robot, a hálózat szempontjából host, de a TCP kapcsolatban kliensként vesz részt. Ez a megoldás azért előnyös, mert a robot hálózatára csak egy eszköz fog csatlakozni (egy tetszőleges PC), aminek így mindig fix IP címe lesz. A robot periodikusan próbálkozik a TCP kapcsolat felépítésével, amíg az sikeresen meg nem történik, ehhez pedig a "AT+CIPSTART=\"TCP\", \"192.168.4.2\", 11411" parancsot kell küldeni az ESP-01 eszköznek. A parancs paramétereinek között szerepel a kapcsolat típusa, ami TCP, a hálózatra csatlakozott eszköz fix IP címe és portja.

A Start függvény a TCP kapcsolat sikeres felépülését követően tér vissza, ezzel befejezve az inicializációs lépéseket. A robot ezt követően működő wifi kommunikációs csatornával rendelkezik.

4.3.3 Adat fogadás

Az ESP osztály Process függvénye végzi a beérkezett és küldendő adatok kezelését. Ez a függvény periodikusan meghívásra kerül a ROS node által. A Process függvény két másik függvényt hív meg, ezek közül az egyik felelős a beérkezett adatok kezeléséért, ez a tagfüggvény az Rx_Process névre hallgat.

Az Rx_Process függvény első lépéseként átmásolja az UART osztály megfelelő tárolójából a beérkezett adatokat az esp_inData bufferbe. Ez a buffer string_fifo típusú, tehát képes támogatni az ESP-01 AT parancsainak a feldolgozását. Az Rx_Process

függvény a `string_fifo compare` függvényének a segítségével megvizsgálja, hogy van-e a beérkezett adatok közt olyan karakterlánc, ami az öt számunkra érdekes lehetőség egyikével megegyezik-e. Ha igen, akkor ezek közül kiválasztja azt, amelyik a legelől található. Ha nem talál olyan karakterláncot, ami számunkra érdekes, akkor a függvény egyszerűen visszatér.

A számunkra érdekes karakterláncokat az `at_command_reply_vector` string tömb tárolja, ennek elemei a következők:

A "CONNECT" illetve "CLOSED" karakterláncot tartalmazó üzenetek a wifi kapcsolat állapotára adnak visszajelzést. Ha ilyen üzenet érkezik, akkor az ESP osztályon belül is eltároljuk ezt az információt, ugyanis, ha nincs kapcsolat, akkor adat küldéssel nem érdemes próbálkozni.

További két elem a ">", "SEND OK", ezek az üzenetek az adatküldésnél játszanak szerepet. Lásd 4.3.4 fejezet.

Az adatfogadás szempontjából a legfontosabb karakterlánc, ami előfordulhat a "+IPD,", ugyanis ez jelzi, hogy adat érkezett a wifi csatornán. A teljes AT parancs felépítése a következő: `+IPD,<len>:<data>`, ahol `len` az érkező adatok száma, `data` pedig maga az adat. Az `Rx_Process` függvény az adat feldolgozásakor ellenőrzi, hogy a méret mező valóban számokból áll-e, és ha az konzisztensnek tűnik, akkor az adat mező elemeit átmásolja az `ros_inData` tárolóba, amit a ROS csomópont tud majd olvasni. Ha a méret nem konzisztens adatokból áll (vagyis nem csak szám karakter van benne), akkor megszakítja a feldolgozást. Amire még figyelni kell a feldolgozásnál, hogy nem biztos, hogy a teljes üzenet rendelkezésre áll a feldolgozás megkezdésének pillanatában. Ezt a problémát nem-blokkoló várakozással tudjuk áthidalni.

4.3.4 Adatküldés

Az ESP osztály `Tx_Process` függvénye felelős az adatok küldésének kezeléséért. Ezt a függvényt is a periodikus `Process` függvény hívja meg társához, az `Rx_Process`-hez hasonlóan, annyi különbséggel, hogy a `Tx_Process` csak akkor lesz meghívva, ha van éppen működő wifi kapcsolat.

Az adatküldéshez az `"AT+CIPSEND=<length>"` parancsot tudjuk használni. A küldés folyamata több lépésből áll, amely során az ESP-01 több nyugtázó üzenetet is küld nekünk. A küldés lépései:

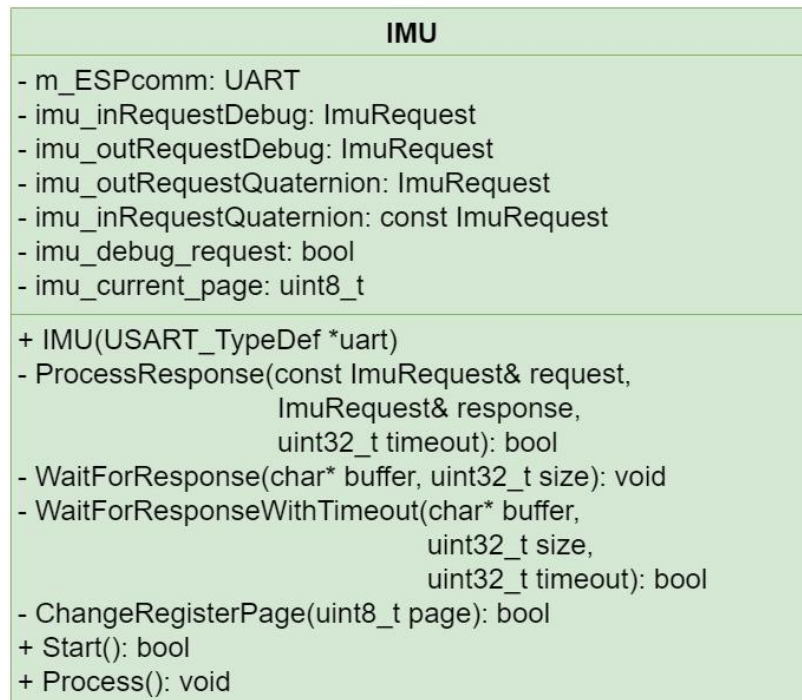
Először el kell küldenünk az AT+CIPSEND parancsot, amiben benne van az általunk elküldeni kívánt adat hossza. Az ESP-01 specifikációja alapján ez a hossz, maximum 2048 bájt lehet. Ezt a méretet érdemes kihasználni, mert optimálisabb feldolgozási időt kapunk, ha ugyanannyi adatot kevesebb nyugtázási ciklussal tudunk elküldeni. Az AT+CIPSEND parancs elküldéséért az Tx_SetupSend tagfüggvény felelős. A ROS csomópont az esp_outData tárolóba helyezi az elküldendő bájtokat, így a Tx_SetupSend az esp_outData mérete alapján határozza meg az elküldendő adat méretét. Az elküldendő adat mérete vagy a tároló aktuális mérete, vagy maximum 2048 bájt lesz. Ha a buffer 2048-nál több elemet tárol, akkor több ciklusban fognak elküldésre kerülni a tárolt elemek.

Az AT+CIPSEND parancs megérkezését az ESP-01 egy ">" üzenettel nyugtázza, így a Tx_Process addig nem-blokkoló módon várakozik, amíg ez a jelzés meg nem érkezik. Ha megtörtént a nyugtázás, akkor az esp_outData bufferből megkezdhetjük a küldést. Az UART kezelő osztály tárolója kisebb, mint a maximálisan kérhető adatküldés, ezért a másolás egy while ciklus segítségével kerül végrehajtásra, mindig csak annyi adatot helyezünk át egyszerre, ami még elfér.

Utolsó lépésként meg kell várni, hogy az ESP-01 jelezze, hogy a küldés a wifi csatornán is megtörtént. Ha ezt nem tesszük meg akkor előfordulhat, hogy túl gyorsan adunk adatot a wifi chipnek, ami egyszerűen eldobja azt, így a küldendő adat elveszik. Az ESP-01 egy "SEND OK" üzenettel jelzi, hogy az adattovábbítás sikeresen megtörtént, és készen áll az újabb kérés fogadására.

4.4 IMU szenzor – BNO055 vezérlése

A robot pozíciójáról egy BNO055 típusú IMU szenzor segítségével nyerhetünk információt. Ez az eszköz UART vonalon biztosít interfészt az ST mikrovezérlőnek. Az érzékelő kezeléséért egy IMU-nak keresztelt osztály felelős.



19. ábra: IMU részleges osztálydiagramja

Az IMU osztály főbb feladatai közé tartozik a BNO055 chip felinicializálása a rendszer indulásakor, illetve a mérési adatok periodikus lekérdezése és megfelelő formában való továbbítása.

4.4.1 BNO055 interfésze

A BNO055 szenzor vezérlését regiszterek írásával, olvasásával tehetjük meg. A regisztereket UART vonalon tudjuk elérni megfelelő formátumú üzenetek segítségével.

Az UART csatornán küldött, regiszterírást kérvényező üzenetek felépítése a következő:

- Az első bájt mindig állandó, 0xAA érték.
- A második bájt a kérés iránya, a 0x00 érték jelenti azt, hogy írásról van szó.
- A harmadik bájt az írni kívánt regiszter címét tartalmazza. Ha több regisztert akarunk írni egyszerre, akkor az első regiszter címe kerül ide.
- A negyedik helyen szerepel a méret, vagyis hogy a kezdőcímtől számítva hány helyet akarunk felülírni.
- A maradék bájtok pedig a mérettel megegyező számú adatot tartalmazzák.

