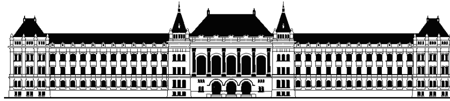


Budapesti Mûszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék

Autonóm robotrepülőgépek állapotbecslő és szabályzó módszerei

TDK dolgozat

Készítette
Zsoldos Balázs

Konzulens
Belső Zoltán, Rucz Péter

2015. október 26.

Tartalomjegyzék

Kivonat **3**

Bevezető **4**

1. Az UAV-ok fedélzeti elektronikus rendszere **6**
1.1. Szenzorok . . . 6
1.1.1. Koordináta-rendszerek . . . 6
1.1.2. Giroszkóp . . . 7
1.1.3. Gyorsulásmérő . . . 8
1.1.4. Mágneses szenzor . . . 8
1.1.5. GPS . . . 9
1.1.6. Ultrahangos távolságmérő . . . 9
1.1.7. Barometrikus nyomásmérő . . . 9
1.1.8. Légsebességmérő . . . 10
1.2. Robotpilóta . . . 10
1.2.1. Állapotbecslés . . . 10
1.2.2. Guidance . . . 11
1.2.3. Szabályzás . . . 11
1.3. Aktuátorok . . . 11
1.4. Payload . . . 12
1.5. Kommunikáció . . . 12

2. Amores hardver rendszer **13**
2.1. Amores ProHD . . . 13
2.2. Amores Minibot . . . 17

3. Szenzorfúzió **19**
3.1. Orientáció leírása . . . 19
3.1.1. Forgatási mátrix . . . 19
3.1.2. Euler szögek . . . 20
3.1.3. Kvaterniók . . . 20
3.1.4. Rodrigez paraméterek . . . 21
3.2. Orientáció mérése . . . 21
3.2.1. Szögsebességek mérése . . . 21

3.2.2. Vektormegfigyelések 22
3.3. Szenzorfúzió 22
3.3.1. Ardupilot 22
3.3.2. Madgwick AHRS 23
3.3.3. Kálmán-szűrő 24
3.3.4. Nemlineáris Kálmán szűrő 26
3.4. Összefoglalás 26

4. Szoftveres keretrendszer **27**
4.1. Multitask alkalmazások Linux alatt 28
4.2. Perifériák kezelése Linux alatt 29
4.2.1. Megszakítás kezelés 31
4.3. A Linuxbot 31
4.3.1. Bemeneti szenzor taskok 32
4.3.2. Elsődleges inerciális szenzor 33
4.3.3. Másodlagos inerciális szenzorok 34
4.3.4. GPS 35
4.3.5. Külső mágneses szenzor 35
4.3.6. Barometrikus nyomásszenzor 35
4.3.7. Külső AHRS rendszer 36
4.3.8. Ultrahangos távolságmérő 36
4.4. Pilot task 36
4.5. Üzenetkezelő task 37
4.6. Hardver in the loop szimuláció 38
4.7. Holder task 38

5. Eredmények **41**

Irodalomjegyzék **47**

Kivonat

Napjainkban egyre több helyen találkozhatunk ember nélküli repülővel, UAV-val (Unmanned Air Vehicle). A technológia fejlődésével, a kisméretű, nagyteljesítményű processzorok elterjedésével a robotrepülőgépek is egyre elérhetőbbé válnak civil felhasználásra is. A mezőgazdaságtól a katasztrófavédelemig az élet számtalan területén nagy előnyt jelent légi felvételek készítése, területfelmérések, műtárgyak, utak, távvezetékek, antennák levegőből történő vizsgálata. Hagyományos repülőgépekkel ezek meglehetősen költséges feladatok, de a kisméretű robotrepülőgépek egyre olcsóbban és könnyebben hozzáférhetővé válnak.

Egy robotrepülő fejlesztése során számtalan mérnöki probléma merül föl. Ami a hagyományos repülőnél egy magasan képzett pilóta feladata, azt mind egy fedélzeten futó programnak kell elvégeznie. A környezet jellemzőit, a jármű mozgását érzékelni kell, ehhez nagy pontosságú, összetett szenzorrendszerre van szükség. Ezen adatok és a kívánt repülési pálya alapján a kellően gyors szabályzó algoritmusoknak el kell végeznie a kormánysíkok állítását, a motorok teljesítményszabályzását. Továbbá rádiós kapcsolatot kell tartani a földi operátorral, kezelni kell a kapott parancsokat, vezérelni kell a kamerákat és egyéb adatgyűjtő rendszereket. Ezek önmagukban is elég komoly problémák, de egy jól működő UAV ezeket egyszerre, nagy megbízhatósággal képes elvégezni, megfelelve a feladat jellegéből fakadó valós idejű követelményeknek.

Jelen dolgozatban egy komplex robotpilóta rendszer fejlesztésének részleteit fogom ismertetni, elsősorban a szenzorok, állapotbecső és szabályzó algoritmusok vonatkozásában. Tárgyalva az architektúrát, az egyes részegységek felépítését, az alkalmazott eszközöket, platformokat és részletesen kitérek a szenzorok alkalmazására és problémáira és a teszteléshez használt környezet és a szimuláció kérdéseire. Rávilágítok a szenzorfúzió fontosságára és összehasonlítok néhány elterjedtebb szenzorfúziós algoritmust. A bemutatott megoldás egy valós igényeket kielégítő, folyamatos fejlesztés alatt álló rendszer.

Bevezető

Dolgozatom egy három éve tartó, mobil robotikai technológiák, azon belül is légi robotok, UAV[1]-ok fejlesztését megcélzó projekt, az úgynevezett AMORES[2] projekt keretében elvégzett munkám eredményeit mutatja be. A projekt egyik célja egy teljesen saját UAV kifejlesztése, hardver, szoftver elemeket és algoritmusokat is beleértve. Egy ilyen nagyszabású projekt természetesen nem egyszemélyes feladat, a teljes rendszer fejlesztését egy több tagú csapat végezte, aminek én is része voltam.

Feladatom, a még jelenleg is tartó fejlesztésben a különböző, a jármű mozgásáról információkat gyűjtő szenzorok vizsgálata, rendszerhez illesztése, illetve a keretrendszer fejlesztése. A szenzorok különböző tulajdonságokkal, hibákkal rendelkeznek és az általuk mért adat kiolvasási módja is szenzoronként változik. Ezeket az adatokat egységesen kell kezelni, az adatokat feldolgozó szenzorfúziós algoritmusnak nem szabad hogy tudomása legyen a perifériakezelés részleteiről. Az architektúrának olyan kiépítésűnek kell lennie, hogy a sokféle szenzort rugalmasan lehessen kezelni, bővíteni, továbbá a robotpilóta algoritmusok függetlenek legyenek a rendszer többi részétől, külön tesztelhetőek, esetleg cserélhetőek legyenek.

Dolgozatomban egy ilyen rendszer kialakítását szeretném bemutatni, az egyes modulok, szenzorok hibáinak és tulajdonságainak figyelembe vételével hogyan építhetünk egy olyan rendszert, ami alkalmas arra, hogy egy bonyolult, szerteágazó funkciójú feladatokat integráljunk egy működő rendszerré. Az implementált szoftveren keresztül bemutatom, hogy a Linux operációs rendszer szolgáltatásait hatékonyan kihasználva hogy valósíthatunk meg egy multifunkcionális, bővíthető UAV keretrendszert.

Az első fejezetben egy UAV rendszer általános felépítését mutatom be. A fontosabb komponensek általános ismertetése után ebben a fejezetben ismertetem részletesen az egyes szenzor típusokat és ezek különböző tulajdonságait, hibáit tárgyalom. A második fejezetben az AMORES projektben megvalósított hardver elemeket mutatom be röviden.

A harmadik fejezetben a szenzorok mért adatainak felhasználásáról, a szenzorfúziós algoritmusok problémájáról lesz szó. Itt kiderül, hogy az egyes szenzorok hibáinak milyen hatása van az irányításhoz nélkülözhetetlen állapotbecslésre vonatkoztatva, és milyen lehetőségeink vannak ezen hibák eliminálására.

A negyedik fejezetben a robotpilóta szoftver keretrendszerének megvalósítását mutatom be. A rendszer architektúra és a fontosabb taskok tulajdonságait és funkcióit az AMORES

[1] Unmanned Air Vehicle - ember nélküli légi jármű
[2] Autonomous Mobile Remote Sensing

projekt linuxbotnak nevezett fedélzeti szoftverén keresztül tárgyalom. A rendszer általános feladatai, követelményei után részletesen kitérek a szenzorok kezelésén túl az aktuátorok vezérlése és egyéb, a rendszerhez nélkülözhetetlen modulok megvalósítására.

1. fejezet

Az UAV-ok fedélzeti elektronikus rendszere

Ebben a fejezetben az UAV rendszerekben található fedélzeti elektronika fő komponenseit mutatom be.

1.1. Szenzorok

A robotnak valamilyen módon érzékelnie kell a külvilágot, hogy tudomást szerezzen saját állapotáról. Pozíciója, térbeli orientációja, repülési magassága, sebessége mind olyan információ, amit a robotpilótának ismernie kell a repülő vezetéséhez. Ezen mérések elvégzéséhez számos szenzor áll rendelkezésünkre.

1.1.1. Koordináta-rendszerek

Hogy a szenzorok jelét értelmezni tudjuk, definiálnunk kell a koordináta-rendszereket, amelyikben a mérés történik. Dolgozat további részeiben is ezekre a koordináta-rendszerekre fogok hivatkozni.

Inerciarendszerek

Az inerciarendszerek olyan koordináta-rendszerek, amelyekben érvényes Newton tehetetlenségi törvénye. Ilyen lehet például távoli csillagokhoz rögzített koordináta-rendszer, ahol figyelembe lehet venni a Föld forgását, keringését. Egy ilyen rendszerben leírni járművünk mozgását meglehetősen bonyolult, de kis méretekben jó közelítéssel inerciarendszernek tekinthetünk egy Föld, a felszínéhez rögzített koordináta-rendszert.

Ezt a síknak tekintett földfelszín egy rögzített pontjába helyezett origójú, jobbsodrású derékszögű koordináta-rendszert nevezzük továbbiakban Föld koordináta-rendszernek. Ahol nem egyértelmű, a Föld koordináta-rendszerben, röviden E-rendszerben, írt mennyiségeket E felső indexszel fogom jelölni.

Testhez rögzített koordináta-rendszerek

Sok esetben célszerű a mozgó járműhöz rögzített koordináta-rendszerben ábrázolni mennyiségeket. Ezt a koordináta-rendszert nevezzük test koordináta-rendszernek, angolul body frame-nek. Továbbiakban B felső index jelölést használom a test koordinátákban, továbbiakban B-koordináták, leírt mennyiségek jelölésére.

A MEMS[1] technológia fejlődésével manapság a legtöbb szenzor chip méretben gyártható, és költségük is drasztikusan csökken, ezért lehetőség nyílik sokféle szenzort használni. A MEMS technológia hibáiból fakadóan az ilyen szenzorokat többféle zavaró hatás éri,[2] ezért a pontosságuk elmarad a hagyományos mechanikai elven működő szenzorokhoz képest.

A következőkben bemutatom az UAV-ok fedélzetén leggyakrabban alkalmazott szenzorfajtákat. Azokat a szenzorokat, amivel a szenzor inerciarendszerhez képesti mozgásállapotának a megváltozását tudjuk mérni, inerciális szenzoroknak nevezzük. Ezek a szenzorok a test koordináta-rendszerben szolgáltatnak méréseket.

A gyorsulásmérő és a giroszkóp inerciális szenzorok, a belőlük készített rendszert inerciális szenzorrendszernek, röviden IMU-nak (Inertial Measurment Unit) nevezzük. Egy IMU-val képesek vagyunk egy test elmozdulásának hatszabadságfokú[3] (röviden 6DoF[4]) leírására.

A többi szenzor valamilyen külső, járművön kívüli hatást mér, jellemzően a Földhöz rögzített koordináta-rendszerben.

1.1.2. Giroszkóp

A giroszkóp, vagy pontosabban szögsebességmérő, a saját tengelye körüli szögsebességet képes mérni B-koordináta-rendszerben. Három egymásra merőleges tengelyű giroszkóp képes megadni a háromdimenziós térben a test szögsebesség-vektorát. Ha ismerjük a kiindulási orientációt, a szögsebességeket integrálva ideális esetben kiszámolhatjuk az új orientációt. A probléma az, hogy minden szenzor mérést zaj és torzítás terhel. A giroszkóp ω_m kimenete egy egyszerű modell alapján a következő komponenseket tartalmazhatja:[5]

$$\omega_m = m_g \omega + c_g + b_g + w_g \tag{1.1}$$

Ahol m_g a skálahiba, ω_g a szenzor koordinata-rendszerben mért szögsebessége, c_g a konstans ofszet hiba, b_g az időben változó ofszet, az úgynevezett bias, w_g pedig a mérést terhelő fehér zaj.

Ha a mért értékeket σ_0 szórású valószínűségi változók T_s mintavételi idővel vett sorozatának tekintjük, és ezt a sorozatot t időtartamra integráljuk, azt kapjuk, hogy az integrált érték szórása:

$$\sigma(t) = \sigma_0 \sqrt{T_s t}. \tag{1.2}$$

[1] Microelectromechanical systems

[2] Termikus zaj, Flicker-zaj

[3] Három szabadságfok a térbeli pozíció, további három a test állása, orientációja

[4] Six Degrees Of Freedom

Tehát a szórás, és így a hiba nagysága az idővel nő. Az ilyen jellegű hibát nevezzük random walk hibának. A gyártó általában a random walk értékkel szokta jellemezni a szenzort.[5]

Minél nagyobb a random walk érték, annál pontatlanabb lesz a számolt szögelfordulás, annál kevésbé lesz alkalmas orientáció számítására.

1.1.3. Gyorsulásmérő

A gyorsulásmérővel adott tengely mentén fellépő lineáris gyorsulást mérhetünk a szenzor B koordináta-rendszerében. Három egymásra merőleges tengelyű gyorsulásmérő a test gyorsulásvektorát adja. Nyugalomban, vagy egyenes vonalú egyenletes mozgásban lévő test mért gyorsulása megegyezik a Föld nehézségi gyorsulásával, tehát ilyen esetben részben megbecsülhető a test földhöz képesti orientációja.[6] Sajnos mozgó jármű esetében, melynek B-koordináta-rendszere nem tekinthető inerciarendszernek, ez nincs így. Ha más gyorsulás is fellép, például a repülő gyorsít vagy fordul, a mért vektorból nem tudjuk meghatározni, hogy az orientáció változott, vagy tényleges lineáris gyorsulás lépett föl.

A mért gyorsulás hasonlóképpen modellezhető, mint a giroszkóp esetén:

$$a_m = m_a a + c_a + b_a + w_a. \tag{1.3}$$

A szenzor gyorsulásához hozzáadódik egy állandó (c_a) és egy változó (b_a) ofszet, plusz a mérés zaja. Ha a gyorsulást integrálva sebességet, vagy tovább integrálva pozíciót szeretnénk meghatározni, a mérésre rakódott zajok a giroszkóphoz hasonlóan random walk jellegű hibaként jelennek meg.