Register write

Command:

| Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | | Byte (n+4) |
|------------|--------|----------|--------|--------|-------|------------|
| Start Byte | Write | Reg addr | Length | Data 1 | | Data n |
| 0xAA | 0x00 | <..> | <..> | <..> | | <..> |

Acknowledge Response:

| Byte 1 | Byte 2 |
|-----------------|---|
| Response Header | Status |
| 0xEE | 0x01: WRITE_SUCCESS 0x03: WRITE_FAIL 0x04: REGMAP_INVALID_ADDRESS 0x05: REGMAP_WRITE_DISABLED 0x06: WRONG_START_BYTE 0x07: BUS_OVER_RUN_ERROR 0x08: MAX_LENGTH_ERROR 0x09: MIN_LENGTH_ERROR 0x0A: RECEIVE_CHARACTER_TIMEOUT |

20. ábra: BNO055 regiszterírási kérés

Az UART csatornán küldött, regiszterolvasást kérvényező üzenetek felépítése a következő:

- Az első bájt mindig állandó, 0xAA érték.
- A második bájt a kérés iránya, a 0x01 érték jelenti azt, hogy olvasásról van szó.
- A harmadik bájt az olvasni kívánt regiszter címét tartalmazza. Ha több regisztert akarunk olvasni egyszerre, akkor az első regiszter címe kerül ide.
- A negyedik helyen szerepel a méret, vagyis hogy a kezdőcímtől számítva hány helyet akarunk kiolvasni.

Register read

Command:

| Byte 1 | Byte 2 | Byte 2 | Byte 3 |
|------------|--------|----------|--------|
| Start Byte | Read | Reg addr | Length |
| 0xAA | 0x01 | <.,> | <.,> |

Read Success Response:

| Byte 1 | Byte 2 | Byte 3 | | Byte (n+2) |
|--------------|--------|--------|--------|------------|
| ResponseByte | length | Data 1 | | Data n |
| 0xBB | <.,> | | | |

Read Failure or Acknowledge Response:

| Byte 1 | Byte 2 |
|-----------------|---|
| Response Header | Status |
| 0xEE | 0x02: READ_FAIL 0x04: REGMAP_INVALID_ADDRESS 0x05: REGMAP_WRITE_DISABLED 0x06: WRONG_START_BYTE 0x07: BUS_OVER_RUN_ERROR 0x08: MAX_LENGTH_ERROR 0x09: MIN_LENGTH_ERROR 0x0A: RECEIVE_CHARACTER_TIMEOUT |

21. ábra: BNO055 regiszterolvasási kérés

A regiszterek olvasása és írása hasonló felépítésű üzeneteket igényel, ezért létre lett hozva egy típus, amely ezeket kényelmesen tudja kezelni: ez az ImuRequest struktúra.

```
typedef struct __attribute__((__packed__)) ImuRequest{
    uint8_t header;
    uint8_t direction;
    uint8_t address;
    uint8_t length;
    uint8_t data[FREERTOS_IMUREQUEST_DATA_SIZE];
} Imu_request;
```

4. kódrészlet: ImuRequest struktúra

4.4.2 BNO055 inicializálása

Az IMU szenzor induláskori felkonfigurálásáért az IMU osztály Start függvénye felelős. A BNO055 inicializálása annyiban tér el a többi külső hardver elemétől, hogy a szenzort minden induláskor kalibrálni is kell. A kalibráció automatikusan megtörténik, a felhasználónak mindössze hat fix pozícióba kell mozgatnia az eszközt. A kalibráció állapotáról a szenzor CALIB_STAT nevű regisztere tartalmaz információt. Az eszköz inicializálását nem tekintjük befejezettnek, amíg ennek a regiszternek az értéke nem áll be a helyes értékre. A Start függvény adott időközönként (ez pár perc) vissza fog térni, és ha a kalibráció nem volt sikeres, akkor az egész inicializálási folyamatot előlről kezdi. A Start függvény a következő lépéseket végzi el:

1. Hard reset előállítása a BNO055 részére a megfelelő pin földre húzásával, majd várakozás a dokumentációban meghatározott induláshoz szükséges ideig.
2. Operációs mód lekérdezése az OPR_MODE regiszterből.
3. Ha az operációs mód nem egyenlő a CONFIGMODE értékkel – az eszköz csak ebben az állapotban engedélyezi a konfigurációs beállításokat – akkor átváltunk CONFIGMODE-ba. Az eszköz induláskor után CONFIGMODE állapotban van, ez csak extra védelem.
4. PWR_MODE regiszter beállítása Normal Mode-ba, ez a normális működési mód. További beállítási lehetőségek lennének a Low Power illetve Suspend mode, de ezeket nem használjuk.
5. OPR_MODE regiszter beállítása NDOF (nine degree of freedom – kilenc szabadságfokú mód) módba. Ilyenkor az eszköz mindhárom belső szenzort használja, és a mérési eredményeket fúziós eredményként adja a rendelkezésünkre.
6. Az utolsó lépés a kalibráció ellenőrzése, ezt a CALIB_STAT regiszter értékének olvasásával tehetjük meg. A Start függvény fél másodpercenként ellenőrzi a regiszter értékét, és csak akkor tér vissza igaz értékkel, ha a kalibráció sikeres. Ha nem sikeres a kalibráció, akkor egy idő után a Start függvény hamis értékkel visszatér.

4.4.3 Mérési adat lekérdezése

A BNO055 szenzor periodikus lekérdezéséért az IMU osztály Process függvénye felelős. A mérési adatot kvaternió [16] formában kérjük le, amit úgy tehetünk meg, hogy egy olvasási kérést küldünk, amiben címként a QUA_Data_w_LSB regisztert adjuk meg, méretnek pedig nyolcat. Így a válaszban megkapjuk a w,x,y,z koordináták értékeit 16 biten ábrázolva. A kérést egy ImuRequest struktúra objektumba rendezbe tudjuk kényelmesen előállítani, a Process függvény ezt fogja az UART vonalon elküldeni a szenzornak.

```
const ImuRequest imu_inRequestQuaternion = {  
    .header = IMUREQUEST_HEADER_START,  
    .direction = IMUREQUEST_DIRECTION_READ,  
    .address = QUA_DATA_W_LSB,  
    .length = 8u  
};
```

5. kódrészlet: Quaternion adat kérés

A kérés elküldése után az eredmény megvárása és feldolgozása van hátra. Ezt a minden kérésnél a `ProcessResponse` függvény teszi meg. A függvény paraméterként vár két `ImuRequest` referenciát. Egyet, ami az elküldött kérésre mutat, ez azért kell, hogy tudja, hogy milyen válaszra kell számítani, a másik referencia pedig arra az elemre kell mutasson, ahova az eredményt be tudja írni. Ezen kívül még egy argumentummal rendelkezik a függvény, ez egy time-out érték. Ha a time-out érték nulla, akkor a `ProcessResponse` időkorlát nélkül fog várakozni az eredményre, ha pedig ettől eltérő, akkor a megadott ideig. A mérési adat kérésénél mindig megadott időkorláttal várakozunk, ugyanis előfordulhat, hogy az UART-on nem érkezik meg az üzenet, és ilyenkor végtelen ciklusba kerülne a taszk.

Tehát egy kérés elküldése után a `ProcessResponse` vagy igaz értékkel fog visszatérni, ilyenkor az eredmény belekerül a második argumentumként megadott `ImuRequest`-be, vagy time-out miatt hamis értéket ad vissza, ilyen esetben újra el tudjuk küldeni a kérést, addig próbálkozva, amíg sikeresen választ nem kapunk.

4.4.4 Debug kérés

A robot fel lett készítve arra, hogy asztali számítógépen keresztül is lehessen az IMU szenzor regisztereit írni vagy olvasni. Ez a funkció elsősorban debuggolási célokra készült. A működésének a lényege, hogy a PC-n futó ROS node egy adott topic-ra küld egy kérést, ami formájában megegyezik az érzékelőnek küldendő kéréseknek. A beágyazott ROS csomópont ezt a kérést továbbítja az IMU osztálynak, ami pedig elküldi a BNO055 chipnek, pont úgy, mint egy sima kérést küldene. A kérés feldolgozása után kapott eredményt pedig visszaküldjük, hasonló formában, mint ahogy a kérést is.