1.1.4. Mágneses szenzor

A Föld mágneses tere a megfelelő szenzorokkal pontosan mérhető, a Földhöz rögzített koordináta-rendszerben mindig ismert irányba mutató mágneses vektor. A vektor iránya és nagysága a globális pozíció ismeretében táblázatból meghatározható.[7] Elméletileg a vektort mérve meghatározható lenne a jármű irányultsága.[8] A probléma hasonló, mint a gyorsulásmérő esetében, a mágneses erőteret befolyásoló külső hatásokat nem tudjuk megkülönböztetni a Föld mágneses terétől, így ezek zavarjelként terhelik a mágneses mérést.

A mágneses zavaroknak két lényeges fajtája lehet. Az egyik az úgynevezett hard-iron torzítás, amikor a Földön kívül egy másik mágneses tér is hat a szenzorra. Például állandó mágnesek, konstans áramú vezetékek okozhatnak ilyen hibát. A hard-iron torzítás hatása állandó ofszetként jelentkezik a mért mágneses vektorban. Ha ismerjük a mágneses forrásokat a jármű fedélzetén, az ilyen típusú hibák kalibrációval csökkenthetőek.

A másik típusú zavar, az úgynevezett soft-iron torzítás, a Föld mágneses terének torzulása révén jön létre. Ekkor megváltozhat a tér iránya és nagysága is, és az ofszet függhet a szenzor irányától is. Soft-iron hibát okoznak a ferromágneses anyagok, például vas vagy

[5]például $ARW = 2.0[\frac{deg}{\sqrt{(h)}}]$ Angular Random Walk

[6]A gyorsulás vektorra merőleges síkban nem kapunk információt, részletek a 3. fejezetben

[7]http://www.ngdc.noaa.gov/IAGA/vmod/igrf.html

[8]Gyorsulásmérőhöz hasonlóan a mágneses vektorra merőleges síkról nincs információnk.

nikkel közelsége.

1.1.5. GPS

Egy GPS (Global Positioning System, globális helymeghatározó rendszer) vevővel a Föld körül keringő műholdakhoz viszonyítva határozhatjuk meg a globális pozíciónkat. Egy műhold és a GPS vevő közötti jelátviteli idő ismeretében meghatározhatjuk a vevő és a műhold távolságát. Legalább négy műholdhoz viszonyított távolság alapján meghatározható a vevő térbeli pozíciója.

A GPS műholdak mindenkori pozíciója az úgynevezett almanach-ban van tárolva. Ezt ha nem ismeri a vevő, megkaphatja közvetlenül a műholdaktól,[9] vagy valamilyen egyéb forrásból, internetről vagy GSM hálózaton keresztül.[10]

A pozíciómeghatározás pontosságát sok tényező befolyásolja. A légkör paraméterei módosíthatják a rádiójel terjedésének az idejét, felléphet többutas jelterjedés, ami szintén növeli a terjedési időt. A látható műholdak száma, egymáshoz képesti helyzete, pályahibái illetve a vevő órájának pontossága mind okozhatnak hibát a mérésben.

A pozíció bizonytalanságát, szórását a GPS vevő adott esetben meg tudja becsülni műholdak által küldött illetve az almanach-ban tárolt információk alapján.

Két mért pont közti különbségből megbecsülhető a jármű elmozdulásvektora, ami segíthet az orientáció meghatározásában. Az egyik gond, hogy a helymeghatározás pontossága legjobb esetben is néhány méter, ami egy kisméretű repülő irányításához sok esetben nem elegendő. Továbbá a pozíció frissítés gyakorisága nem túl nagy, legfeljebb néhány Hz és két vett pozíció közti állapotról csak becslést adhatunk.

1.1.6. Ultrahangos távolságmérő

Az eszköz ultrahangot bocsát ki egy meghatározott irányba és a visszavert hullámot detektálja. A kibocsátás és detektálás közti idő alapján meg lehet mondani a reflexiós felület távolságát. A szenzor maximálisan néhány méteres távolságot képes mérni, leszállásnál hasznos lehet a repülési magasság becslésére. További korlátozás, hogy a visszavert ultrahangot befolyásolja a felület egyenessége, reflexiója, illetve a mérés pontossága függ a mért távolságtól. Viszint kis távolságok mérésénél jóval pontosabb, mint a GPS vagy a barometrikus nyomásmérő.

1.1.7. Barometrikus nyomásmérő

A légköri nyomás a magasság függvényében változik, tehát a nyomást mérve becsülhetjük a repülési magasságunkat. A tengerszint feletti magasság kiszámolható a képlettel

$$h = h_b + \frac{T_b}{L_b}\left(\frac{P}{P_b}^{\frac{-RL_b}{gM}} - 1\right), \tag{1.4}$$

[9]Amikor ez közvetlenül a vevő bekapcsolása után történik az az úgynevezett hideg indítás, és több percig is eltarthat.

[10]AGPS, Assisted GPS

ahol h_b a referencia magasság, T_b és P_b a referencia magasságon mért nyomás illetve hőmérséklet, $L_b = -0.0065$ K/m a hőmérséklet magasság-függését leíró állandó, P a mért nyomás, $R = 8.31432$ Nm/molK az univerzális gázállandó, g a nehézségi gyorsulás értéke, $M = 0.0289644$ kg/mol a moláris atom tömeg. Ez az egyenlet körülbelül 11 km magasságig jó közelítést ad.[9]

1.1.8. Légsebességmérő

Repülés közben az egyik legfontosabb információ a jármű levegőhöz viszonyított sebessége. Ettől függ a szárnyfelületekre ható felhajtóerő, ami az egyik legfontosabb információ a pilótának. Épp ezért a légsebességmérő a repülés kezdete óta minden repülőben megtalálható. Nincs ez másként a robotrepülőgépeknél sem.

Hagyományosan elterjedt konstrukció a légsebesség mérésre az úgynevezett Pitot-cső. Ez egy haladási iránnyal párhuzamos cső, aminek a belsejében és az oldalán van egy-egy barometrikus nyomásmérő. Az oldalsó szenzor méri az abszolút légnyomást, a csőben lévő pedig a torlónyomást, a sztatikus nyomással plusz a légsebesség négyzetével arányos tagból. A két nyomás értékből meg lehet becsülni a repülő levegőhöz képesti sebességét[11][6]

$$v = \sqrt{\frac{2(p_v - p_s)}{\rho}}, \tag{1.5}$$

ahol p_v a torló nyomás, p_s a statikus nyomás, ρ pedig a levegő sűrűsége. Természetesen ρ értéke változhat, függhet a magasságtól, hőmérséklettől, a levegő páratartalmától, ezért sok esetben ennél pontosabb modellre van szükség.

1.2. Robotpilóta

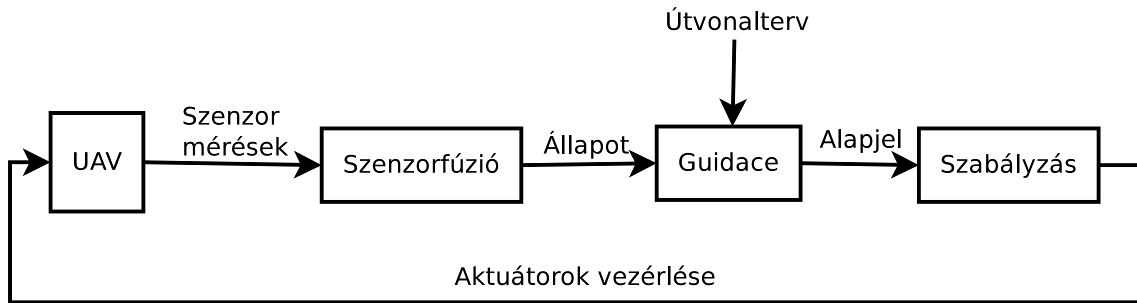
A robotpilóta a repülő azon része, ami az emberi beavatkozást és irányítást hivatott helyettesíteni. Ez az egység foglalja magába azokat az algoritmusokat, amik a repülő vezetéséhez és navigálásához feltétlenül szükségesek.

A robotpilóta három modulja szigorúan egymás után hajtódik végre, és minden modul használja az előző kimeneteit. Végeredményben az UAV irányítás egy visszacsatolt rendszer, hatásvázlata a 1.1 ábrán követhető.

1.2.1. Állapotbecslés

A probléma a szenzorok ismertetéséből látható: egyedi szenzorokkal nem lehetséges az orientáció és a pozíció pontos meghatározása, ami elengedhetetlen a jármű irányításához. A különböző szenzorok által mért, sok esetben zajos, pontatlan adatokból kell egy optimális becslést adni a repülő mozgásállapotára, absztraktabban fogalmazva meghatározni a rendszer állapotterét. A becslést megvalósító algoritmusokat nevezzük szenzorfúziónak. A vonatkozó fejezetben részletesen ismertetek néhány szenzorfúziós algoritmust.

[11] Ez az összefüggés csak akkor igaz, ha a levegőt összenyomhatatlannak tekintjük. Ez a közelítés kicsiny hibát okoz, ha a sebesség kisebb a hangsebesség harmadánál.



1.1. ábra. *UAV irányítás rendszere*

1.2.2. Guidance

A pilótának egy előre definiált pályán kell tartani a gépet. A pályakövetéshez szükséges vezérlő jeleket a guidance algoritmus határozza meg. Az aktuális pozíció és orientáció alapján a repülő képességeinek figyelembe vételével számoljuk ki a szükséges elmozdulást a pálya minél pontosabb követésére.

1.2.3. Szabályzás

A rendszer állapota és dinamikai modellje ismeretében ki kell számolni, milyen beavatkozó jelek szükségesek a kívánt alapjel követéséhez.

Egy jól működő szabályzás elengedhetetlen előfeltétele egy megfelelően pontos matematikai modell, ami leírja a jármű dinamikus viselkedését. Ez a modell nagyban függ a repülő fizikai felépítésétől, a szárnyfelületek, kormányfelületek tulajdonságaitól, és minden repülő típusra egyedi. Ezek a modelleket nemlineáris differenciálegyenlet-rendszer formájában adhatjuk meg.[4] A szabályzó feladata, hogy a dinamikai modell által szabott kényszerek és a bementek figyelembe vételével a rendszert az állapottér egy kívánt vektorába irányítsuk.

A szabályzó összetettségét meghatározza a kezelni kívánt manőverek „bonyorultsága". Minél radikálisabb beavatkozást akarunk végezni, annál jobban érvényesül a jármű nemlineáris tulajdonsága, annál bonyolultabb és számításigényesebb szabályzó-struktúrát kell alkalmaznunk az optimális irányításhoz.

1.3. Aktuátorok

Aktuátorok alatt a repülő mozgását befolyásoló eszközök tartoznak ide: a kormányfelületek mozgatását végző szervomotorok és a légcsavarokat hajtó motorok. Merevszárnyas repülő esetén a kormányfelületek állásszögének változtatásával a szárnyakra ható felhajtóerőt változtathatjuk, a légcsavarokkal pedig húzóerőt fejthetünk ki járműre.

A motorok vezérlése általános esetben PWM (Pulse-Width Modulation, impulzus szélesség moduláció) jellegű periodikus négyszögjellel történik, a négyszögimpulzus hossza határozza meg a motor fordulatszámát vagy a szervo állásszögét.

Az aktuátorok általában egytárolós taggal modellezhetjük. Az Az időállandó és az esetleges nemlinearitások, telítődések az egyes motorok paraméterei, az aktuátor minőségét jellemzik.

1.4. Payload

Payload, azaz „hasznos teher” minden olyan eszköz, amire nincs közvetlenül szükség a repülő vezetéséhez. Ide tartoznak a különböző kamerák, ezek mozgatórendszere (gimbal), mérőrendszerek, nem navigációs célú szenzorok (például gázszenzorok), szállító-konténerek, és tulajdonképpen bármi, ami miatt egy robotrepülőt használni szeretnénk.

1.5. Kommunikáció

Minden robotrepülő valamilyen rádiós csatornán kapcsolatban van egy földi irányító egységgel. Alapvetően háromféle rádiót különböztetünk meg, amik egymás mellett működnek:

- Telemetria rádió: kétirányú kapcsolat a repülő és a földi állomás között. A repüléshez szükséges adatok, vezérlő jelek, beállítások, pályák felküldésére, illetve státusz, telemetria adatok leküldésére szolgál. Fontos, hogy nagy megbízhatóságú legyen a kapcsolat.
- Payload rádió: Robottól a földi állomásra küldött payload adatfolyam. Lehet videó stream, mérési adatok, vagy bármi amit a feladat megkíván.
- RC vezérlés: hagyományos modellrepülőknél használatos távvezérlő, amivel felszállásnál, leszállásnál, vagy a robotpilóta meghibásodása esetén be lehet avatkozni a repülésbe. Közvetlenül az aktuátorokat vezérli.

2. fejezet

Amores hardver rendszer

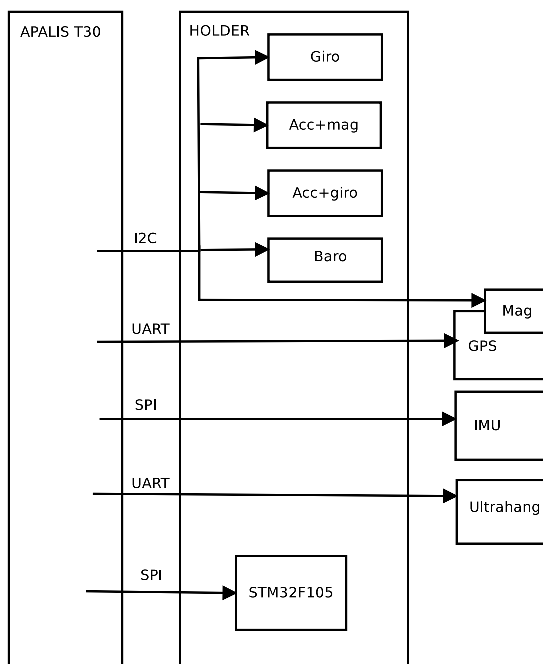
Az AMORES mobil robotikai technológiák fejlesztését megcélzó projekt keretében elkészült illetve fejlesztés alatt áll több robotpilóta rendszer. Két különböző fedélzeti elektronika készült el, különböző igényekre. Ezek hardverelemeit mutatom be a fejezet további részében.

2.1. Amores ProHD

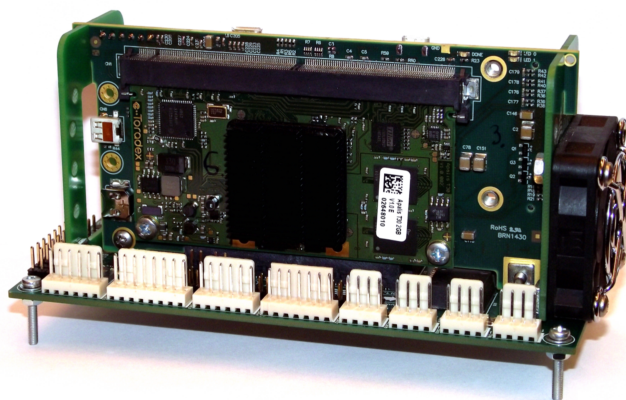
A elsődlegesen kifejlesztett UAV hardver rendszer egy Toradex gyártmányű, NVIDIA Apalis T30 system-on-module köré épül, ami önmagában tartalmaz egy négymagos ARM Cortex A9 processzort, 512 MB memóriát, NEON videó feldolgozó coprocesszort, és rengeteg egyéb perifériavezérlőt, ami egy komplett nagyteljesítményű beágyazott rendszerhez kellhet. Ehhez a modulhoz csatlakozik PCIexpress buszon keresztül egy Xilinx FPGA, ami a payload rádió megvalósításáért felelős. A modul legtöbb perifériája kivezetésre került: ethernet, USB, SATA, CAN, USART, I2C, SPI mind elérhető a modult fogadó központi elektronika, az úgynevezett holder csatlakozóin. A holder, a processzor modul és a szenzorok kapcsolatát a 2.1 ábra mutatja. A 2.2 ábrán a látható a teljes proHD rendszer, a 2.3 ábrán pedig a rendszer látható UAV-be építve.

A modulon Linux operációs rendszer fut.

A holderen helyet kapott egy STM32F105 mikrokontroller, ami az aktuátorokat vezérlő PWM jeleket állítja elő és az aktuátorokat összekötő CAN buszt vezérli. A buszra kisméretű, mikrokontrolleres CAN-RC átalakító kártyákat csatlakoztatva lehet vezérelni a szervókat és a motorokat. A CAN rendszer kiépítése a 2.4 ábrán látható. Egy kártya 4 aktuátort tud vezérelni és 8 kártya csatlakoztatható a buszra. A busz az erre fejlesztett RC-CAN kártyákkal fogadni is tud RC jeleket, így hagyományos RC távirányítóval is vezérelhetővé válik a rendszer. Továbbá szenzorokat is elhelyezhetünk a buszon, például a légsebességet mérő Pitot-csövet, a szenzorok olvasását szintén a mikrokontroller végzi. A mikrokontroller és a központi processzor között full-duplex SPI kapcsolat van, aminek az Apalis modul a mastere. Adott ütemezéssel küldi az üzeneteket melyet szétküld az egyes aktuátoroknak, miközben a mikrokontroller a gyűjtött adatok és az aktuátorok összes státuszát felküldi a masternek. Ha valamilyen oknál fogva nem kívánjuk használni a CAN rendszert, közvetlenül a holder PWM kimeneteire is irányíthatóak a kimenő jelek, illetve



2.1. ábra. *Toradex Apalis T30 modul, a holder és a szenzorok kapcsolata*



2.2. ábra. *AMORES ProHD*

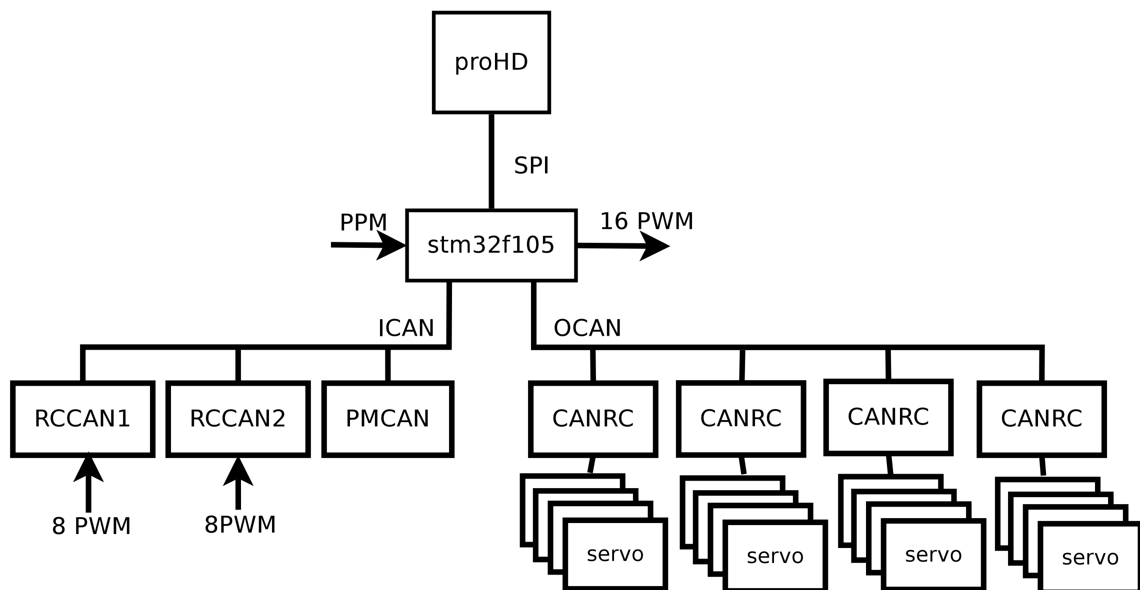
2.3. ábra. *A teljes fedélzeti elektronika X-8 UAV-ba beépítve teszt repülésre*

képes fogadni az RC technikában elterjedt impulzus pozíció modulációjú (PPM) vezérlő jeleket. A mikrokontroller képes a mastertől, CAN buszról vagy PPM forrásból jövő vezérlőjeleket bármely kimenetre elküldeni, az aktuális konfiguráció a masterről vezérelhető. További funkciója a mikrokontrollernek, hogy méri a holder feszültségeit és információt szolgáltathat az akkumulátor feszültségéről.

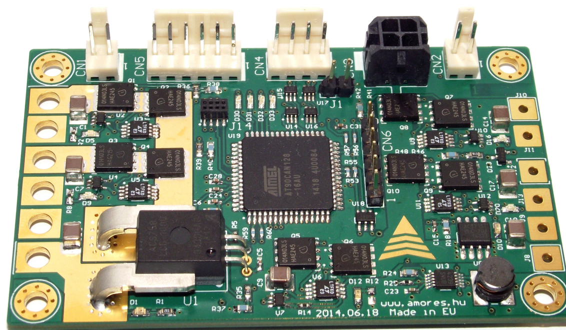
A különálló aktuátor vezérlő mikrokontroller értelme, ha a nagy bonyolultságú, Linux operációs rendszer alatt komplikált algoritmusokat futtató központi modul valamilyen oknál fogva meghibásodik, lefagy, az alapvető vezérlő funkciók továbbra is megmaradnak és egy földi operátor vagy másodlagos robotpilóta átveheti az irányítást. A fő rendszer meghibásodásának detektálása szintén a mikrokontroller feladata.

A renszer HDMI bemeneten keresztül képes HD videó forrást fogadni és a grafikus koprocesszoron keresztül H264 kódolású videó adatfolyamot előállítani. Ezzel lehetőség nyílik valós idejű, nagyfelbontású digitális videójel feldolgozására és továbbítására a payload rádión keresztül. Továbbá egy kiegészítő frame-grabber áramkörrel képes analóg videó bemenetet fogadni a processzor modul párhuzamos videó portján.

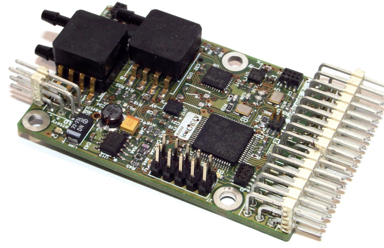
A rendszerhez tartozik még egy intelligens tápegység, ami az akkumulátoros tápellátás felügyeletéért felelős. Külön akkumulátor táplálja az elektronikát és a motorokat, ráadásul mindkét akkumulátor a redundancia miatt duplikálva van. A tápegység folyamatosan méri az akkumulátorok feszültségét, a modulok áramfelvételét, számontartja a fogyasztást és ezek alapján igyekszik optimálisan kapcsolni az akkumulátorokat. A modul külön mikrokontrollert tartalmaz, és I2C vagy CAN buszon keresztül kiolvashatóak az aktuális értékek illetve kapcsolhatók az egyes akkumulátorok. (2.5 ábra)



2.4. ábra. *CAN rendszer*



2.5. ábra. *AMORES Intelligens tápegység*



2.6. ábra. *AMORES Minibot*

A rendszeren található szenzorokat a szoftver ismertetésénel fogom tárgyalni a negyedik fejezetben.

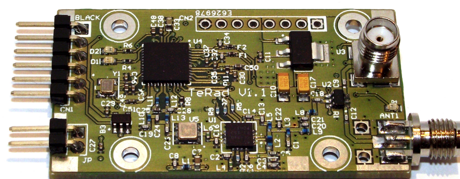
2.2. Amores Minibot

Az előzőnél egy jóval szerényebb képességekkel rendelkező, olcsó robotpilóta, ami rendelkezik egy repülő vezetéséhez szükséges összes funkcionális egységgel. Természetesen a szűkös erőforrások csak korlátozott teljesítményű állapotbecslő és szabályzó algoritmusokra adnak lehetőséget.

A rendszer egy kisméretű panelből áll, amin található a központi STM32F103 típusú mikrokontroller, gyorsulásmérő, giroszkóp, abszolút és differenciális nyomásmérő. Képes egy CAN buszra csatlakozni a ProHD rendszerrel és ugyanazokat az aktuátorokat ugyanolyan parancsokkal vezérelni. Így alkalmazható másodlagos, tartalék robotpilótának. (2.6 ábra)

Mindkét rendszer ugyanolyan telemetria rádión tartja a kapcsolatot a földi állomással. Egy 868 Mhz frekvencián kommunikáló, pont-pont összeköttetésre használható rádiópárral, ami transzparens soros UART adatátvitelt biztosít akár 10 km-es távolságba. Ezen keresztül egy mobil robotok kommunikációjára elterjedt, magas szintű, üzenet alapú protokollal, úgynevezett MAVLINK[1] üzenetekkel zajlik a kommunikáció. (2.7 ábra)

[1] Micro Air Vehicle Communication Protocol



2.7. ábra. *AMORES TERAD relemetria rádió*

3. fejezet

Szenzorfúzió

A megvalósított keretrendszernek feladata végső soron a robotpilóta algoritmusok futtatása. Ezért gyűjtjük a különböző szenzorokból az adatokat, hogy azt feldolgozzuk, felhasználjuk a repülő irányítására. A dolgozatnak nem célja az algoritmusok részletes ismertetése, de hogy a szenzor adatok célja világossá váljon, szükséges szót ejteni néhány szenzorfúziós algoritmusról, ezek céljáról, alkalmazásáról és tulajdonságaikról.

Ebben a fejezetben a repülő orintációjának és pozíciójának meghatározásának problémájáról lesz szó. Első lépésben ismertetem az alkalmazott koordináta rendszereket és egy test 3 dimenziós térbeli állásának, orientációjának leírására szolgáló módszereket. Utána bemutatom, az alkalmazott szenzorokkal milyen lehetőségeink vannak meghatározni az orientációt. Miután belátjuk, hogy önmagában egyetlen szenzorral nem vagyunk képesek megmondani a repülő orientációját, bemutatok néhány lehetséges megoldást több szenzort használó, úgynevezett szenzorfúziós algoritmusra.

3.1. Orientáció leírása

Orientációnak nevezzük egy test térbeli irányultságát. Precízebben kifejezve az orientáció az a forgatás, ami a test koordináta-rendszert Föld koordináta rendszerbe viszi. Az adott feladatnak megfelelően többféle leírásmódot találhatunk az orientáció leírására, hogy a továbbiakat tárgyalni tudjuk, ezeket röviden ismertetném.

3.1.1. Forgatási mátrix

Az orientációt kifejezhetjük a test és a Föld koordináta-rendszer közötti lineáris leképezés mátrixával. Ez egy 3x3-as méretű mátrix lesz, aminek az oszlopaiból képzett vektor a Föld koordináta-rendszerben a test koordináta-rendszer tengelyeinek irányába mutatnak.

$$R = \begin{bmatrix} R_{xx} & R_{yx} & R_{zx} \\ R_{yx} & R_{yy} & R_{zy} \\ R_{zx} & R_{zy} & R_{zz} \end{bmatrix}$$

A v^B vektor test koordináta-rendszerben felírt vektor Föld koordináta-rendszerbe transzformálva:

$$v^E = R_B^E v^B$$

Az R_B^E mátrixot szokás DCM[1] mátrixnak is nevezni. Ha másik irányban, a föld koordináta rendszerből szeretnénk áttérni a test koordináta rendszerbe, az $R_E^B = R_B^{E-1}$ összefüggéssel adható meg a forgatási mátrix. Mivel a koordináta-rendszerek ortogonálisak, a mátrix is ortogonális lesz. Ez előnyös számunkra, mivel a mátrix invertáltja meg fog egyezni a transzponáltjával, így az inverz numerikus számítása egyszerű.

Forgatási mátrix segítségével egyértelműen megadhatjuk egy test térbeli orientációját. Segítségével egyszerűen áttérhetünk a két koordináta-rendszer között. Hátránya, hogy relatív sok elemből áll, és az elemei nem függetlenek egymástól. Ez állapotteres leírás esetén numerikus problémákat vethet föl. Továbbá törekedni kell arra, hogy a forgatási mátrixokhoz kapcsolódó számítások eredménye ortogonális legyen. Ezek miatt a problémák miatt használunk egyéb leírás módokat is.

3.1.2. Euler szögek

A test orientációját megkaphatjuk, ha az E-rendszert saját tengelyei körül háromszor megfelelő sorrendben elforgatjuk. Egy adott orientációba 12 különböző forgatás-hármassal kerülhetünk, de a repülés technikában legelterjedtebb az x-y-z tengelyek menti forgatás, ezt a három szöget nevezzük Euler-szögeknek. A három szög neve és jelölése rendre ϕ bedöntési (roll), θ bólintási (pitch) és ψ irányszög (yaw).

A B-koordinátrendszerben mért szögsebesség vektor és az Euler-szögek megváltozása közti kapcsolatot az alábbi összefüggés írja le:

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin(\phi)\tan(\theta) & \cos(\phi)\tan(\theta) \\ 0 & \cos(\phi) & -\sin(\theta) \\ 0 & \sin(\phi)\sec(\theta) & \cos(\phi)\sec(\theta) \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}^B \tag{3.1}$$

Euler szögekkel tetszőleges orientációt le lehet írni mindössze három paraméterrel, de $\cos(\theta) = 0$ szögek esetén a ψ és θ szögek ugyanazon tengely körüli elfordulást írnak le, ezért az ilyen orientáció többféle Euler-szög hármassal leírható és ez szingularitást okoz az Euler-szögek változását leíró mátrixban. Ez numerikus instabilitást okozhat bizonyos orientációknál.[2] Ezért az Euler szögekkel való leírás szemléletessége ellenére csak korlátozottan használható.

3.1.3. Kvaterniók

A forgatási mátrixnál egy tömörebb, de szingularitás mentes leírást adhat ha az orientációt egy négy elemű valós vektorral, úgynevezett kvaternió segítségével írjuk le. Egy kvaternió a forgatás tengelye és az elfordulás szögének megadásával írja le az E és a B

[1] Direct Cosine Matrix

[2] Például ha $\theta = \pm 180°$

koordináta-rendszer egymáshoz képesti állását. Így tetszőleges orientációt írhatunk le négy paraméterrel, k egységvektorral és α forgatási szöggel:

$$q = \begin{bmatrix} \cos(\frac{\alpha}{2}) \\ k_x \sin(\frac{\alpha}{2}) \\ k_y \sin(\frac{\alpha}{2}) \\ k_z \sin(\frac{\alpha}{2}) \end{bmatrix} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} \tag{3.2}$$

A kvaternió változási sebessége és a ω szögsebesség vektor közötti kapcsolat:

$$\dot{q} = \frac{1}{2} \begin{bmatrix} -q_1 & -q_2 & -q_3 \\ -q_0 & -q_3 & q_2 \\ q_3 & q_0 & -q_1 \\ -q_2 & q_1 & q_0 \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}^B \tag{3.3}$$

A kvaterniókkal az orientáció leírása tömör, jól kezelhető, és bármely orientáció leírása szingularitás mentes. Ha három szabadságfokú rendszert négy paraméterrel írunk le, azok nem lesznek függetlenek. Ha állapotteres leírásban szeretnénk használni a kvaternióban adott orientációt, az állapottér kovariancia-mátrixa az összefüggő paraméterek miatt szinguláris lesz. Ha ez problémát okoz, találni kell a kvaterniók egy három dimenziós minimál-reprezentációját, amit alkalmazhatunk.