Egy debug kérés az IMU osztály szempontjából egy egyszerű `ImuRequest` (a ROS csatornán érkezett/küldött adatot a ROS csomópont alakítja át, lásd később). A

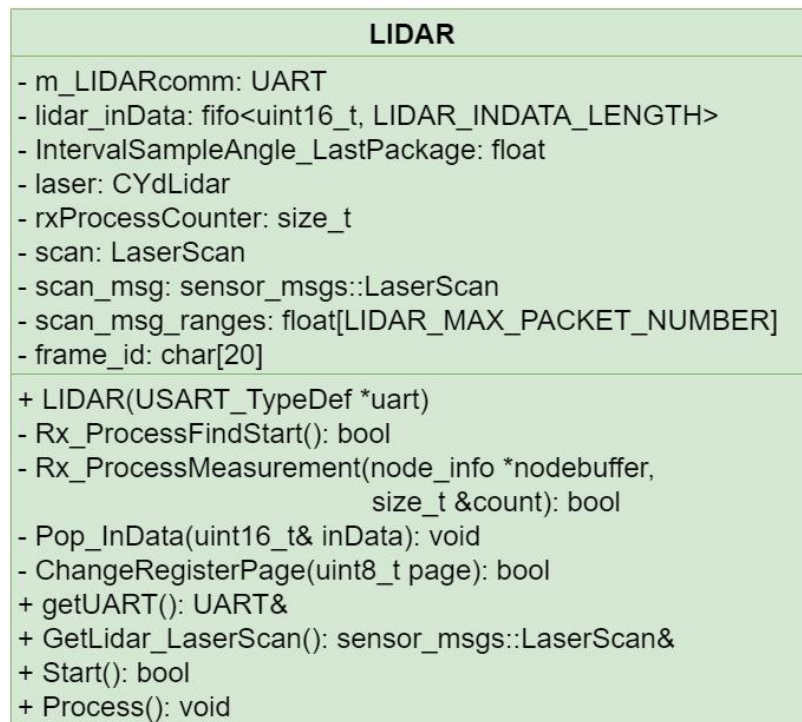
kérések feldolgozásáért a Process függvény felelős. Ha van aktív kérés, akkor a Process függvény a mérési adatok kérésénél használt módon jár el.

A debug kéréseknél előfordulhat, hogy egy olyan regisztert szeretnénk elérni, amelyik nem az alapértelmezett 0. regiszter oldalon van. Ez a probléma nem jön elő sem az eszköz inicializálásakor, sem a mérési adatok lekérdezésénél, mert mindegyik ott használt regiszter a 0. oldalon található. A nulladik regiszter ráadásul az alapértelmezetten kiválasztott oldal a hardver indulásakor.

Az IMU osztály működését tekintve túlnyomórészt a nulladik regiszter oldalt kell használnunk, csak kivételes esetben kell ettől eltérni, ezért regiszter oldalak kezelése úgy lett megoldva, hogy a Process függvény csak addig vált át az 1. oldalra, amíg az arra szóló debug kérést fel nem dolgozza. A feldolgozás végeztével pedig mindig visszavált a 0. oldalra. Az első oldalra szóló kérést úgy tudunk megadni, hogy az előre meghatározott 0xAA értékű header helyett 0xFF értéket adunk meg, ezt a Process függvény természetesen le fogja majd cserélni 0xAA értékre, amikor az UART csatornára küldi a kérést. Az oldalváltást pedig egy olyan kéréssel tudjuk elérni, ami az eszköz PAGE ID regiszterjét írja a kívánt oldal számát megadva adatként.

4.5 LIDAR szenzor – YDLIDAR X2 vezérlése

A robotot felszereltem egy YDLIDAR X2 típusú LIDAR érzékelővel. Ezt elsősorban akadályfelismerésre és térképkészítő algoritmusok támogatására használhatunk. A szenzor UART vonalon küld mérési adatokat, azonban a soros vonal csak egyik irányát használja, adatot nem tudunk küldeni az eszköznek. A készülék által küldött adatok feldolgozásáért egy LIDAR nevezetű osztály felel.



22. ábra: LIDAR osztálydiagramja

4.5.1 YDLIDAR könyvtár

A YDLIDAR gyártó biztosít egy előre elkészített példakódot, amely segítségével az X2-es szenzort is tudjuk használni. Ez a példakód többek között tartalmaz soros port kezelést, a szenzortól küldött üzenetcsomagok feldolgozását támogató osztályokat, függvényeket, és egy ROS csomópontot is. A példakódban sok olyan rész található, ami a robot szempontjából felesleges, ilyen például az előbb felsoroltak közül a soros port kezelése és a ROS csomópont is, hiszen ezekből már van saját leimplementált megoldásunk. Azonban vannak olyan részek is, amelyekre kifejezetten szükségünk lenne, mint például a mérési adatok feldolgozását szolgáló részek. Emiatt készítettem el a robot számára egy YDLIDAR nevű könyvtárba egy olyan drivert, amely bár a YDLIDAR gyártó kódját használja, annak csupán a robot számára fontos részeit tartalmazza.

4.5.2 Mérési adat feldolgozása

A LIDAR szenzor mindössze egyetlen bemenettel csatlakozik az ST mikrokontrollerhez, ez egy UART csatorna egyik szála. Az érzékelő folyamatosan küldi az általa készített mérések eredményeit. Mivel a hardver interfész ilyen egyszerű, ezért az eszköz nem igényel semmilyen induláskori inicializálást. A LIDAR jelek feldolgozását a periodikusan meghívott Process tagfüggvény végzi.

4.5.2.1 X2 csomagok

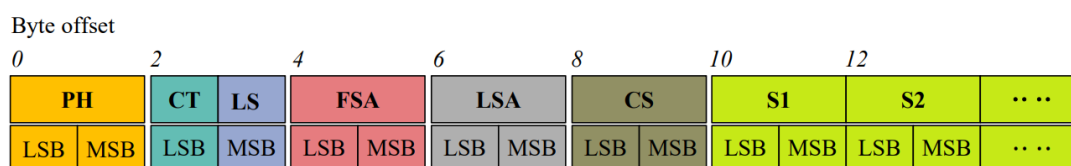


FIG 3 SCAN COMMAND RESPONSE CONTENT DATA STRUCTURE

CHART 2 SCAN COMMAND DESCRIPTIONS

| Item | Name | Description |
|---------|-------------------|--|
| PH(2B) | Packet header | The length is 2B, fixed at 0x55AA, with the low position in the front and the high position in the back. |
| CT(1B) | Packet type | Indicates the type of the current packet; 0x00: Point cloud packet 0x01: Start packet |
| LSN(1B) | Number of samples | Indicates the number of sampling points contained in the current packet; there is only one starting point in the starting packet, and the value is 1. |
| FSA(2B) | Starting angle | Angle data corresponding to the first sample point in the sampled data |
| LSA(2B) | End angle | Angle data corresponding to the last sample point in the sampled data |
| CS(2B) | Check code | The current data packet check code, using double-byte XOR to verify the current data packet |
| Si(2B) | Sampled data | The sampling data of the system test is the distance data of the sampling point, and the interference flag is also integrated in the LSB of the Si node. |

23. ábra: X2 LIDAR szenzor által küldött csomagok

Az X2 kisebb csomagok formájában küldi az eredményeket, ezek felépítését a 23. ábra is szemlélteti. Minden csomag rendelkezik egy fejléccel – ez a csomag első 10 bájtja – illetve adat elemekkel, ezeket a dokumentáció "sample"-nek, vagyis mintának nevezi.

4.5.2.2 16 bites adatok kezelése

Az ábrán is látható, hogy a csomagok alapvetően 16 bites adatokból állnak, szerencsére mind az X2 szenzor, mind az ST processzor little endian bájtrendet használ, ezért az adatok 16 bites értelmezése nagyon egyszerű, a bájtok megegyező sorrendben helyezkednek el a két eszköz esetében. A feldolgozást egyszerűsíthetjük tehát, ha eleve olyan tárolót alkalmazunk, amely 16 bites adatokat kezel. A lidar_inData buffer pont ilyen. Az UART buffere és az lidar_inData közötti másoló függvény biztosítja, hogy az adatok konzisztensek maradjanak, így a lidar_inData-ban levő elemeket a LIDAR osztály függvényei már közvetlenül 16 bites értékeként tudják használni.

4.5.2.3 Start csomagok

Az X2 által küldött csomagok fejlécének második eleme (értelmezhető második 16 bites adat) két részből áll: az első bájt a csomag típusát tartalmazza, a második bájt pedig az minták számát, vagyis az üzenet hosszát.

A dokumentáció alapján kétféle csomag létezik: az alapértelmezett adatcsomag, ez mérési eredményeket tartalmaz, és az úgynevezett "Start packet", tehát kezdő csomag. A kezdőcsomag abból a szempontból különleges, hogy egy teljes – a forgó szenzor teljes tartományát bejáró – mérés kezdetét jelzi. Vagyis két kezdő csomag között küldött összes mérési csomagban kapott adatot nézve kapunk egy teljes, 360°-os mérést. A gyártó által biztosított kód is úgy dolgozta fel az adatokat, hogy mindig egy kezdőcsomagtól kezdte el az adatok lementését, egészen addig, amíg egy újabb kezdőcsomagot nem kapott – ezt már nem értette bele az előző mérésbe.

4.5.2.4 Csomagok feldolgozása

A robot szoftverében a LIDAR osztály két tagfüggvénye felelős az X2 által küldött csomagok feldolgozásáért, ezeket természetesen a Process függvény hívja meg.