3.1.4. Rodrigez paraméterek

Ilyen minimál-reprezentációt ad, ha az egységkvaterniók által körbejárt négy dimenziós gömbfelületet egy három dimenziós hiperfelületre vetítjük. Megfelelő projekcióval 360°-os forgatást írhatunk le. Az így kapott paramétereket nevezzük Rodrigez-paramétereknek.[11]

A kvaterniók q_i és az s_i Rodrigez paraméterek közötti átváltást az alábbi összefüggések adják meg:

$$s_i = \frac{q_i}{1 + q_0}, i = 1, 2, 3 \tag{3.4}$$

$$q_0 = \frac{1 - s^T s}{1 + s^T s} \tag{3.5}$$

$$q_i = \frac{2 s_i}{1 + s^T s} \quad i = 1, 2, 3 \tag{3.6}$$

3.2. Orientáció mérése

Ebben a szakaszban bemutatok néhány lehetőséget, hogy lehetne egy adott szenzor segítségével megmérni az orientációt a szenzorok által mért adatok alapján.

3.2.1. Szögsebességek mérése

Giroszkóppal mérjük a repülő B koordináta-rendszerben fellépő szögsebességeket. A szögsebességeket T_s mintavételi periódussal mérve kiszámolható a T_s idő alatti orientáció változás.

Ha ismerjük a kiindulási orientációt, elméletileg az elfordulásokat akkumulálva meghatározható az orientáció.

Ez a módszer már első ránézésre is alkalmatlan a pontos orinentáció becslésére. Ha a szenzornak legcsekélyebb hibája van, ami ahogy láttuk az első fejezetben mindig van, a hiba az integrálás miatt az időben akkumulálódik, és a random-walk jelenség miatt nagyon hamar elromlik az orientáció. Ha egyszer a hiba túl nagy lesz, nem fogjuk tudni többé megmondani az orientációt kizárólag ilyen módszerrel.[12]

3.2.2. Vektormegfigyelések

Ha a test koordináta-rendszerben képesek vagyunk mérni egy E térben állandó referencia-vektort, abból tudunk következtetni a test térbeli állására. Egy vektorral két szabadságfokot rögzíthetünk, a vektor körüli elfordulást nem tudjuk megmondani, ezért legalább két ismert E koordináta-rendszerbeli vektor mérése szükséges.[8]

Ilyen referencia-vektorok lehetnek a föld gravitációs gyorsulása és mágneses tere. A gravitáció vektorát mérhetjük a fedélzeten gyorsulásmérővel, a mágneses teret pedig magnetométerrel.

Több matematikai eljárás is létezik a vektormegfigyelések alapján történő orientáció becslésre, mint például a gradiens módszer[7] vagy a Wahba-algoritmus.[13]

A probléma, hogy egyik vektort se vagyunk képesek pontosan megmondani. A Föld gravitációs gyorsulás vektorát nem tudjuk szétválasztani a repülő mozgásából fakadó gyorsulásoktól és a mágneses tér esetén is problémás az anomáliák elválasztása a Föld mágnesességtől.

3.3. Szenzorfúzió

A feladata továbbra is a jármű orientációjának meghatározása. Láttuk, hogy erre egyféle szenzor nem elég, több különböző szenzor mérési eredményéből kell a repülő állapotát megbecsülni. Az ezt elvégző algoritmust hívjuk általánosan állapotbecslőnek, vagy szenzorfúziós algoritmusnak. A repüléstechnikában a szenzorfúziót végző komponens neve AHRS[3]

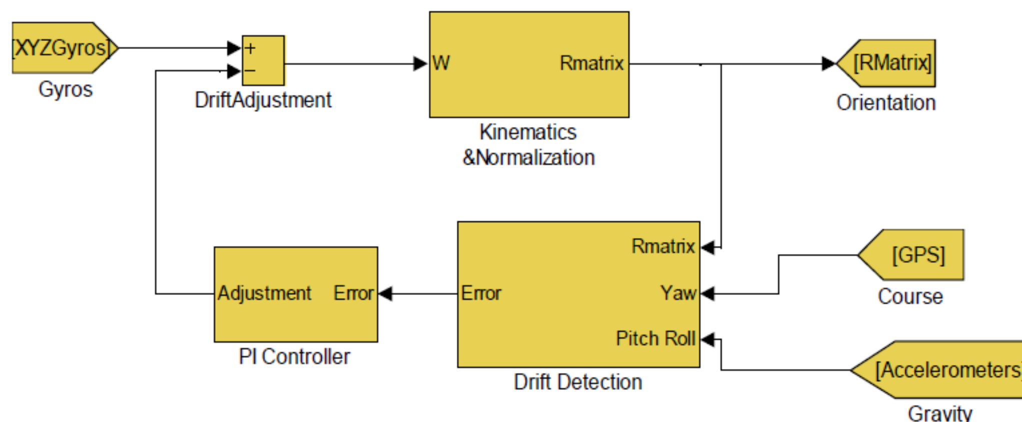
A szoftver keretrendszer úgy lett kialakítva, hogy az AHRS algoritmusok cserélhetőek legyenek. Így többféle implementáció kipróbálható.

A cél, hogy találjunk egy olyan algoritmust, amivel az egyes szenzorok hibáit kiküszöböljük és optimális állapotbecslést adjunk. A továbbiakban néhány ilyen algoritmust fogok ismertetni.

3.3.1. Ardupilot

Az Ardupilot[4] egy komplett, nyílt forráskódú robotpilóta megoldás, többféle járműre, saját földi infrastruktúrával és szoftverrendszerrel. Folyamatos közösségi fejlesztés alatt áll, sok kis teljesítményű, hobbi célú UAV használja. Az AHRS moduljának alapja a forgatási

[3] Altitude Heading Reference System
[4] http://ardupilot.com/



3.1. ábra. *Ardupilot által használt DCM algoritmus, forrás: [10]*

mátrix kiszámolása a szögsebességek alapján, majd a hiba korrigálása a gyorsulásmérő és a GPS alapján.

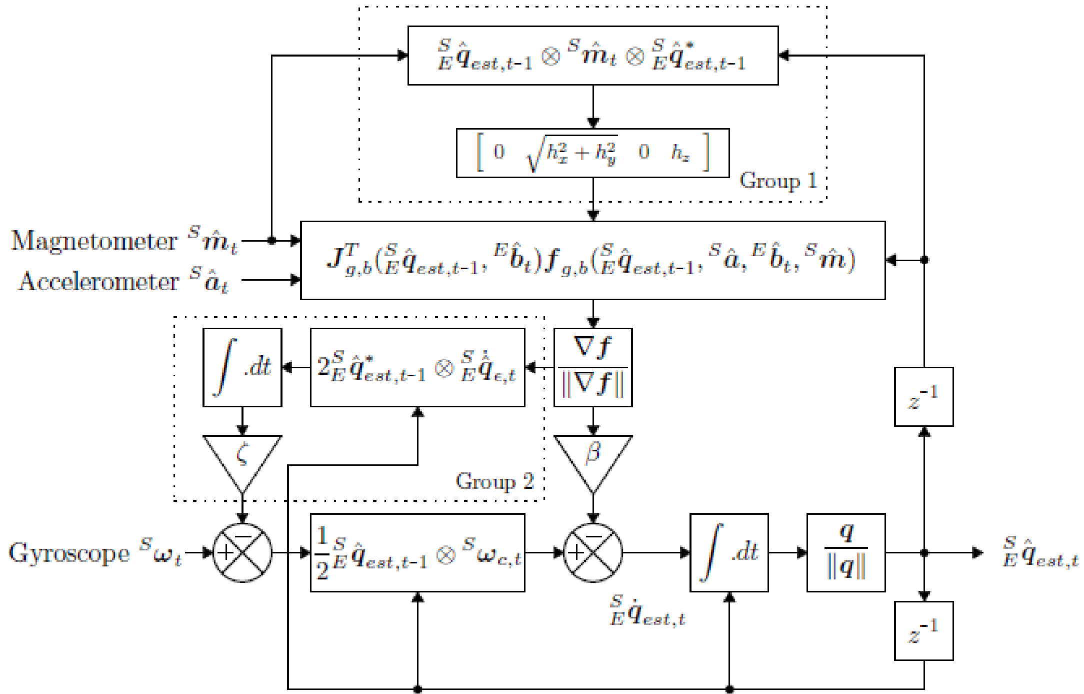
Az elsődleges jelforrás az orientáció meghatározására a giroszkóp szögsebessége. A szögsebességből integrálva meghatározozzuk a forgatási mátrix változását az előző mérés óta, kihasználva azt, hogy a DCM mátrix mindig ortogonális, normalizálással eliminálhatóak az integrálásból származó numerikus hibák. A gyorsulásmérővel mért gravitációs vektort függőlegesnek feltételezve megbecsülhetjük a bedöntési és a bólintási szöget, GPS vevővel pedig mérhetjük a Földhöz képesti sebesség vízszintes komponensét, amivel becsülhetjük az irányszöget. Az előzetesen becsült orientációhoz képesti hibát ilyen módon becsülhetjük, és a hibajelet egy integráló szabályzón keresztül felhasználjuk a következő mérés korrigálása.

Az algoritmusnak számos korlátja van. Feltételezi, hogy a repülő a Földdel párhuzamosan repül, és hogy nem hat rá külső gyorsulás. Ennek ellenére a gyakorlat azt mutatja, hogy sok esetben hatékonyan használható egyszerű robotpilótákban. Előnye az algoritmusnak, hogy nagyon egyszerű és kis erőforrás igényű, így kicsi, olcsó mikrokontrollerekben implementálható.

3.3.2. Madgwick AHRS

Nyílt forráskódú, nagyon elterjedt szenzorfúziós algoritmus, nevét Sebastian Madgwick-ról kapta, aki PhD dolgozatában publikálta 2009-ben.[7] Erőforrás igényében nem tér el az ardupilot megoldásától, de annál egy jelentősen precizebb állapotbecslést ad.

A algoritmus kvaterniókat használ az orientáció leírására. A megoldás a szögsebességből becsült orientáció változást pontosítja gravitációs vektor és a mágneses vektor megfigyelésével. A szögsebességekből a **??** összefüggés alapján számoljuk ki a kvaterniók megváltozását. A mért E rendszerbeli vektorokkal az úgynevezett gradiens módszerrel adunk becslést az orientációra.



3.2. ábra. *Madgwick AHRS felépítése, forrás: [7]*

3.3.3. Kálmán-szűrő

Az előző két eljárás speciálisan az orientáció meghatározására szolgáló szenzorfúziós algoritmus. Ismertetem az általános metódust, lineáris rendszer optimális állapotbecslőjét, az úgynevezett Kálmán-szűrőt.

Az algoritmus két lépésre bontható. Első lépésben az előző állapotból megbecsüljük az új állapotot, ezt nevezzük predikciónak. Második lépésként mérések alapján frissítjük a becsült állapotot.

A Kálmán-szűrő az állapotegyenletével adott lineáris rendszer pillanatnyi állapotát becsli az előző állapot, és megfigyelések, mérések alapján. Ehhez ismernünk kell a rendszer állapotegyenletét:

$$\dot{x}(t) = f(x(t), u(t), t) \tag{3.7}$$

$x(t)$ a rendszer belső állapotát leíró vektor, $u(t)$ a rendszer bemeneteinek vektora.

A rendszer belső állapotait legtöbb esetben nem tudjuk közvetlenül mérni, csak az ebből számolt mennyiségeket, a rendszer kimeneteit. Az állapotváltozók, a bemenetek és a kimentek közti összefüggést a kimenti egyenlet írja le:

$$y(t) = g(x(t), u(t), t) \tag{3.8}$$

Mivel az algoritmusokat processzoron, mintavételes rendszerben kívánjuk megvalósítani, a továbbiakban áttérünk diszkrét idejű leírásra. Ha f és g függvény lineáris, a rendszer és

a kimenti egyenlet az alábbi formában írható:

$$x(n+1) = Ax(n) + Bu(n) \tag{3.9}$$

$$y(n) = Cx(n) + Du(n) \tag{3.10}$$

Az A, B, C, D mátrixok a rendszert jellemző, úgynevezett rendszerleíró mátrixok.

A feladat $\hat{x}(n)$ becslő meghatározása a korábbi $\hat{x}(n-1)$ becslő, $u(n)$ bemenet valamilyen pontossággal ismert és $y(n)$ kimenet valamilyen pontossággal mért értékének ismeretében.

Első lépésben a Kálmán-szűrő kiszámítja a rendszeregyenlet alapján az $n+1$. időlépésre x^- előzetes becslést.

$$x^- = A\hat{x}(n) + Bu(n) \tag{3.11}$$

$$\hat{y}(n) = C\hat{x}(n) \tag{3.12}$$

Ezután a kimenet becsült és mért értéke közti

$$d = y(n) - \hat{y}(n) \tag{3.13}$$

különbség alapján egy megfelelő K_n súlyozómátrixal megszorozva pontosítja az állapotbecslést.

$$x^+ = x^- + K_n d \tag{3.14}$$

Ez a becslő lesz a következő időlépés állapotbecslője:

$$\hat{x}(n+1) = x^+ \tag{3.15}$$

A feladat K_n mátrix meghatározása minden időlépésben. Ez megtehető, ha a rendszer lineáris és időinvariáns, a bemenet és kimenet mérései pedig 0 várható értékű, Gauss eloszlású fehér zajjal terheltek. Ekkor minden lépésben kiszámolható a $\hat{x}$ állapottér kovarianciamátrixa,

$$P_n^- = AP_{n-1}A^T + Q_n, \tag{3.16}$$

ahol Q_n a mérést terhelő zaj kovariancia mátrixa, amiből zárt alakban kiszámolható a visszacsatoló K_n mátrix.

$$K_n = P_n^- C^T (CP_n^- C^T + R_n)^{-1} = P_n^+ C^T R_n^{-1} \tag{3.17}$$

Ezek az egyenletek, mint látható, nagy számítás igényű mátrix műveleteket használnak. Ezért megfelelő teljesítményű hardver szükséges az állapotbecslés számításához.

Sajnos a repülő dinamikai modellje nem lineáris,[4] ezért a Kálmán-szűrő ilyen formában nem használható.

3.3.4. Nemlineáris Kálmán szűrő

Ha a rendszeregyenlet vagy a kimeneti egyenlet nem lineáris, vagy a rendszert támadó zaj nem 0 várható értékű fehér zaj, az eljárás ugyan az, a becslés egy méréssel kerül korrigálásra a K_n súlyozómátrixszal visszacsatolva, a különbség hogy nem tudjuk kiszámolni a mátrix optimális értékét zárt alakban, pusztán egy közelítő értéket tudunk rá adni. A Kálmán-szűrőnek több nemlineáris kiterjesztése terjedt el a ezekre mutatok egy példát.[3]

Extended Kalman Filter (EKF)

A módszer a nemlineáris rendszer munkapont linearizálásán alapul. Minden időlépésben kiszámoljuk a becsült állapotvektor körül a rendszeregyenlet és kimeneti egyenlet lineáris közelítését, majd az így kapott lineáris rendszeregyenletből számoljuk ki a K visszacsatoló mátrixot.

Az EKF algoritmus elsőrendben jó közelítést ad, de a linearizálásból fakadóan a becslés nem optimális. Ha a kiindulási állapot nem megfelelő, előfordulhat, hogy a becslés divergálni kezd. Továbbra is érvényes, ha a zaj nem 0 várható értékű fehér zaj, a becslés nem alkalmazható.

Mindezek ellenére az EKF algoritmus sok esetben megbízhatóan használható, és számos AHRS implementáció előszeretettel alkalmazza[5].

3.4. Összefoglalás

Ezeken túl még sok különböző szenzorfúziós algoritmus létezik, különböző előnyökkel, hátrányokkal, egyedi tulajdonságokkal. De egy jól működő szoftveres keretrendszerrel elkülöníthetjük ezeket a rendszer többi részétől. A cél, hogy az algoritmusok fejlesztése közben ne kelljen az operációs rendszer, ütemezés, adatgyűjtés, szenzorok egyedi tulajdonságait figyelembe venni, ezek egy jól kezelhető és bővíthető interfész mögé legyenek rejtve. Természetesen ezek a megállapítások ugyanúgy vonatkoznak a szenzorfúziós algoritmusokon túl a guidence és szabályzó algoritmusokra is. Az elkészített szoftverrendszer ezeket a követelményeket teljesíti és kellő alapot nyújt egy fejlett UAV robotpilóta fejlesztésére.

[5] Újabban az Ardupilot is ezt használja

4. fejezet

Szoftveres keretrendszer

Az UAV fedélzeti rendszerének sarkalatos pontja a rendszer egészét átfogó szoftver. A hardver komponensek kezelését, szenzorok adatainak feldolgozását, beavatkozó szervek vezérlését, robotpilóta algoritmusok futtatását egy összetett szoftver végzi. Ebben a fejezetben ezt a szoftver rendszert fogom ismertetni.

A robotpilóta implementálásához kidolgozásra került egy keretrendszer, aminek feladata a szenzorok és beavatkozó szervek időzített olvasása illetve vezérlése, állapotbecslő, szabályzó és guidance algoritmusok megfelelő ütemezéssel történő futtatása és a földi állomással való kapcsolattartás.

Az UAV irányításához elengedhetetlen a gyors működés. Amint a szenzor mérések elkészülnek, az adatokat minél hamarabb fel kell dolgozni és megfelelőképp reagálni az aktuátorokon keresztül. Ennek a folyamatnak kiszámíthatónak kell lennie, nem engedhetjük meg, hogy a beavatkozás késlekedjen. Ezért az olyan operációs rendszert kell választani, ami ezt a kritériumot teljesíti.

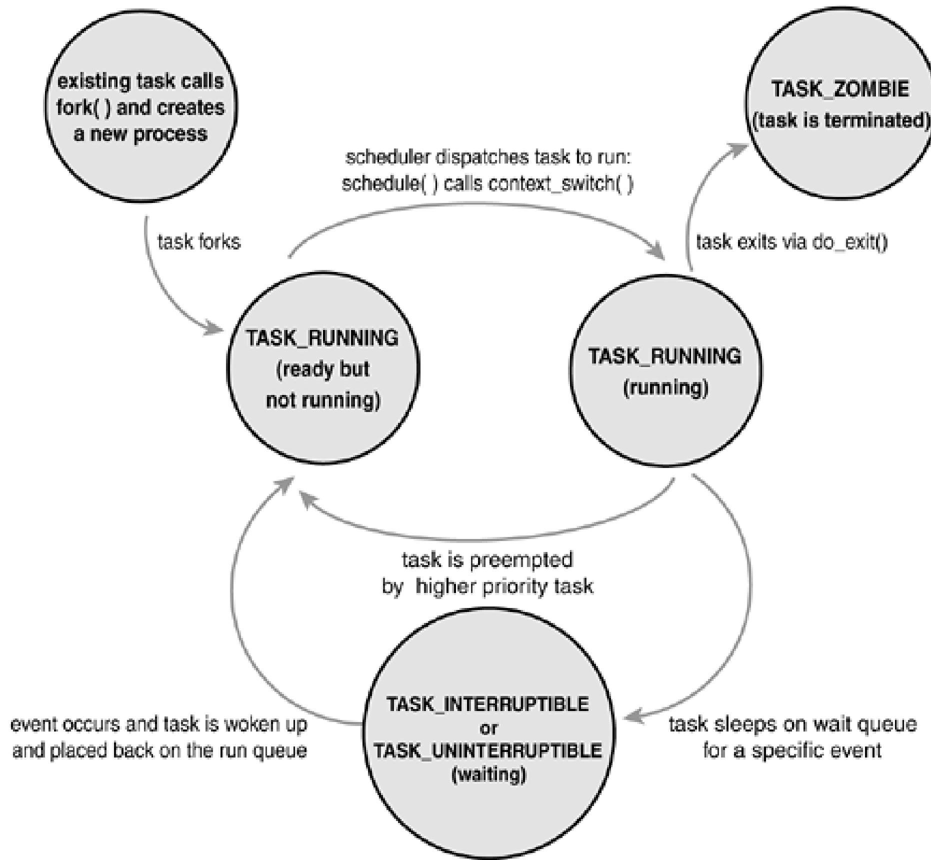
Az ilyen rendszereket nevezzük real-time rendszernek, ami azt jelenti, hogy az összes folyamat végrehajtási idejének pontosan kézben tarthatónak kell maradnia és minden folyamatnak adott idő belül le kell futnia.

A keretrendszer elkészült linux operációs rendszer alatt, POSIX IPC[1] eszközök felhasználásával, és STM32F103 mikrokontrollerre freeRTOS[2] operációs rendszert használva. A két keretrendszer között csak implementációs különbségek vannak, logikailag és architektúrálisan megegyzik a két szoftver, így tárgyalható közösen.

A freeRTOS egy kisméretű, beágyazott rendszerekhez fejlesztett real-time operációs rendszer. Ingyenes, forráskód szinten elérhető rendszer, gyakorlatilag bármilyen architektúrára lefordítható. Egész kis méretű, csak egy preemptív[3] ütemezőt és taskok közötti kommunikációt megvalósító eszközöket, várakozási sorokat, szemaforokat tartalmaz.

A linux alapvetően nem real-time operációs rendszer, de a kernelt megfelelő opciókkal lefordítva lehetőség van real-time taskok futtatására. Továbbá fontos, hogy a rendszerben futó folyamatokat az erőforrásokat kihasználva mind kezelni és felügyelni tudjuk. Így a linux operációs rendszer is megfelel a valós idejű követelményeinknek.

[1]Inter Process Communication
[2]http://www.freertos.org/
[3]Kivétel nélkül mindig a legnagyobb prioritású aktív task fut



4.1. ábra. *Linux processzek lehetséges állapotai. Forrás:[2]*

A továbbiakban az összetettebb, Linux alatt implementált rendszert, az úgynevezett Linuxbot-ot mutatom be. A rendszer alapvetően a szenzorokkal méréseket végez, ezeket feldolgozza majd az aktuátorokat vezérli, közben menedzseli a kommunikációs feladatokat, payload adatokat, naplózza és kezeli a hibákat.

4.1. Multitask alkalmazások Linux alatt

Mivel sok egymás mellett működő szenzort használunk, követelmény a feladatok, taskok párhuzamos ütemezése. Amikor egy program elindul, létrejön egy egyedi ID-val rendelkező process, ami saját virtuális memóriával rendelkezik. Az operációs rendszer egyik fő feladata a futó processzeket ütemezni, az erőforrásokat kezelni. Ütemezéskor a rendszerben jelenlevő, futásra kész taskok közül lesz kiválasztva a következő futó task. A taskok lehetséges állapotai és a köztük lévő átmenetek a 4.1 ábrán látható.

Egy processz készíthet magáról másolatot a fork() rendszerhívással. Ekkor létrejön egy úgynevezett gyerek processz, saját virtuális memóriával, ami a szülő másolata, és a továbbiakban ugyanúgy részt vesz az ütemezésben. Ezzel a módszerrel hozhatunk létre több szálon futó, multitask alkalmazásokat linux alatt.[4]

Multitask alkalmazásoknál meg kell küzdenünk a taskok közötti kommunikáció prob-

[4]Használhatunk még könnyűsúlyú processzeket, úgynevezett szálakat is

lémájával.[5] Mivel a taskok saját címtérrel rendelkeznek, közvetlenül nem érik el egymás változóit, az operációs rendszer által biztosított szolgáltatásokkal kell megvalósítani a kommunikációt.

Linux alatt is elérhető, általános IPC mechanizmusok:

Osztott memória Kijelölhető egy közös memória terület, amit mindkét processz elér. Ezen keresztül lehet lebonyolítani a kommunikációt, adatcserét. Ezzel a módszerrel sok adatot tudunk gyorsan átvinni, de a párhuzamos elérés miatt adatinkonzisztencia léphet föl. Ezért az adatelérést atomivá kell tenni.

Várakozás sor Egy várakozási sor FIFO[6] jellegű adatátvitelt tesz lehetővé két processz közt. A rendszerben létrejön egy speciális fálj, amibe az egyik task bele tud írni, a másik task pedig várakozni tud üzenet érkezésére, és ki tudja olvasni a legrégebben beleírt adatot.

Szemafor Szemaforral megvalósítható az erőforrások védelme kölcsönös kizárással. Egy szemafor objektumot egyszerre egy task birtokolhat. Ha közben egy másik task is el akarja érni, a task alvó állapotba kerül egészen addig, amíg a szemafor fel nem szabadul. Amíg egy task alvó állapotban van, nem vesz részt az ütemezésben. Ezzel a mechanizmussal jelzéseket küldhetünk taskok között osztott erőforrásokat, például memóriaterületet védhetünk vele.

A linuxbot program is több párhuzamos task-ból áll. Miután a főprogram a megfelelő paraméterezéssel elindul és a konfigurációs állományt beolvassa, elindítja a gyerek processzeket.

4.2. Perifériák kezelése Linux alatt

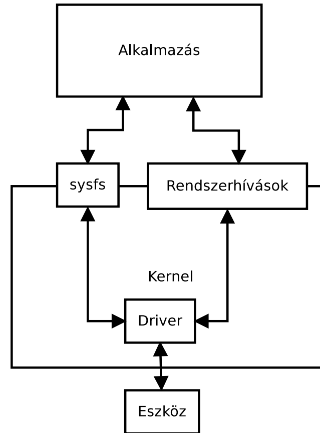
A különböző szenzorok különböző perifériákként csatlakoznak a rendszerhez, és a feladat, hogy a mért adatok minél kisebb késleltetéssel eljussanak a felhasználói programhoz.

Linux alatt az alkalmazások nem látják közvetlenül az eszközöket. minden futó processz saját virtuális memóriával rendelkezik, ezen kívüli memória címeket nem érhet el, így a perifériákhoz sem férhet hozzá.

A rendszer magja, a kernel, biztosítja a periféria-kezeléssel kapcsolatos alacsonyszintű funkciókat, buszok és megszakítások kezelését. A külső eszközök valamilyen szabványos buszon, I2C, SPI, UART stb. keresztül csatlakoznak a rendszerhez. A busz kezeléséhez kapcsolódó általános funkciókat a kernel biztosítja, de az eszközspecifikus feladatokat úgynevezett eszközmeghajtókban, driverekben kerülnek implementálásra. A driverek a rendszerbe modulokként illeszkednek és futás közben is betölthetőek. A driverek feladata, hogy menedzselje az eszközt és kapcsolatot létesítsen a felhasználói alkalmazások és a specifikált busz műveletek között. A felhasználói programok és a kernel közötti kommunikáció főbb lehetőségei:

---

[5] IPC - Inter Process Communication
[6] First In First Out



4.2. ábra. *Alkalmazások és eszközök kapcsolata Linux rendszerben*

Rendszerhívások Az eszközök egy fálj leíróval kerülnek regisztrálásra a rendszerbe, általában a /dev könyvtárba. Ezen a leírón keresztül speciális függvényekkel, rendszerhívásokkal, léphetünk interakcióba a driverrel. A rendszerhívások funkciója definiálva van, de a konkrét működést a driver implementációja határozza meg.[7]

Például egy driverben ami egy szenzort kezel, implementálva van az open(), read() és write() függvény. Az open függvénnyel lefutnak az inicializáló rutinok, a read függvénnyel kiolvassuk a legutóbbi mérési eredményt, a write fügvénnyel konfigurációs adatokat küldhetünk a szenzornak. A szenzor kezelésének részleteit elrejti a kernel, az alkalmazás csak a szabványos interfészt látja.

Ezzel a módszerrel megvalósíthatunk egészen összetett funkciókat, aszinkron kommunikációt például poll() rendszerhívással, vagy saját adastuktúrákat mozgathatunk a kernel és a user-space között.

sysfs A linux kernel biztosít egy virtuális fáljrendszert, amin keresztül a kernel egyes alrendszereit lehet elérni hierarchikus elrendezésben. A fáljrendszer a /sys könyvtárban található. Ebbe a fáljrendszerbe lehet a driver egyes paramétereit exportálni, amik hagyományos állományokként jelennek meg.

Ez a módszer jól alkalmazható az eszköz egyes tulajdonságainak beállítására, információk kiolvasására.

mmap Ha nagy sebességgel, nagy mennyiségű adatot mozgatnánk a kernel és a user-space között, kijelölhetünk egy közös memóriaterületet, amit a driver és az alkalmazás is elér az mmap rendszerhíváson keresztül. A módszer előnye hogy gyors, viszont nagy körültekintést igényel a megvalósítása.

Sok periféria hasonló módon kezelhető. Például ha két szenzor UART-on keresztül bájtokat küld, amiket az alkalmazás olvas, nem szükséges külön driver mindkét eszközhöz, kezelhető egy közös generikus driverrel. Ilyen általános driverek a legtöbb buszhoz a kernelben alapértelmezetten rendelkezésre állnak és használhatóak. Saját kernel driver írásába

[7] A rendszerhívások teljes listája például itt megtalálható itt: http://man7.org/linux/man-pages/man2/syscalls.2.html

akkor kell belevágni, ha valamilyen speciális vezérlést igényel az eszköz. A linuxbot alkalmazásban a legtöbb periféria kezelése generikus driverekkel megoldható, de ha a szenzort saját külső megszakítása ütemezi, szükséges saját eszközspecifikus driver.

4.2.1. Megszakítás kezelés

Külső eseményeket feldolgozó, real-time rendszereknél nagy kihívás az eseményekre válaszoló gyors reakció. Processzor alapú rendszerekben a külső események jelzésének módja a megszakítás. Az esemény bekövetkeztekor, például ha egy szenzor elkészült egy méréssel, a processzor futása megszakad, és egy eseményhez rendelt megszakítás-kezelő függvény lefut. Operációs rendszer által menedzselt környezetben a megszakítások kezelését is az operációs rendszer, így a kernel kezeli. A probléma, hogy a kernel nem kezdeményez kommunikációt az alkalmazásokkal, az alkalmazásnak kell figyelnie, hogy érkezett-e megszakítás.

Szenzorok esetén tipikusan a megszakítás lekezelése után egy kiolvasás történik, majd a kiolvasott adatot kell átmásolni a kernelből az alkalmazás címterébe. Ha read() rendszerhívással olvassuk az eszköz leírót, és így várjuk, hogy az adat előálljon, a futó processz blokkolódni fog amíg nem érkezik meg az adat, és más feladatok nem fog tudni elvégezni.