A Process először az Rx_ProcessFindStart függvényt fogja futtatni. Ennek a metódusnak az a szerepe, hogy találjon egy kezdő csomagot a beérkezett adatok között. A Rx_ProcessFindStart, társaival ellentétben, nem 16 bites adatokkal dolgozik, hanem közvetlenül az UART bufferét használva bájtokat vizsgál. Erre azért van szükség, mert az UART tárolójában levő elemekről nem tudjuk, hogy melyiket kéne alsó vagy felső bájtjának értelmezni. Ha elfogytak az tárolóból az elemek, és nem találtunk kezdő csomagot, akkor feltételezzük, hogy még nem érkezett ilyen és hamis értékkel visszatér a Rx_ProcessFindStart függvény, ez a taszk rövid ideig tartó altatását fogja eredményezni.

Ha sikeresen találtunk kezdőcsomagot az Rx_ProcessFindStart függvény segítségével, akkor folytatódhat a feldolgozás, vagyis a Rx_ProcessMeasurement meghívása következik. Ez a függvény addig fogad csomagokat, amíg egy újabb kezdő csomag nem érkezik, vagy ha valamilyen hiba miatt félbe kell szakítania a feldolgozást. A félbeszakításnak két oka lehet:

- A gyártó úgy implementálta a csomagkezelést, hogy folyamatos adatkonzisztencia ellenőrzés folyik a beérkezett adatokra, bit ellenőrző

összeg előállításával. Ez a módszer a Rx_ProcessMeasurement függvénybe is bekerült, ha az ellenőrzés elbukik, akkor a visszatérünk hamis értékkel.

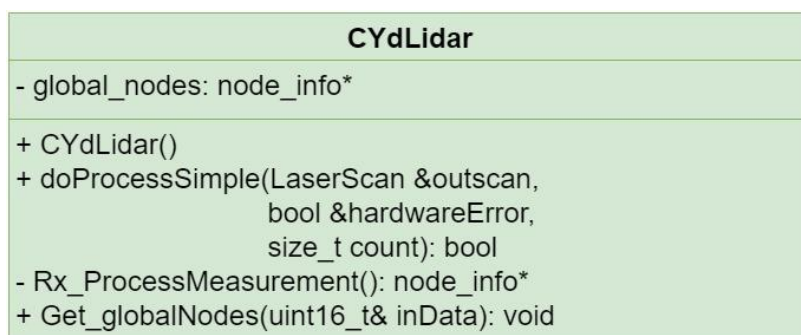
- A LIDAR eredmények kezelése memória igényes, két kezdőcsomag között nagyjából ezer minta érkezik, ezeket és az ebből számított eredményeket valahol ideiglenesen tárolni kell, ha az erre a célra létrehozott buffer túl kicsi, a függvény kénytelen hamis értékkel visszatérni.

Az Rx_ProcessMeasurement függvény hamis értékkel való visszatérése a mérési eredmények eldobását és az UART tároló kiürítését jelenti, és a feldolgozás előlről kezdjük a Rx_ProcessFindStart függvénytől.

Az Rx_ProcessMeasurement sikeres lefutása esetén a laser tagváltozó bufferjében találhatóak az eredmények.

4.5.2.5 CYdLidar osztály

Ez az osztály az X2 gyártója által készített könyvtárból származik. A robot kódjában csak a robot számára fontos részek kerültek bele az eredeti osztályból, azonban a feldolgozást végző algoritmusba nem került változás.



24. ábra: CYdLidar részleges osztálydiagramja

A LIDAR osztály rendelkezik egy laser nevű tagváltozóval, aminek típusa ez a CYdLidar osztály. A csomagokból kinyert mérések a laser változó global_nodes tömbjébe kerülnek, minden mintához egy-egy elem. Láthatjuk, hogy ha ezer körüli minta érkezik, akkor ennek a tömbnek a mérete egyáltalán nem elhanyagolható. A global_nodes node_info típusú elemeket tárol.


```

struct node_info {
    uint16_t    sync_quality;
    uint16_t    angle_q6_checkbit;
    uint16_t    distance_q2;
    uint64_t    stamp;
    uint8_t     scan_frequency;
    uint8_t     index;
} __attribute__((packed)) ;

```

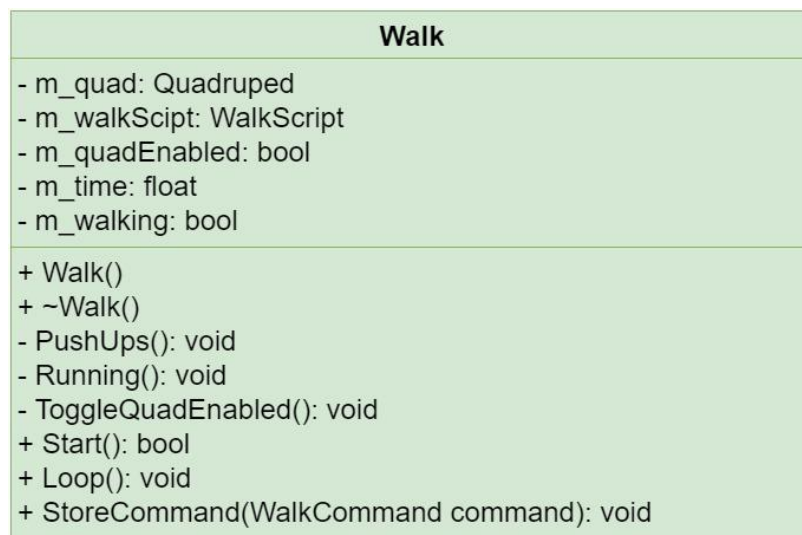
6. kódrészlet: node_info struktúra

CYdLidar osztály doProcessSimple tagfüggvénye a megkapott node_info változók alapján visszaad egy eredményt, amit már át tudunk alakítani a ROS LaserScan típusára. A konverzió után pedig továbbítani tudjuk az immár feldolgozott mérési eredményeket.

A LaserScan típus már olyan formában tárolja a mérési eredményeket, amit a ROS különböző eszközei már értelmezni tudnak, például az RVIZ képes megjeleníteni a mérési pontokat.

4.6 Mozgás vezérlése

A robot négy lábbal rendelkezik, amelyek mindegyike három szervomotorból áll. A mozgás irányítására létezik egy Walk nevű osztály, amely magas szintű interfészt biztosít a számunkra.



25. ábra: Walk osztálydiagramja

A Walk osztály és az általa által használt osztályok nagyrészen építenek Babics Mátyás munkájára. A korábbi megoldás szerint a létezett egy Program osztály, aminek leszármazottja volt a WalkProgram osztály – ezek összevonásával született meg a Walk.

4.6.1 Mozgás elemek

A Walk osztály egy tagváltozó formájában rendelkezik egy WalkScript osztálypéldánnyal. Ez az osztály használható mozgáselemek végrehajtására. A WalkScript egy bufferben tárolja a végrehajtandó mozgásokat – ez az m_script változó – amelyeket majd az Update függvény hívásával lehet végrehajtani. Ezt az Update függvényt a Walk osztály Loop függvénye hívja meg, ha a mozgás engedélyezve van.

A WalkScript osztály tárolójába két magas szintű függvénnyel tehetünk be elemet. Ezek az AddPathElementWalkStraight, és az AddPathElementTurn függvények. Az első egy távolságot, a második egy szög értéket vár argumentumként. A lábak megfelelő vezérléséről már a WalkScript osztály gondoskodik.

4.6.2 Mozgás kérése

Bár a Walk osztály nagyrészt már korábban is létező kódra épít, egy nagyon jelentős újítás is bekerült az osztály interfészébe: képesek vagyunk mozgást kérni a robottól ROS interfészen keresztül. Ezt a StoreCommand függvény segítségével tehetjük meg, amelynek egy WalkCommand típusú kérést tudunk beadni.

```
typedef struct __attribute__((__packed__)) WalkCommand{
    uint8_t direction;
    float parameter;
} WalkCommand;
```

7. kódrészlet: WalkCommand struktúra

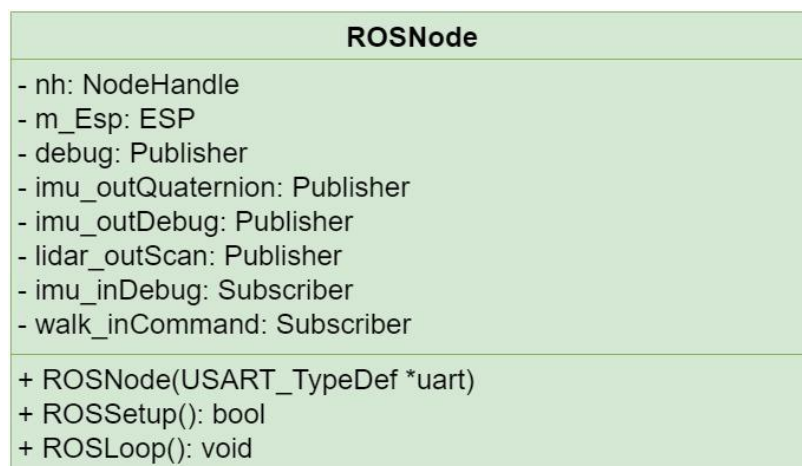
A WalkCommand struktúra két elemből áll – egyik a kérés irányára vonatkozik, a másik, pedig egy paraméter, amit az irány függvényében lehet értelmezni. Jelenleg négy féle kérést tud feldolgozni a függvény, érvénytelen kérés esetében pedig nem reagál semmit. Az érvényes kérések:

- Mozgás előre, ebben az esetben a paraméter egy távolság. A függvény az AddPathElementWalkStraight függvény segítségével helyez el egy elemet a feldolgozandó mozgások sorába.