Ezt úgynevezett aszinkron kommunikációval lehet elkerülni. A poll() vagy select() rendszerhívással az eszköz leírón lehet várakozni egy flag-re, hogy egy adat csomag előálljon, és read() függvénnyel kiolvasható legyen. A kernel driverben a megszakítás kezelő függvényből beállítható az aszinkron olvasást ütemező flag, így a megszakítás eljuthat a felhasználói alkalmazáshoz.

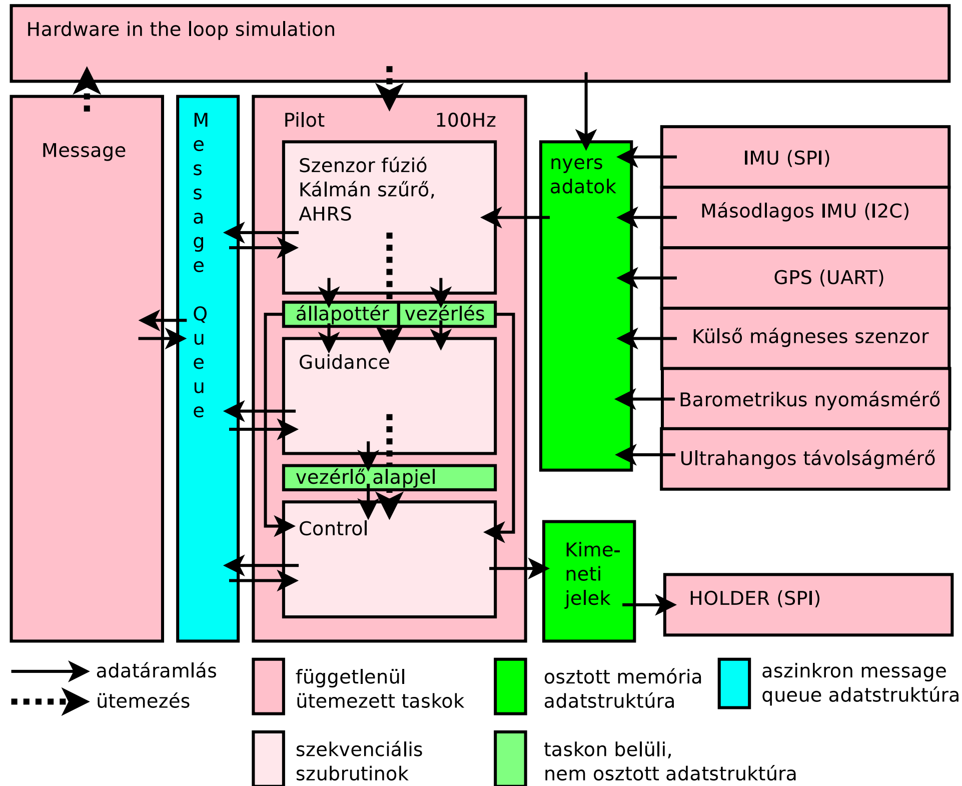
4.3. A Linuxbot

Az AMORES proHD rendszeren futó robotpilóta keretrendszer az úgynevezett Linuxbot. A multitask szoftver vezérli a teljes fedélzeti elektronikát, olvassa a szenzorokat, kezeli a kommunikációs csatornákat és futtatja a robotpilóta algoritmusokat.

A szoftver architektúrája a 4.3 ábrán látható. A fő program elindulása után kiolvassa az egyes modulokra vonatkozó konfigurációs állományt és az alapján elindítja az a szükséges taskokat. A központi pilot tasknak, ami a robotpilóta algoritmusokat futtatja, kötelező futnia, a többi task a rendszer kiépítésétől és elérhető szenzoroktól függően engedélyezhetőek.

A taskok közötti adatáramlás osztott memóriaban tárolt struktúrákban valósul meg. A szenzor kezelő taskok a bejövő szenzor adatokat a bemeneti struktúrákba másolják a saját ütemükben, az algoritmusokat futtató pilot tast pedig szintén a saját ütemében olvassa ki őket. Miután a pilot task lefut, a kimenetét szintén egy osztott memóriába másolja, amit a kimeneti, aktuátor vezérlő taskok olvashatnak. Az egyes taskoknak lehet saját belső állapota. Ezt a taskhoz tartozó lokális adatstuktúran tárolja, és message queue segítségével érhetőek le más taskok számára.

A program C nyelven íródott, a Linux IPC szolgáltatásait használva a multitasking megvalósításához. A fejlesztés alatt nagy segítség, hogy a program könnyen lefordítható ARM architektúrára és futtathatjuk a proHD rendszeren, illetve lefordítva x86-os rendszerre és



4.3. ábra. *Linuxbot rendszer architektúrája*

így a fejlesztésre használt PC-n is futtatható a program.[8]

A következő szakaszban a rendszer egyes taskjainak működését ismertetem.

4.3.1. Bemeneti szenzor taskok

A szenzorok különböző módon, különböző mintavételezéssel mérik az adatokat, ezért minden szenzor egy független processzt kap, ami az adott szenzornak megfelelő ütemezéssel fut.

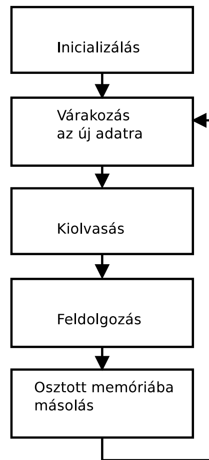
A task indulásakor lefut egy inicializáló függvény, ami regisztrálja az eszközt, megbizonyosodik az elérhetőségéről, beállítja a szenzor konfigurációs regisztereit és az ütemezés forrását.

Ezen ötemezés forrása lehet az operációs rendszer, ha például adott periódussal ki lehet olvasni a szenzor regisztereit, vagy ütemezhet maga a szenzor: ha elkészült egy méréssel, megszakítást küld a processzornak, és az adatgyűjtő task lefut.

Amikor a szenzor olvashatóvá válik, a task kiolvassa a mért adatokat. Az adatok a task egy lokális memória területére kerülnek.

Az egyes szenzorokhoz adott esetben tartalmazhat előfeldolgozást, szűrést és hiba kezelést, ezek is a szenzorhoz tartozó taskban kerülnek végrehajtásra. A szenzorok mérései történhetnek egészen más ütemezéssel, mint a feldolgozó algoritmusok ütemezésre. Ezért minden szenzor task adatstruktúrájához tartozik egy úgynevezett dirty flag, amit új adat beérkezésekor 0-ra állít. Ebből az olvasó task látja hogy elkészült egy új mérés és akár

[8]Szimulátor módban, vagy külső perifériák nélkül



4.4. ábra. *Szenzor taskok folyamatábrája.*

számolhatja hány ciklus telt el a legutóbbi adat óta. A dirty flag-et lehet a hibakezelés után hibajelzésre is használni.

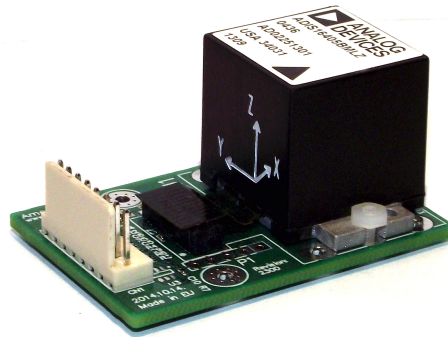
A szenzor taskok feladata a különböző szenzorok adatainak mértékegység konverziója, hogy az osztott memóriába az előre definiált, koherens mértékegységek kerüljenek, például az összes mágneses szenzor adat *mgauss* egységekben legyen. További feladat, hogy az iránnyal rendelkező mennyiségek azonos koordináta rendszerben legyenek értelmezve, ezért a szenzorok állása alapján a mért vektorokat megfelelően kell korrigálni. Az erre vonatkozó információk egy konfigurációs állományban beállíthatóak.

A mért és feldolgozott adatok egy osztott memória területre kerülnek, amit más processzek is elérhetnek, és a saját ütemükben olvashatják. Az osztott memória kezelése a POSIX shm API segítségel történik. Minden processz pontosan egy adatstruktúrával rendelkezik, ami kötelezően tartalmaz egy szemafort. Ezzel a szemaforral valósítható meg a kölcsönös kizárás, így ha több task akar egyidejűleg hozzáférni a struktúrához, fenntartható az adatokkonzisztenciája. A szenzor taskok általános folyamatábrája a 4.4 ábrán követhető.

4.3.2. Elsődleges inerciális szenzor

A rendszer egyik legfontosabb szenzora, az Analog Devices ADIS16405 tipusú inerciális szenzorrendszer. Ez tartalmaz három szabadságfokú giroszkópot, gyorsulásmérőt és magnetométert. A szenzorok gyárilag kalibráltak, nagy pontosságúak. A szenzorok a belső AD konverterrel 819.2 Hz-el vannak mintavételezve, amit egy modulon belüli átlagoló szűrő 100 Hz-re decimál. A 100 Hz-es frekvenciával előálló adatok SPI buszon keresztül jutnak a rendszerbe.

Ennek a szenzornak a méréseit használja elsősorban az állapotbecslő algoritmus, az adatok pontosan ütemezett beolvasása fontos. Ezért a használjuk a szenzor DATA_READY megszakítását, ami egy GPIO porton keresztül jut be a rendszerbe. A szenzor a megszakítást az AD konverter mintavételezésének ütemében küldi, ezért a decimálást a szűrő beállításának megfelelően a kernelben kell elvégezni. Ha beérkezett a kellő számú megszakítás, a kernel SPI tranzakciót kezdeményez, és a szenzor kimeneti regisztereinek tartalmát egy buszciklus alatt kiolvassa. Az adatcsomag megérkezéséről poll() rendszerhíváson ke-



4.5. ábra. *ADIS16405 imu*

resztül értesíti a kernel az adatgyűjtő taskot, majd read() rendszerhívással kiolvashatóvá válik.

A driver konfigurációját iocntl() rendszerhíváson keresztül végezhetjük el. Ezzel egy parancsot és hozzá tartozó adatstruktúrát küldhetünk a drivernek. Ezen az interfészen keresztül módosíthatjuk a szenzor konfigurációs regisztereit, például a szűrő paramétereit állíthatjuk, ekkor SPI tranzakciók zajlanak le. Itt módosíthatjuk a driver tulajdonságait, például letilthatjuk a megszakítás kezelést és olvashatjuk a szenzort a kernel által adott ütemezéssel.

A szenzor összes funkciója egy interfész mögé van rejtve, így a rendszerhívások megfelelő paraméterezése és hibavédelme biztosítva van.

4.3.3. Másodlagos inerciális szenzorok

A külső fő IMU-n kívül tartalmaz a rendszer több kisebb, olcsó szenzort is. Szerepük a fő szenzor mérésének validálása, esetleges meghibásodása esetére tartalékolás. Ezek a repülő fedélzeti elektronikájának része. Itt található a mobil eszközökben népszerű ST gyártmányú L3GD20H giroszkóp és az LSM303D gyorsulásmérő és magnetométer modul, továbbá egy Invensense MPU6000 giroszkóp és gyorsulásmérő. A szenzorok közös I2C buszon csatlakoznak a rendszer többi részéhez.

A szenzorok szabványos I2C interfészt használnak, ami kezelésére a kernelben megtalálható általános I2CDEV driver megfelelő. Ezen a driveren keresztül read() és write() függvényekkel olvashatunk illetve írhatunk bájt sorozatokat a buszra. Hogy az adott tranzakció melyik buszon lévő eszköznek szól, a tranzakció előtt egy ioctl() hívással lehet kiválasztani.

A szenzorok olvasását a task ütemezi, a program indulásakor beállítható frekvenciával. Minden ciklusban a task kezdeményez minden modulhoz egy I2C tranzakciót, és kiolvassa kimeneti regisztereiket, ami a legutóbbi mérés eredményét tartalmazza.

4.3.4. GPS

Az UBLOX gyártmányú GPS modul a központi elektronikán kívül kapott helyet és UART interfészen keresztül kapcsolódik a rendszer többi részéhez. A GPS szabványos NMEA[9] mondatokat küld. Ez egy sztring alapú protokoll, ami elsősorban tengerészeti rendszerek közötti kommunikációra lett kifejlesztve, de a GPS modulok is nagyrészt ezt a protokollt használják. A GPS-t kezelő taskot a soros vonalon beérkező bájtok ütemezik. Az UART interfész kezelésére a linux kernelben generikus driver áll rendelkezésre, így annak fejlesztésével külön nem kell foglalkozni. Aszinkron I/O mechanizmusokkal, open(), poll() és read() rendszerhívásokkal a soros vonalon érkező karakterek beolvashatók. A fogadott bájtok a taskon belül egy pufferbe kerülnek, ha összegyűlt egy szabványos NMEA mondat,[10] a task értelmezi és a kinyert hasznos információt az osztott memóriába másolja. A kinyerhető adatok közül a legfontosabb a globális pozíció, szélesség-hosszúság-magasság koordináta rendszerben (LLA), de ezen kívül megkapjuk a földhöz viszonyított sebességet, a mérés becsült pontosságát, pontos időt, vételi viszonyokra vonatkozó adatokat, műhold információkat is.

4.3.5. Külső mágneses szenzor

Mivel a mágneses szenzor nagyon érzékeny a különböző mágneses zavarokra, így a közelében lévő elektronika mágneses térerősségére, érdemes a magnetométert minél távolabb helyezni a központi elektronikától. Ilyen céllal kerül egy mágneses szenzor a GPS modul illetve a Pitot-cső mellé. Mindkét szenzor a nagy pontosságú Honeywell HMC5883-as iránytű modul. A GPS melletti szenzor a másodlagos szenzorokhoz hasonlóan I2C buszon keresztül kommunikál a processzor modullal és állandó 100Hz-es ütemezéssel kerül kiolvasásra.

Mivel a Pitot-cső a CAN alrendszerbe illeszkedik, a mellette lévő magnetométer is a CAN buszon keresztül küldi a mérés eredményeit. Így azt a CAN rendszerrel kommunikáló task kapja meg és másolja az osztott memóriába.

4.3.6. Barometrikus nyomásszenzor

A központi elektronikán, a másodlagos inerciális szenzorok mellett található a barometrikus nyomásszenzor, amivel az abszolút tengerszint feletti magasságot lehet meghatározni. A szenzor az MS5611 típusú, egyedileg gyárilag kalibrált szenzormodul, saját belső hőmérővel. A gyári kalibrációs kvóciensek az eszköz saját memóriájában vannak tárolva, ahonnan kiolvashatóak. Ezen adatok és a szenzor hőmérséklete alapján korrigálhatóak a kiolvasott adatok. A kalibrációs folyamat a szenzor adatlapjában megtalálható.

A modul I2C buszon keresztül csatlakozik a rendszer többi részéhez. Az olvasásért felelős task 10 Hz-es periódussal fut és végzi a mérés korrigálását. A kiolvasott nyomás értékből itt számol a rendszer magasságot. A felszállás előtt kalibrálni kell a mérést, hogy a felszállás helyéhez legyen viszonyítva a repülési magasság. A magasság mérése ehhez a referencia

[9] National Marine Electronics Association

[10] Például egy mondat lehet az alábbi karakter sorozat:
$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47

kiindulási ponthoz mért nyomás különbségből lehet kiszámolni a valós magasságot.

4.3.7. Külső AHRS rendszer

Az inerciális szenzorok és az orientáció becslő algoritmus működését egy nagy pontosságú XSENS gyártmányú külső AHRS modullal tudjuk ellenőrizni, ezért fejlesztés alatt és teszt repülések alkalmával egy ilyet alkalmazunk.[11] Egy ilyen rendszer komplett AHRS megoldást nyújt és kész, feldolgozott pozíció és orientációs adatokat biztosít, DCM mátrix, Euler-szögek vagy kvaterniók formájában. Természetesen a nyers giroszkóp, gyorsulás és mágneses adatokat megkaphatjuk belőle.

Az XSENS modul UART-on keresztül küldi adatait a rendszernek, és saját bináris protokolját használja. A kezelő taskját a beérkező bájtok ütemezik, és ha beérkezett egy adatcsomag, azt értelmezi és az osztott memóriába másolja.

4.3.8. Ultrahangos távolságmérő

Az alkalmazott ultrahangos távolságmérő modul tartalmazza az ultrahang adó és vevő elektronikán kívül a feldolgozáshoz szükséges mikrokontrollert, ami folyamatosan számolja a mért távolságot. Az eredményt UART-on keresztül küldi a rendszernek, ami a centiméterben mért távolság. A néhány Hz periódussal érkező adatok ütemezik az olvasó taskot, ami a távolság alapján elvégez egy szórás becslést és az osztott memóriába másolja a távolsággal együtt.

A szonzor 20-750 cm közötti távolságokat képes megmérni. Minél nagyobb a távolság, annál nagyobb lesz a mért érték szórása.

4.4. Pilot task

A rendszer fő modulja a szenzorfúziót, guidance-t és szabályzást megvalósító pilot task. A három rutinnak szigorúan egymás után kell lefutnia, ahogy az 1.1 ábra mutatja.

Az aktuátorok vezérlése PWM jellel történik, aminek a periódusideje 10 és 20 ms között van. A periódusidőnél nagyobb felbontású vezérlő jeleknek nincs értelme, ezért robotpilóta algoritmusokat elegendő 10 ms ütemezéssel futtatni.

A szenzorfúzió bemenete a szenzor taskok által az osztott memóriába írt legutóbbi adatok, és az előző pilot ciklus alatt kiszámolt, a jármű orientációját és pozícióját becslő állapottér. A szenzorfúzió a ciklus végén előállítja a mérési adatokkal frissített állapotteret.

A guidance modul az ismert berepülendő pálya, és az aktuális állapottér alapján határozza meg a repülő szükséges irányváltozásait, azaz a szabályzó alapjelet. A pálya telemetria csatornán keresztül feltölthető illetve repülés közben módosítható.

A szabályzó az alapjelekből a repülő dinamikai modellje és a közvetlen vezérlőjelek ismeretében meghatározza a manőverhez szükséges kimeneti jeleket, a beavatkozó szervek

[11] Az ok, hogy külön szenzorokat és algoritmusokat alkalmazunk, ahelyett hogy egy készen kapható és többször bizonyított rendszert alkalmazunk, az XSENS modulok igen borsos ára. Az tesztek során alkalmazott modul ár hozzávetőlegesen 3000 euro

(légcsavarok, kormányfelületek) beállítását. A kimeneti jelek egy osztott memóriába íródnak.

A pilot task interfészének lényeges része:

```
pilot_sfusion_init();
pilot_guidance_init();
pilot_control_init();
while(1) {
    pthread_cond_timedwait(&cond, &lock, &timeout);
    /* Step the control loop */
    pilot_sfusion_step();
    pilot_guidance_step();
    pilot_control_step();
}
```

A keretrendszer úgy lett megírva, hogy az egyes szenzorfúzió, guidance, szabályzó algoritmusok könnyen cserélhetőek legyenek, így többféle algoritmus és implementáció összehasonlítható. A modulokat, a különböző pilot algoritmusokat statikus függvénykönyvárként kell a programhoz linkelni.

A pilot algoritmusok bonyolult mátrix műveleteket hajtanak végre. Ezek hatékony elvégzése a LAPACK[12] függvény könyvtár segítségével történik. Ezek a gyakran használt lineáris algebrai műveletek olyan alacsonyszintű implementációját tartalmazza, ami az adott architektúra jellemzőit a lehető leghatékonyabban kihasználja. A FORTRAN nyelven megírt algoritmusokat az adott hardverre az ATLAS[13] programcsomag segítségével lehet optimalizálni és lefordítani, hogy végeredményül a C programból elérhető interfészt kapjunk. Ennek az interfésznek a függvényeit használhatjuk a mátrix műveletek elvégzésére.

4.5. Üzenetkezelő task

Repülés közben a repülő küld magáról különböző státuszüzeneteket a földi állomásra, például az aktuális pozíciót, orientációt, akkumulátor feszültségeket, diagnosztikai adatokat, a földi állomás pedig vezérlő jeleket, pálya információkat küldhet a járműnek. Az üzenetek csomagolására használt protokoll a kifejezetten kisméretű légi járművek vezérlésére kitalált MAVLINK[14] protokoll. Az üzenetek kezelését a rendszerben egy külön task végzi és a message task üzenetsorok segítségével kommunikál a többi task-kal. Az üzenetsorok az operációs rendszer POSIX message queue szolgáltatásával vannak megvalósítva. Az üzenet forrása beírhat a sorba, aki olvassa az üzenetsort, mindig a legrégebben beleírt üzenetet kapja meg. Ezzel a mechanizmussal taskok közötti kommunikáció valósítható meg.

Az üzenetek érkezhetnek UART porton keresztül, vagy hálózaton, TCP/IP csomagok formájában. A használt csatorna és paraméterei konfigurációs állományból beállítható. Az üzenetkezelő taskot ütemezhetik a beérkező üzenetek. Ha beérkezett egy üzenet, a modul

[12] Linear Algebra Package [1]
[13] Automatically Tuned Linear Algebra Software
[14] Micro Air Vehicle Link

értelmezni és attól függően, hogy melyik tasknak szól, a megfelelő várakozási sorba írja. A fogadó task, amikor eljut arra pontra, kiolvassa a sort, és megkapja a neki szóló üzenetet.

Az egyes taskok is kezdeményezhetnek üzeneteket. Ekkor a task beleírja a várakozási sorba az üzenetét, az üzenetkezelő pedig, miután kiolvasta a sort, összeállítja az üzenetnek megfelelő MAVLINK csomagot, amit a telemetria csatornán keresztül elküld.

Ha egyik task se kezdeményez üzenetet, az üzenetkezelő akkor is periodikusan küld magáról státus üzeneteket (heartbeat), amiben a jármű azonosítója, állapota és néhány fontos információ található.

4.6. Hardver in the loop szimuláció

A keretrendszer kiegészül egy hardware-in-the-loop szimulátorral, ami használatával a rendszer beilleszthető egy MATLAB szimulációs környezetbe. Ebben az üzemmódban az adatgyűjtő szenzor taskok és aktuátorok le vannak tiltva, szenzorok helyett a szimulátor szolgáltatja az adatokat, és írja azokat az osztott memóriaterületre. A pilot kimeneti adatait az osztott memóriából a szimulátor visszaolvassa. Mivel ebben az esetben a pilot futási sebességét nem korlátozza az aktuátorok válaszideje és a szenzorok adatsebessége, a pilot task ütemezését a szimulátor veszi át. Ezzel a módszerrel lehetőség van az algoritmusok helyes működésén túl tesztelni az implementáció helyességét és vizsgálni a hardver teljesítményét, illetve MATLAB környezetben a számított adatok könnyen feldolgozhatóak és ábrázolhatóak.

4.7. Holder task

A rendszer aktuátorai, a motorok és a szervók, nem közvetlenül a linuxbot alól kapják a vezérlőjeleket hanem a holderen található mikrokontroller[15] állítja elő őket. A linuxbot feladata pusztán a megfelelően összeállított adat csomagok elküldése a mikrokontrollernek. Ezt egy fix periódusidővel, alapértelmezetten 100 Hz, futó task végzi. A task feladata az SPI busz vezérlése, a mikrokontroller konfigurálása a szervó vezérlő jelek kiolvasása az osztott memóriából és elküldése, válaszadatok fogadása.

Az SPI busz kezelése a linux kernel generikus SPI driverével[16] történik, full-duplex módon. Egy adatcsomag küldése közben, ugyanabban a buszciklusban kapja meg a task a mikrokontroller aktuális állapotát. A minden ciklusban cserélt adastuktúrában benne van a linuxbot által küldött 16 PWM csatorna értéke, a kimeneteire jutó PWM értéke illetve az mikrokontroller összes lehetséges bemenetének (két RC-CAN kártya, PPM bemenet, másodlagos robot) értéke, továbbá a pillanatnyilag a CAN rendszerbe csatlakozó kártyák ID-je. Így a linuxbot az összes elképzelhető bemenetet látja, viszont fontos, hogy a linuxbotnak nincs joga eldönteni, aktuálisan melyik bemeneti csatorna kerül a kimenetre. Az adatstuktúra tartalmazza továbbá a mikrokontroller által mért feszültségeket, az elektronika akkumulátorának, és a rendszer 5V és 3.3V-os tápfeszültségét.

[15]STM32F105
[16]SPIDEV

Ha a rendszerben van PM-CAN[17] kártya, annak az adatai is a struktúrában vannak, közvetlenül olyan formában, ahogy szenzor előállította. Ennek a feldolgozása szintén a linuxbot feladata.

Mivel az adatcsere sikeressége létfontosságú, minden adatcsomag tartalmaz egy ellenőrző összeget, amivel az esetleges hibákat detektálni lehet. Hiba esetén az adatcsomag újraküldésre kerül.

A minden ciklusban kicserélt adatstruktúra a következő:

[17] Pitot-csővel és magnetométerrel felszerelve

```
typedef struct {
    uint8_t cmd;
    uint32_t cnt;   /* number of the transfer. Increment by 1 in every transfer */
    uint16_t robot_pwm[PWM_CHANNEL_NUM];    /* pwm value generate by proHD */
    uint16_t input_pwm[5][PWM_CHANNEL_NUM]; /* input channels to prHD */
    uint16_t servo_pwm[PWM_CHANNEL_NUM];    /* Actual servo control signal */
    uint16_t system_voltages[4];
    uint8_t pm[8];
    uint8_t cards[CARD_MAX_NUM]; /* Cards in the can bus, 7-4bit: type; 3-0bit id */
    uint16_t crc;
} canstm_data_t;
```

A rendszer indulásakor elküldésre kerül egy konfigurációs struktúra, amiben a linuxbot elküldi a mikrokontroller konfigurációját, amit egy fáljban adhatunk meg. Itt be lehet állítani, hogy az egyes CAN-RC kártyák kimenetén melyik bemenetek jelenjenek meg. Egy dedikált bemeneten lehet beállítani, például RC távirányítóval, hogy melyik bemenet vezesse a repülőt. Ennek a csatornának a számát is itt adhatjuk meg.

Látható, hogy a helyes működéshez elengedhetetlen a linuxbot és a mikrokontroller szoftverének kompatibilitása. Ezt a linuxbot a verzió kezelő rendszer[18] alapján ellenőrzi. A verziószám az első konfigurációs üzenetben kerül lekérdezésre.

Itt megemlítendő, hogy a rendszer többi része felé a mikrokontrontoller, SPI és CAN busz transzparensen viselkedik, az osztott memóriába közvetlenül az aktuátorokra jutó jel reprezentáció kerül, jelen esetben a vezérlő PWM jelek impulzus hossza, a mikrokontrollernek semmilyen beleszólása nincs a vezérlés tulajdonságaira.

[18]SVN

5. fejezet

Eredmények

A linuxbot keretrendszer jelenlegi formájában alkalmas az összes szenzor egyidejű olvasására, taskok kezelésére. Tesztek, mérések és robotpilóta algoritmusok kipróbálására a linuxbot rendszert használjuk.

A feladatom a szoftver keretrendszer és a szenzorok adatgyűjtű taskjainak implementálása volt, ezért olyan méréseket mutatok be, amin látszik a rendszer működése.

A Linuxbot szoftvert és a rendszerhez kapcsolódó szenzorokat több méréshez használtuk, mint adatgyűjtő rendszert. A proHD-n futó Linuxbot egy egyszerű pilot modult futtatott, aminek egyetlen célja a mért adatok SD kártyára írása. A kinyert adatok később MATLAB-ban feldolgozhatók, ábrázolhatók.

A 5.1 5.2 5.3 ábrákon egy több szenzor adatait használó mérés eredményei láthatóak. A különböző szenzorokat egyidejűleg futtatva a giroszkópok és gyorsulás szenzorok adatai közel megegyeznek. A giroszkópok kimenetén látható tüskék egybeesnek a gyorsulásmérőkön látható gyorsulások helyével. Ez a várt működés, hiszen szögsebességek csak tengely körüli elforduláskor lép föl. Ebből azt is látjuk, hogy az mérési adatok egyszerre kerülnek rögzítésre, a bemeneti taskok tényleg párhuzamosan dolgoznak.

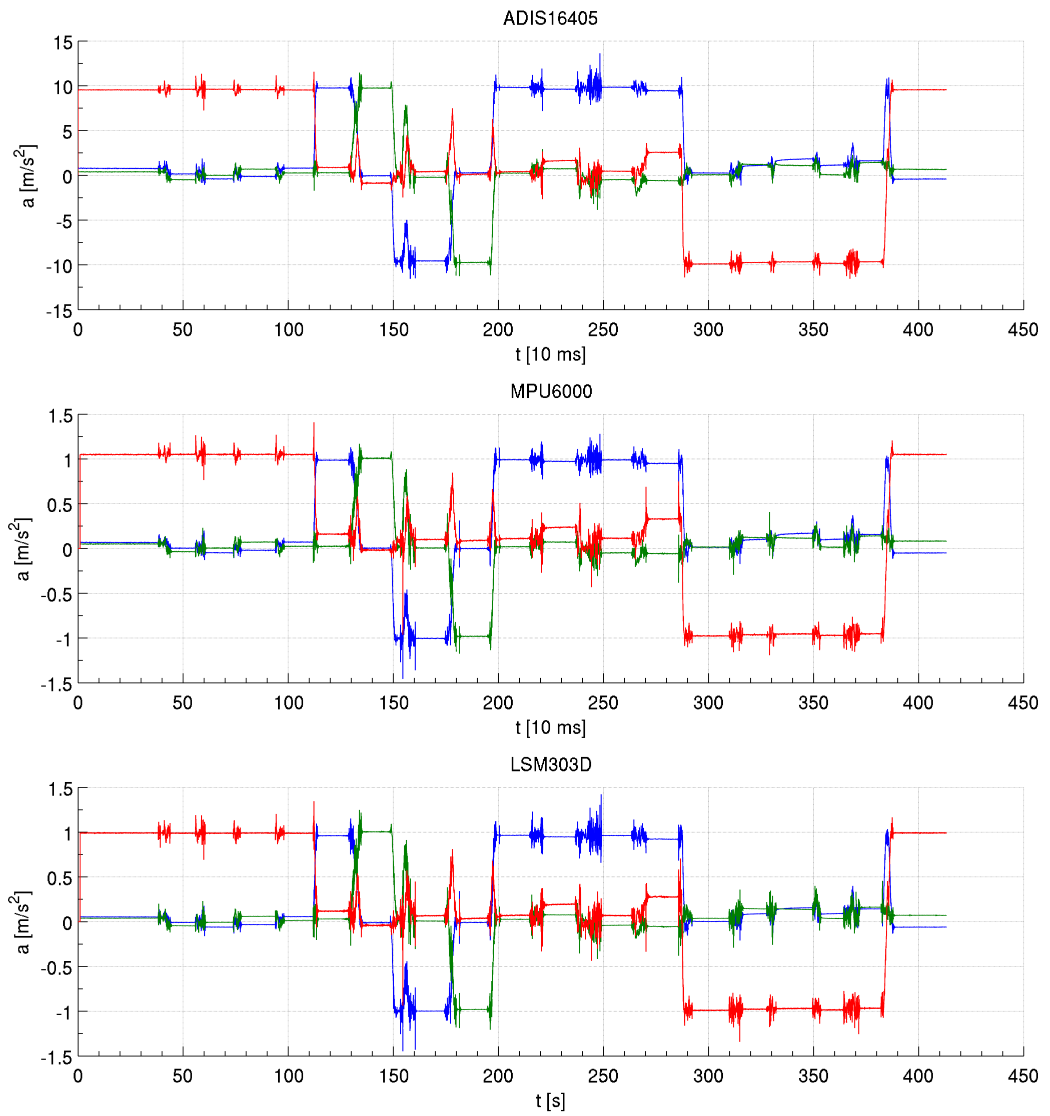
A magnetométerek adatainak különbségének az oka, hogy külső (HMC5886) magnetométer orientációja más volt illetve különböző zavar éri a különböző helyen elhelyezett szenzorokat.