- Forgás, ilyenkor a paraméter egy radiánban értendő szög. A függvény az AddPathElementTurn függvény segítségével helyez el egy elemet a feldolgozandó mozgások sorába.
- Szervomotorok engedélyezése, ilyenkor a paraméter 0 értéke esetében letiltjuk más érték esetében pedig engedélyezük a szervomotorok vezérlését.
- Séta engedélyezése, a paraméter 0 értékénél hamis, egyéb esetben igaz értékre állítjuk az m_walking tagváltozót, ez a mozgás végrehajtását tudja befolyásolni. (Jelen esetben ez a változó is egy engedélyezésnek számít.)

4.7 Beágyazott ROS csomópont

A roboton futó kód rendelkezik egy ROS csomóponttal, ez teszi lehetővé, hogy az eszköz a ROS által definiált interfészeket tudja használni. A ROS csomópontot egy csomagoló osztály kezeli, ez a ROSNode.



26. ábra: ROSNode részleges osztálydiagramja

4.7.1 A ROSNode fontosabb elemei

A többi taszkhoz hasonlóan a ROSNode is rendelkezik egy induláskor lefuttatandó tagmetódussal, ez a ROSSetup, és egy periodikusan meghívható tagfüggvénnyel, ez pedig a ROSLoop.

A NodeHandle típusú tagváltozó maga a ROS node, ezen keresztül tudunk topic-okat definiálni, adatot publikálni, vagy minden egyéb ROS-al kapcsolatos feladatokat elvégeztetni. A ROS node futtatását a spinOnce függvényének meghívásával tehetjük meg, ezt a ROSNode osztály ROSLoop függvénye teszi meg.

A ROSNode osztályban található Publisher illetve Subscriber típusú tagváltozók topic-okat definiálnak. A Publisher változók segítségével adatot tudunk küldeni, ezt a publish függvény végzi. A szoftverben található Publisher-ek:

- debug – Általános rendszer információkat ad, például az IMU szenzor kalibrációs státuszát.
- imu_outQuaternion – Az IMU szenzor mérési eredményeit küldi.
- imu_outDebug – Az IMU szenzornak megadott debug kérés eredményét biztosítja.
- lidar_outScan – A LIDAR szenzor mérési eredményeit küldi.

A Subscriber változók adatfogadásra alkalmasak. Az üzeneteket megfelelő formában várják, ez megegyezik a topic típusával. Ha üzenet érkezik, akkor azt egy callback függvény fogja feldolgozni. A szoftverben található Subscriber-ek:

- imu_inDebug – Az IMU szenzornak küldhetünk debug kérést.
- walk_inCommand – Mozgást kérhetünk vagy engedélyezhetünk.

A ROSNode tagváltozóként rendelkezik egy ESP típusú változóval is, ez a wifi kommunikációért felel. Az ESP osztály működéséről részletesebb információ a 4.3 fejezetben található.

4.7.2 ROS node inicializálása

A rendszer indulásakor az első függvény, ami futni fog az osztály tagfüggvényei közül a ROSSetup.

```

void ROSNode::ROSSetup()
{
    m_Esp.Start();
    nh.initNode();

    /* Advertise publishers */
    nh.advertise(debug);
    nh.advertise(imu_outQuaternion);
    nh.advertise(imu_outDebug);
    nh.advertise(lidar_outScan);

    /* Subscribe */
    nh.subscribe(imu_inDebug);
    nh.subscribe(walk_inCommand);

    m_Esp.Process();
}

```

8. kódrészlet: ROSSetup függvény

A ROSSetup felelős az ESP Start függvényének meghívásáért, illetve a topic-ok létrehozásáért.

4.7.3 A ROS node futása

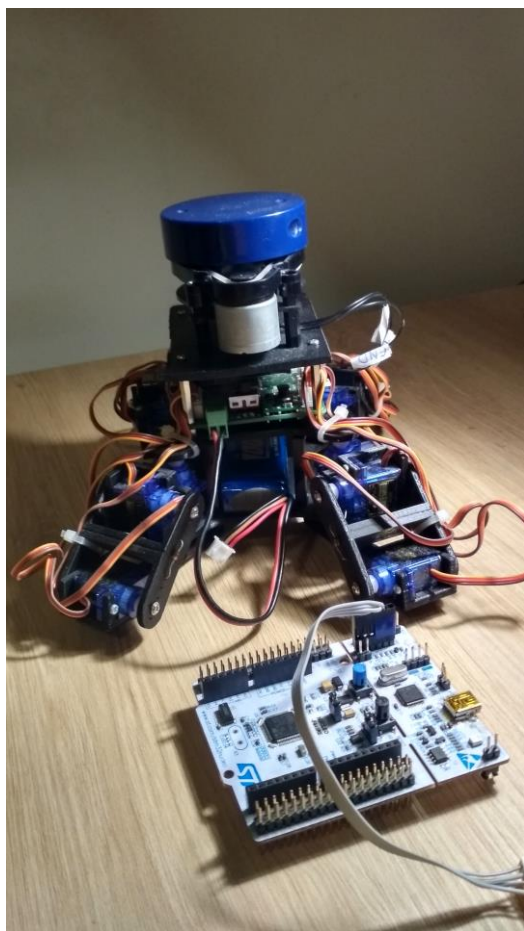
A ROS csomópont periodikus futtatását a ROSLoop függvény végzi. Ez a függvény felel az ESP osztály Process függvényének illetve a NodeHandle spinOnce függvényének meghívásáért is. A topic-okra való írás is ebben a metódusban történik meg. A csomópont az Event Group által beállított bitek alapján tudja, hogy melyik az a topic, amelyekre adatot kell küldeni.

5 Tesztek, eredmények

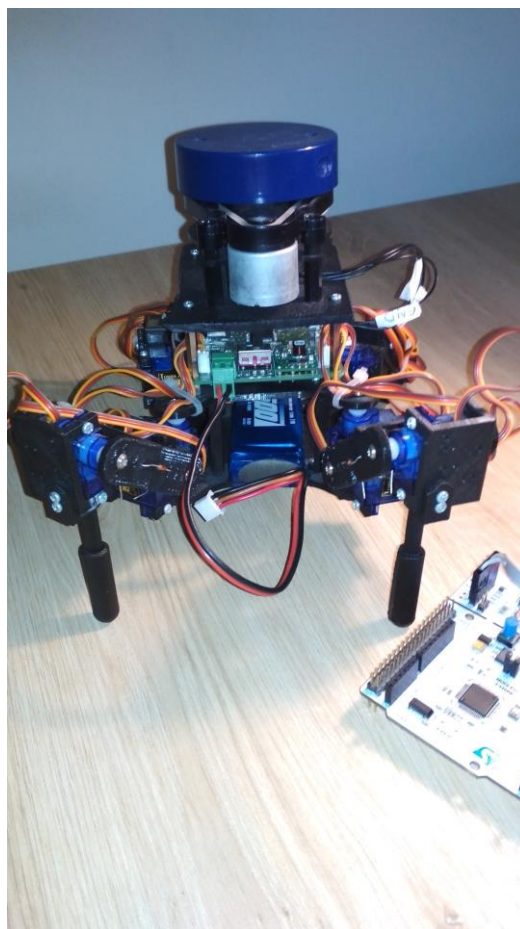
Az előző fejezetek a rendszer egyes elemeire fókuszáltak, azok felépítését, funkcionalitását mutatták be. Ez a fejezet a rendszer egészét tekinti át, ismerteti, hogy milyen lépésekben fejlődött a robot, és szemlélteti a rendszer működését.

5.1 Forrasztás, élesztés

A 2.2 fejezetben taglalt nyomtatott áramkör elkészítését az Eurocircuits PCB gyártón keresztül, az AUT tanszék segítségével oldottuk meg. Az elemek felforrasztását a tanszék erre kialakított laborjában végeztem. Az elkészült áramkör élesztésekor két fontos lépést végeztem el, az első, hogy előállnak-e a megfelelő feszültségszintek. A másik pedig, hogy a szervomotorok számára megfelelően generálódnak-e a PWM vezérlőjelek. Mindkét mérés sikerrel zárult, így az új áramkört be tudtam szerelni a robotba.



27. ábra: Az összeszerelt robot



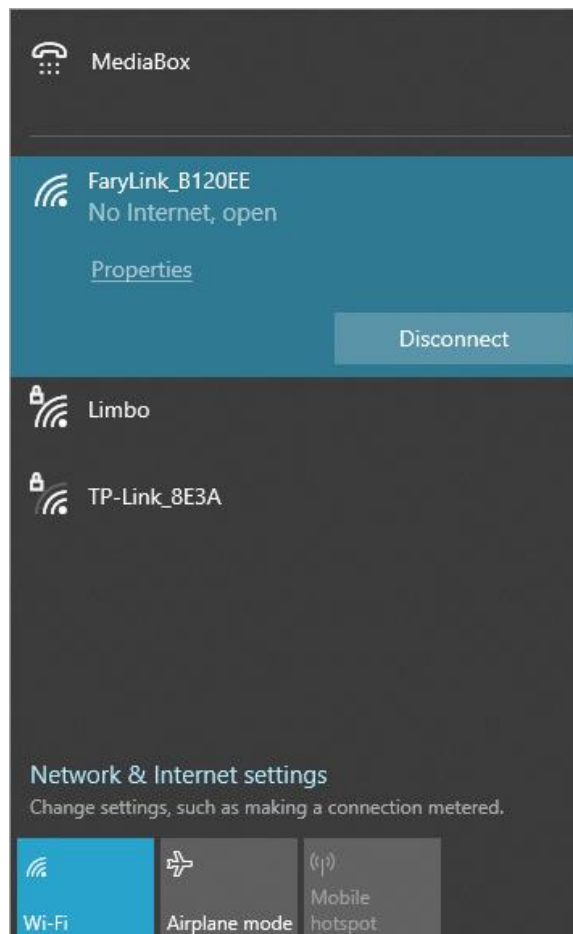
28. ábra: Az összeszerelt robot

5.2 ROS eredmények

A diplomamunkám célja volt, hogy a robot ROS interfésszel kompatibilis legyen. Bár számos egyéb funkcióval is sikerült kibővíteni a robotot, a legszemléletesebb mégis az itt elért eredményt, hiszen a szenzorok mérési eredményei csak akkor érkeznek meg helyesen, ha a szenzorvezérlő kódok és a frissen integrált operációs rendszer is helyesen működik. Ha probléma van a robot bármelyik részével, az ezen a szinten biztosan jelentkezni fog.

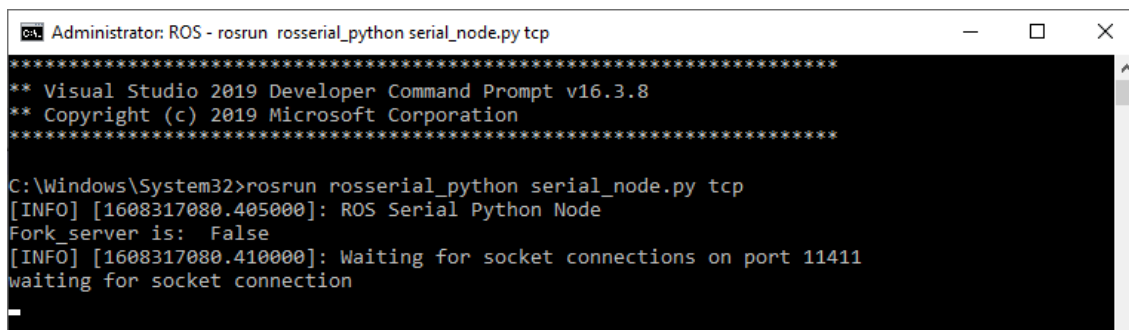
5.2.1 ROS kapcsolat

A robot induláskori első lépése, hogy megpróbálja kiépíteni a kapcsolatot a roboton futó ROS csomópont és az asztali számítógépen futó ROSSerial node között. Ehhez elsőként létrehoz egy hálózati kapcsolatot, amire a PC-s eszköz csatlakozni tud.



29. ábra: A PC csatlakozott az ESP-01 hálózataára

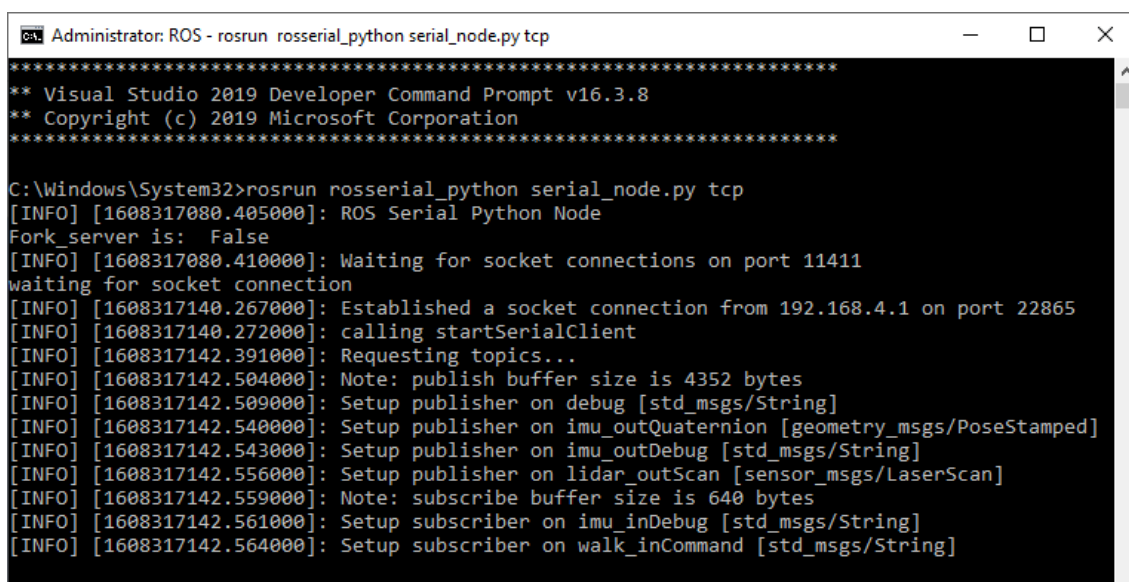
Amíg a robot végzi az indulási folyamatát mi is elindíthatjuk a PC-n a ROS csomópontot, ezt a következő paranccsal tehetjük meg: `roslaunch rosserial_python serial_node.py tcp`



```
Administrator: ROS - roslaunch rosserial_python serial_node.py tcp
*****
** Visual Studio 2019 Developer Command Prompt v16.3.8
** Copyright (c) 2019 Microsoft Corporation
*****

C:\Windows\System32>roslaunch rosserial_python serial_node.py tcp
[INFO] [1608317080.405000]: ROS Serial Python Node
Fork_server is: False
[INFO] [1608317080.410000]: Waiting for socket connections on port 11411
waiting for socket connection
_
```

30. ábra: ROSSerial node várakozik a hálózati kapcsolatra



```
Administrator: ROS - roslaunch rosserial_python serial_node.py tcp
*****
** Visual Studio 2019 Developer Command Prompt v16.3.8
** Copyright (c) 2019 Microsoft Corporation
*****

C:\Windows\System32>roslaunch rosserial_python serial_node.py tcp
[INFO] [1608317080.405000]: ROS Serial Python Node
Fork_server is: False
[INFO] [1608317080.410000]: Waiting for socket connections on port 11411
waiting for socket connection
[INFO] [1608317140.267000]: Established a socket connection from 192.168.4.1 on port 22865
[INFO] [1608317140.272000]: calling startSerialClient
[INFO] [1608317142.391000]: Requesting topics...
[INFO] [1608317142.504000]: Note: publish buffer size is 4352 bytes
[INFO] [1608317142.509000]: Setup publisher on debug [std_msgs/String]
[INFO] [1608317142.540000]: Setup publisher on imu_outQuaternion [geometry_msgs/PoseStamped]
[INFO] [1608317142.543000]: Setup publisher on imu_outDebug [std_msgs/String]
[INFO] [1608317142.556000]: Setup publisher on lidar_outScan [sensor_msgs/LaserScan]
[INFO] [1608317142.559000]: Note: subscribe buffer size is 640 bytes
[INFO] [1608317142.561000]: Setup subscriber on imu_inDebug [std_msgs/String]
[INFO] [1608317142.564000]: Setup subscriber on walk_inCommand [std_msgs/String]
```

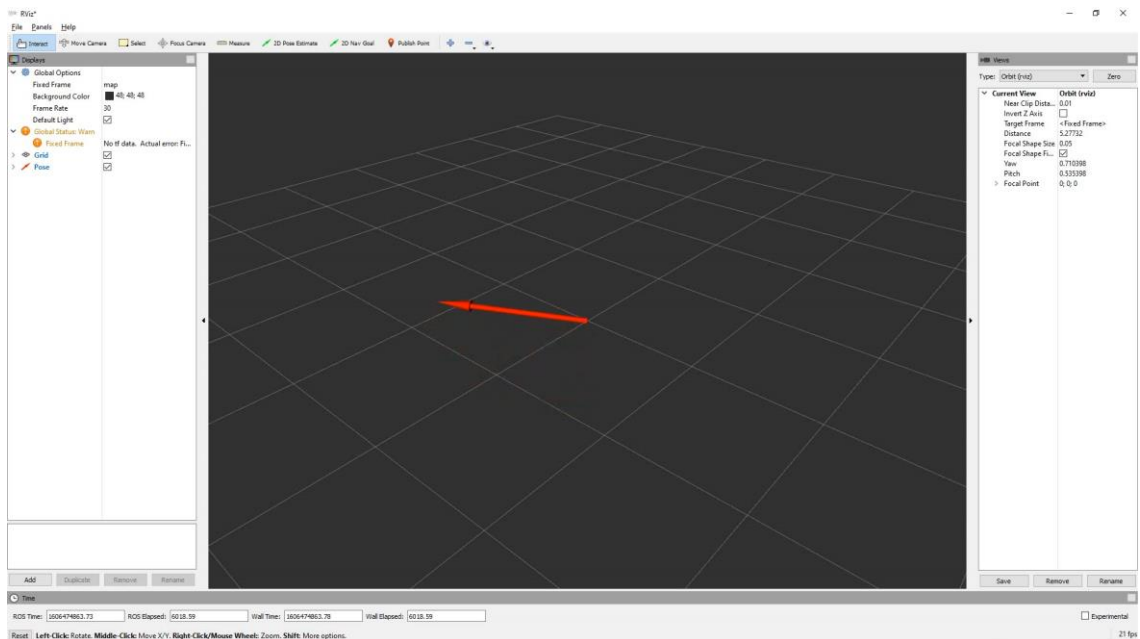
31. ábra: ROSSerial node fut

A node futtatása után láthatjuk, ahogy a kapcsolat felépül: A 30. ábrán látható, hogy a csomópont indulás után szeretne kapcsolatot kiépíteni, ha a PC rácsatlakozik az ESP-01 hálózatra (29. ábra), akkor ezt meg is tudja tenni. A sikeres kapcsolatfelvétel után a node kiírja a konzolba az elérhető topic-okat, ezt a 31. ábrán láthatjuk.

5.2.2 RVIZ

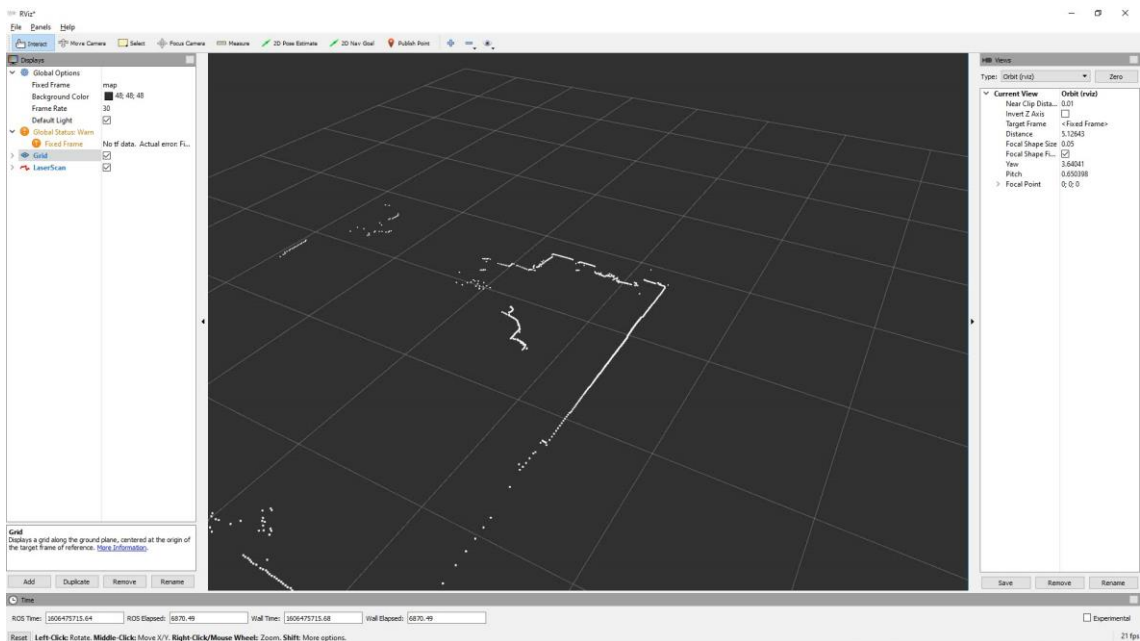
Ha sikeresen el tudtuk indítani a ROS-t mind a PC-n, mind a roboton, akkor a beérkezett adatokat ellenőrizni tudjuk, erre kiváló eszköz például az RVIZ. Az RVIZ 3D-s vizualizációval képes mérési eredményeket szemléletes módon ábrázolni, ráadásul

a megjelenített jelek közé a megfelelő topic segítségével a robot üzeneteit is fel tudjuk venni.



32. ábra: Az IMU mérési eredményének ábrázolása

A 32. ábra szemlélteti, hogy az IMU szenzor által küldött jelet hogyan tudjuk ábrázolni. A piros színű nyíl a robot térbeli elhelyezkedésétől függően változik.



33. ábra: A LIDAR mérési eredményeinek ábrázolása

A 33. ábrán látható a LIDAR szenzor által küldött mérés, a fehér vonalak jelentik, hogy a valamilyen tárgy helyezkedik el abban a távolságban, a kép jobb oldalán látható összefüggő vonal, például egyértelműen egy falat jelez.

5.2.3 Robot mozgatása

A robot mozgatását a `walk_inCommand` topic-ra küldött parancsokkal tehetjük meg. Ez a topic szöveges formában várja az üzeneteket és majd a ROS node fogja átalakítani a 4.6.2 fejezetben leírt adattípusba.

A robot mozgatásához először engedélyeznünk kell azt, ehhez két parancsot kell elküldeni: `"enable 1"` és `"walking 1"`. Az engedélyezés után a robot végre tudja hajtani a kért mozgáselemeket. Jelenleg három féle mozgást kérhetünk:

- `"forward <távolság>"` – A robot előre fog mozogni.
- `"right <szög>"` – A robot jobbra fog forogni a fokban megadott szögnek megfelelően.
- `"left <szög>"` – A robot balra fog forogni a fokban megadott szögnek megfelelően.

5.2.4 Térképezés

A térképkészítéshez a `hector_slam` ROS könyvtárat használtam. A robot paramétereit egy launch fájl segítségével tudtam megadni, ez a 9. kódrészletben látható.

```

<launch>

<node pkg="tf" type="static_transform_publisher" name="map_to_odom"
args="0.0 0.0 0.0 0.0 0.0 0.0 /map /nav 40"/>

<node pkg="tf" type="static_transform_publisher"
name="odom_to_base_link" args="0.0 0.0 0.0 0.0 0.0 0.0 /nav
/base_footprint 40"/>

<node pkg="tf" type="static_transform_publisher"
name="base_link_to_laser" args="0.2245 0.0 0.2 0.0 0.0 0.0
/base_footprint /base_laser 40" />

<include file="$(find hector_mapping)/launch/mapping_default.launch" >
<arg name="scan_topic" value="lidar_outScan"/>
</include>

<node pkg="rviz" name="rviz" type="rviz" />
<include file="$(find hector_geotiff)/launch/geotiff_mapper.launch" />

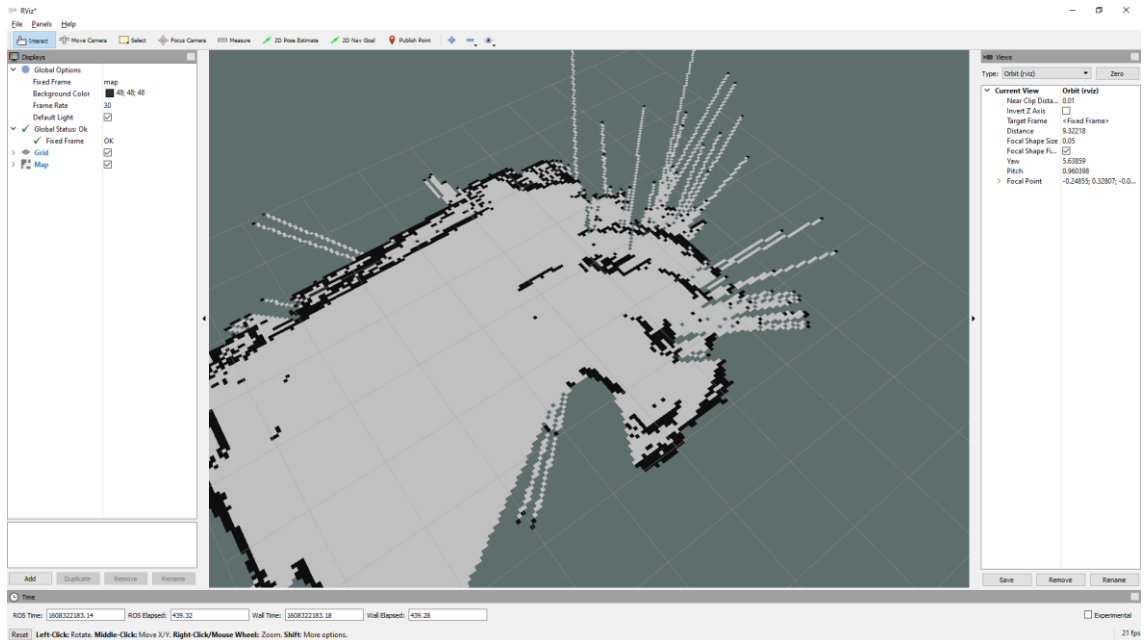
</launch>

```

9. kódrészlet: hector_slam launch fájl

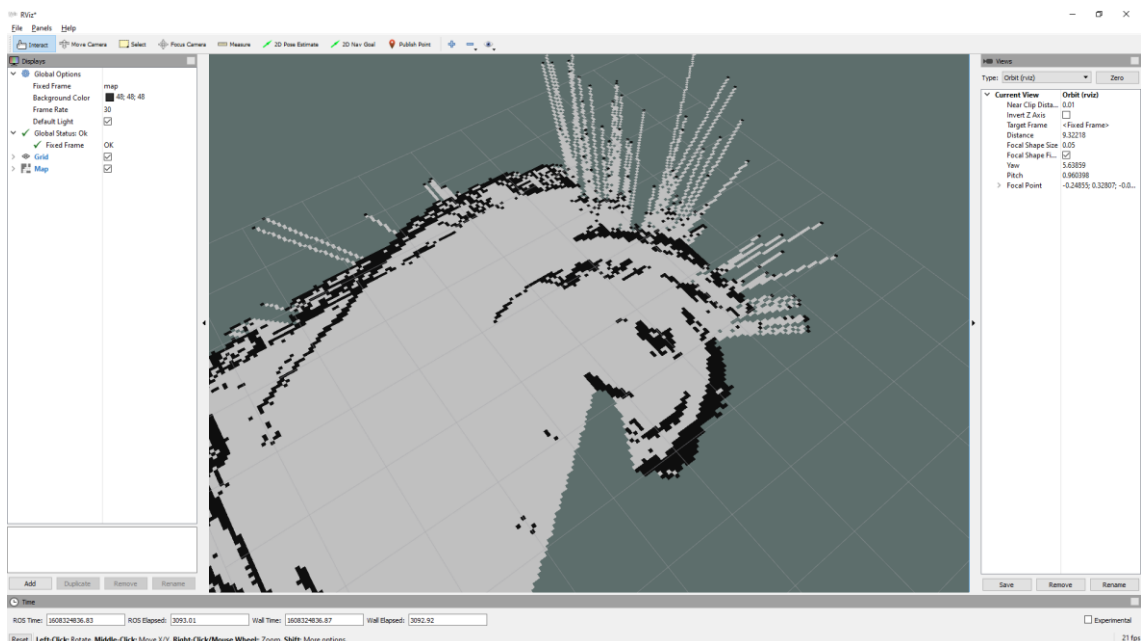
A launch fájl biztosítja, hogy a megfelelő transzformációs elemek rendelkezésre álljanak, ez a gyakorlatban azt jelenti, hogy a robot egyes koordináta rendszerei hogyan helyezkednek el egymáshoz képest. Az itt használt launch fájl egy nagyon egyszerű, statikus viszonyt határoz meg a base_laser, vagyis a LIDAR szenzor, és a base_link, vagyis a robot középpontja között. Ezen kívül ebben a fájlban tudjuk megadni, hogy milyen topic-on érkezik majd a LIDAR mérési eredménye – jelen esetben ez a lidar_outScan topic.

A térképezés folyamatát RVIZ-ben tudjuk nyomon követni, a robot mozgásával a térkép szépen felépül.



34. ábra: Térképkészítés RVIZ-ben

A 34. ábrán látható a SLAM algoritmus eredménye, a kép bal felső részén egy viszonylag egybefüggő, egyenes objektum látható, ez a valóságban egy falnak felel meg. A kép alsó részén látható, hogy a térkép egy részét még nem jártuk be, így ott hiányos az információ. Sajnos a 34. ábrán látható felvétel után az algoritmus eltévedt, és helytelen eredményt kezdett produkálni, ez látható a 35. ábrán. A hiba a leglátványosabb a bal felső részen, ami eddig egy egyenes fal volt, az elcsúszott.



35. ábra: Térkép készítés rossz eredménnyel

Összeségében a SLAM algoritmus helyesen kezd el működni, vagyis megkapja a robot által küldött méréseket, azonban a működésnek még vannak hiányosságai. Ez a fajta elcsúszás nagy valószínűséggel ahhoz köthető, hogy a LIDAR jelek nem elég gyakran és nem elég pontos időbélyeggel érkeznek. Megfigyeltem azt is, hogy a hiba a robot forgatásakor jön elő, transzlációs mozgatsnál a térkép ilyen fajta elcsúszása nem fordult elő.

A térképező eljárást tehát két módon is lehet javítani: a LIDAR jelek sűrűbb küldésével, és az IMU jeleinek beintegrálásával, ami a robot forgásánál tudna segíteni az algoritmusnak.

6 Összegzés, értékelés

Összességében a robotot sikerült számos új funkcióval kiegészítenem. A hardvert kibővítettem két új szenzorral, ezek segítségével a robot képes információt gyűjteni a saját, illetve környezete állapotáról, így a robot lokalizációjának alapvető elemei lehetnek.

A hardveres újítások azonban csak egy kis része volt a robot fejlesztésének, a legtöbb módosítás a robot szoftverét érintette. A robot szoftverébe beintegráltam egy FreeRTOS operációs rendszert, amely elegáns és kényelmes taszkkezelést és ütemezést biztosít számunkra, sok egyéb szoftveres eszközzel együtt, mint például tárolók, esemény csoportok.

Az újonnan hozzáadott szenzorokhoz természetesen vezérlő kódra is szükség volt. Ezt taszkok formájában tudtam beilleszteni a szoftverbe, a wifi kommunikációért felelős kóddal és a ROS csomóponttal együtt. Az így létrehozott taszkok segítségével létre tudtam hozni egy wifin keresztül működő ROS interfészt. A szenzorok által mért adatokat sikeresen biztosítani tudtam egy PC-n futó ROS node számára.

A végső állapotában a robot képes volt szenzor adatokat küldeni, illetve mozgási parancsokat fogadni ROS interfészen keresztül, és sikerült térképet is készítenie. Viszont a térképkészítésnél problémát okozott, hogy a LIDAR szenzor jelei nem érkeznek meg elég gyakran. Ehhez a problémához kötődik az a jelenség is, hogy ha mind a két szenzort egyszerre használjuk, akkor érezhetően romlik a frissítési idő. Ebből is látható, hogy a kommunikáció bár működik, némi optimalizációt még igényel.

6.1 Továbbfejlesztési lehetőségek

Bár a munkám a pókszerű robottal a végéhez ért, ez korán sem jelenti azt, hogy ne lenne több opció, amely mentém a robotot fejleszteni lehetne. Az új szenzorok és a ROS rendszer számtalan új lehetőséget tárnak elénk. A térképezési algoritmus kicsiszolásával lehetőséget kapunk mindenféle bejárési algoritmus kipróbálására. Vagy elmozdulhatunk a másik irányba is és fókuszálhatunk a robot modellezésére, a ROS segítségével tetszőleges pontosságú szimulált modellt alkothatunk a robotról, a jelenlegi mindössze két elemből álló modell helyett.

A robot csuklóit vezérlő kódok fejlesztésére is van lehetőség. Jelenleg a robot előre haladó és forgó mozgásra képes, de a megfelelő interfészek kivezetésével a lábakat, akár csuklónként is tudjuk vezérleni, így lehetőségünk adódna a robotot tetszőleges pozícióba vezérelni.

Akár a robot hardverében is ejthetünk módosításokat, bár szenzorok szempontjából a robot elég felszereltnek tekinthető, legalább is a jelenlegi használat mellett. Ami viszont igényelne fejlesztést az a robot váza, a mostani megoldásban a robot lábai 3D nyomtatott elemekből állnak, ezek már az eredeti tervben is kicsit bizonytalannak tűntek, amin a viszonylag nehéznek tekinthető LIDAR hozzáadása nem sokat segített. Így a robot mechanikai szempontból történő módosítása is egy ígéretes projekt lehet.

Irodalomjegyzék

Forráskód

- [1] Project repository:
https://LorantMassar@bitbucket.org/LorantMassar/pokszeru_robot_ros.git

Fejlesztő eszközök

- [2] Altium Designer: <https://www.altium.com/>
- [3] System Workbench for STM32:
<https://www.openstm32.org/System+Workbench+for+STM32>

Adatlapok, kézikönyvek

- [4] STM32L476RET6 adatlap: <https://www.st.com/en/microcontrollers-microprocessors/stm32l476re.html>
- [5] YDLIDAR X2 dokumentumok:
https://www.ydlidar.com/service_support/download.html?gid=6

- [6] BNO055 dokumentumok: <https://www.bosch-sensortec.com/products/smart-sensors/bno055.html>
- [7] ESP-01 AT Instruction Set:
https://www.espressif.com/sites/default/files/documentation/4a-esp8266_at_instruction_set_en.pdf

ROS

- [8] ROS: <https://www.ros.org/>
- [9] ROSSerial: <http://wiki.ros.org/roscpp>
- [10] Rviz: <http://wiki.ros.org/rviz>
- [11] hector_slam: http://wiki.ros.org/hector_slam

FreeRTOS

- [12] FreeRTOS: <https://www.freertos.org/>

Egyéb források

- [13] SLAM wikipédia oldal: https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping
- [14] LIDAR wikipédia oldal: <https://en.wikipedia.org/wiki/Lidar>
- [15] IMU wikipédia oldal: https://en.wikipedia.org/wiki/Inertial_measurement_unit
- [16] Kvaternió wikipédia oldal: <https://en.wikipedia.org/wiki/Quaternion>
- [17] hector_slam RoboCup 2011 youtube videó:
https://www.youtube.com/watch?v=F8pdObV_df4&ab_channel=TeamHectorDarmstadt