Az adatgyűjtő alkalmazás használható a szenzorok a szórásának, ofszetjének vizsgálatára és az adatlapban adott értékek validálására.

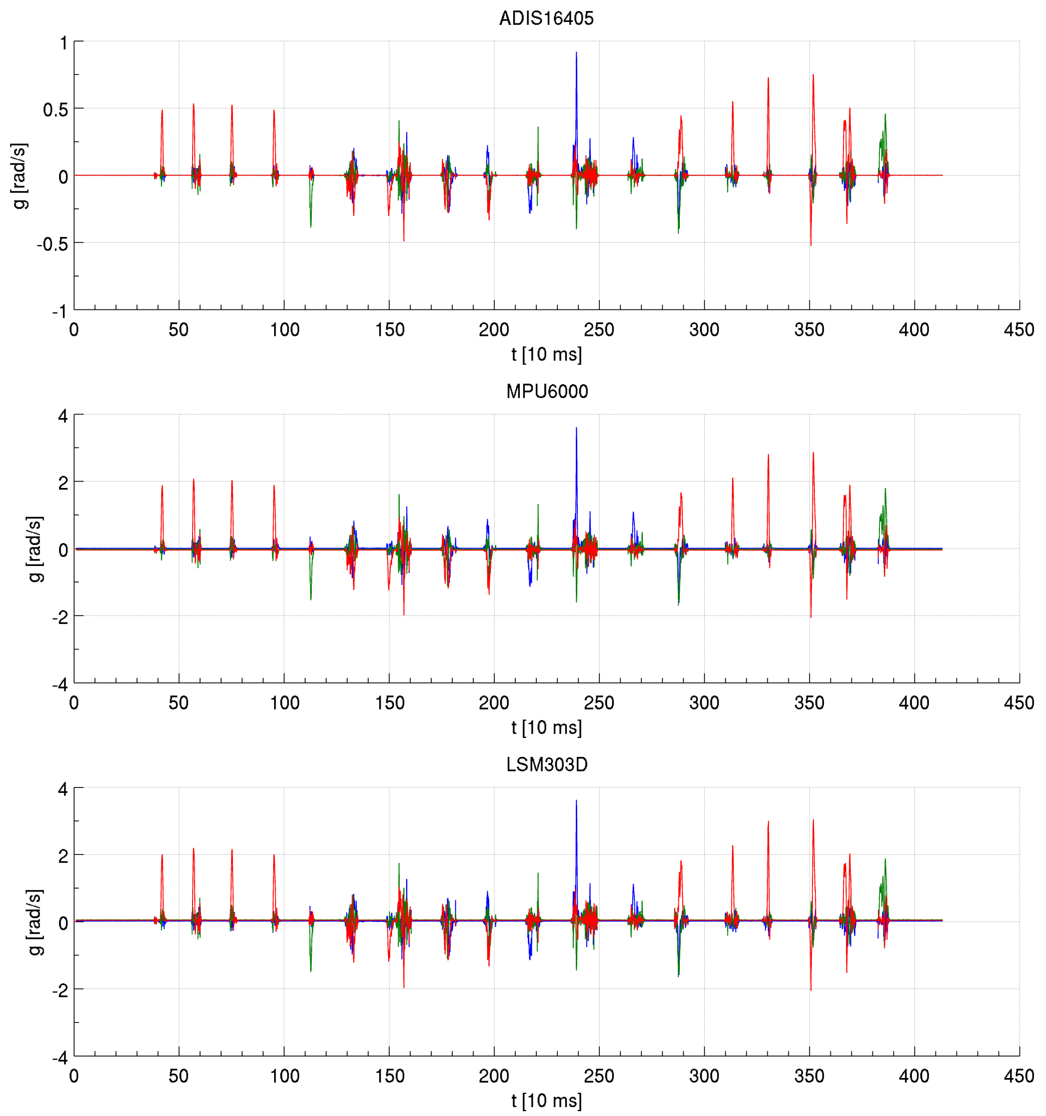
A 5.4 ábrán például a külső IMU mechanikus felfüggesztésének rázópados vizsgálatának egy mérése látható. A különböző frekvenciájú lineáris gyorsulással gerjesztve a gerjesztő gyorsulás és a mért gyorsulás ismeretében kiszámolható a rendszer átviteli függvénye.

Egy másik mérés során rázópadon az IMU 10 Hz-el lett rázva, így egy tengely mentén lineáris gyorsulás hat a rendszerre. A 5.5 ábrán látható a gyorsulásmérő és a giroszkóp kimenete és vizsgálható a gyorsulás giroszkópra gyakorolt hatása.

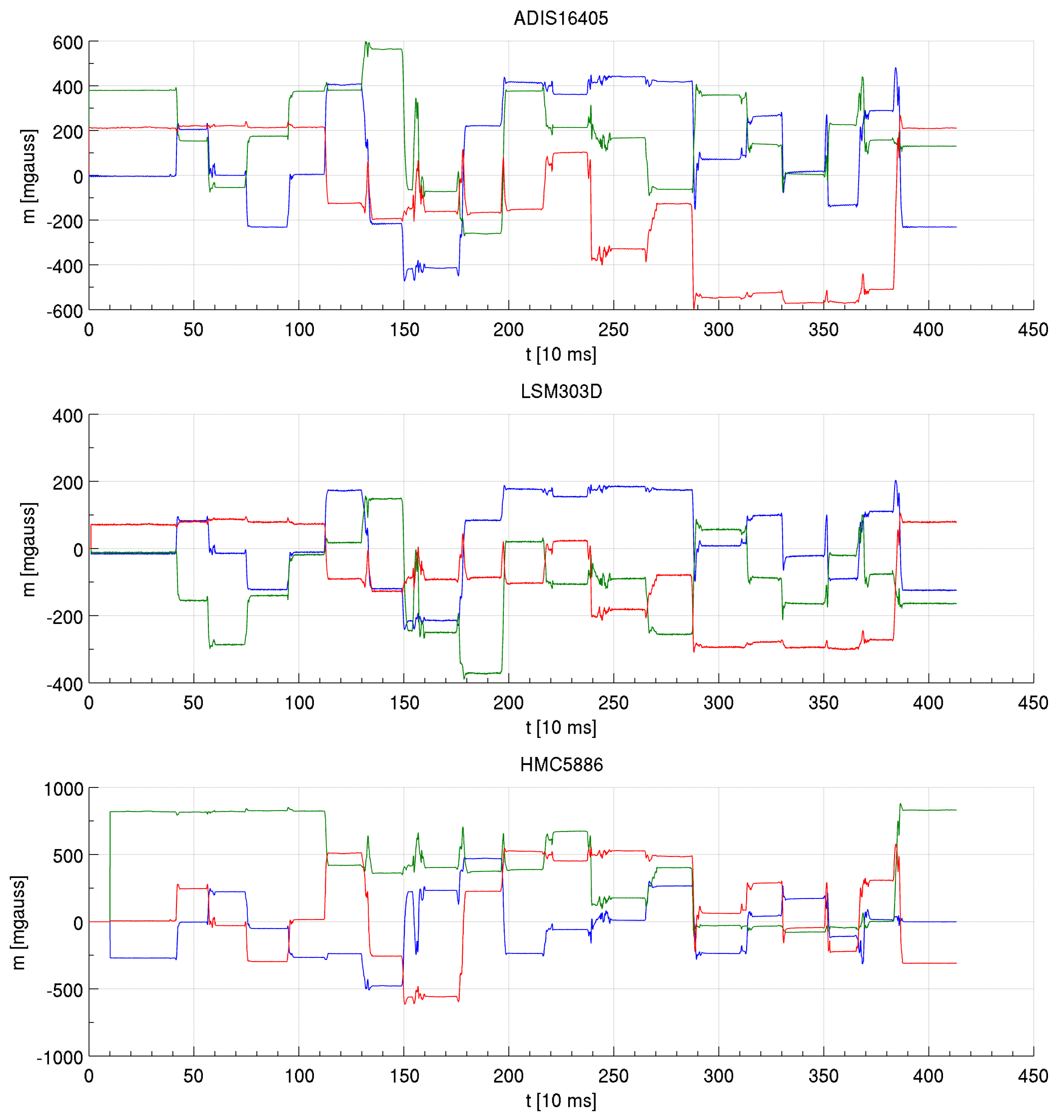
Jelenleg a Linuxbot programmal robotpilóta algoritmusok, szenzorfúzió, szabályozás és guidance algoritmusok tesztelése zajlik. A fejlesztés során a Linuxbot megbízhatóan használható, ez igazolja, hogy az implementált struktúra és a szenzorok adatkezelési mechnizmusa megfelelő egy korszerű UAV fedélzeti szoftver keretrendszerének.



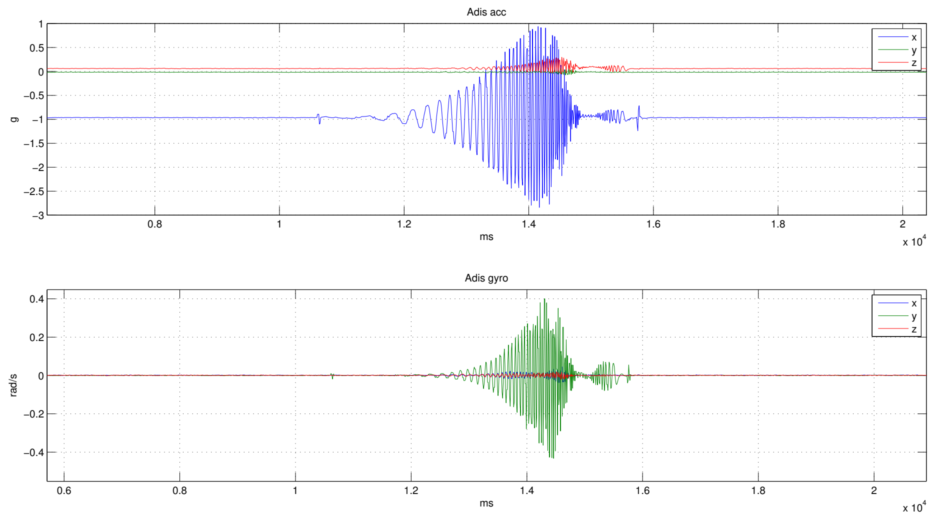
5.1. ábra. *Három különböző gyorsulásmérő adatai*



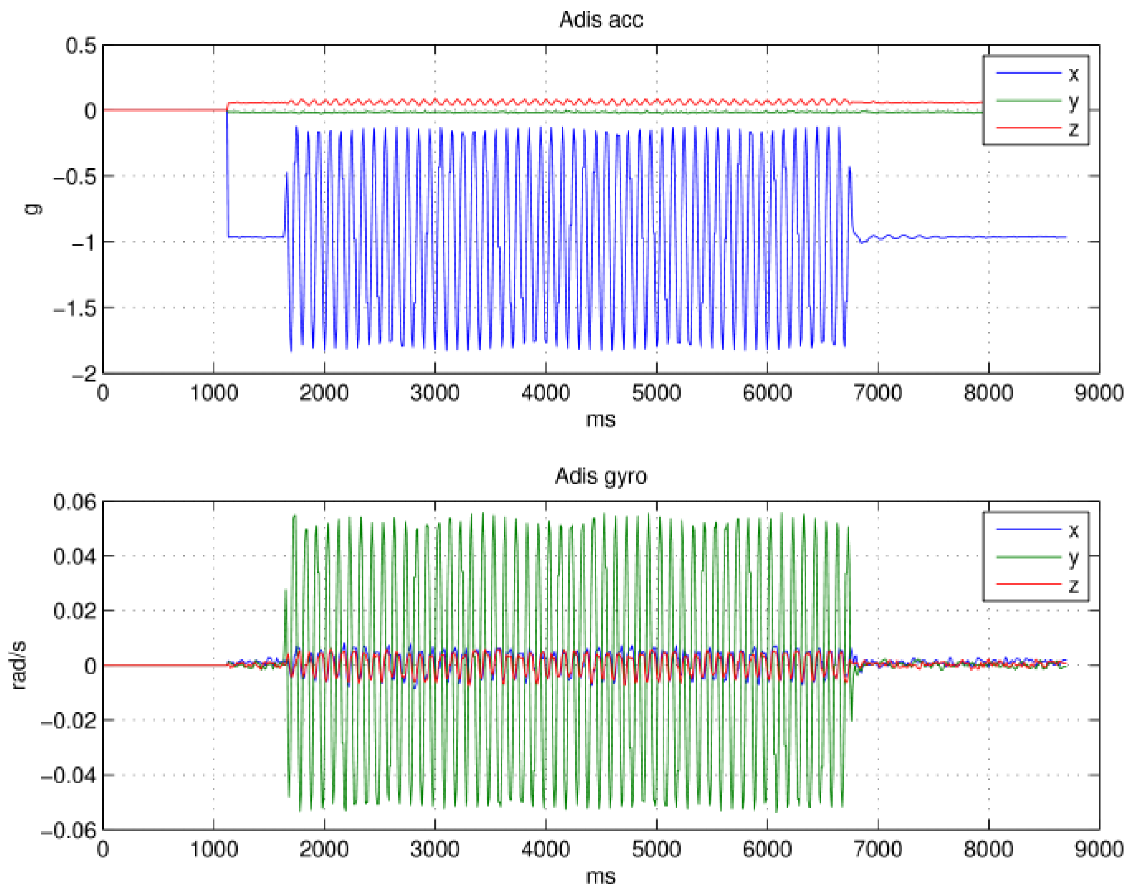
5.2. ábra. *Három különböző giroszkóp adatai*



5.3. ábra. *Három különböző magnetométer adatai*



5.4. ábra. *Sweep lineáris rázás az IMU mechanikai felfüggesztésének átviteli függyényének mérésére*



5.5. ábra. *Gyorsulásmérő és giroszkóp közötti kapcsolat*

Összefoglalás

A dolgozat egy korszerű, civil felhasználású UAV robotpilóta felépítését és fedélzeti szoftverét mutatja be.

Az első fejezetben általánosságban bemutatom milyen részegységekből áll egy UAV, különös hangsúllyal a szenzorokra és azok néhány lehetőségére, hibájára és korlátozására. A második fejezetben az AMORES project keretében megvalósított konkrét hardver elemeket ismertettem.

A harmadik fejezetben az a szenzorok felhasználásának lehetőségeiről volt szó. Röviden összefoglaltam a navigációs szenzorok alkalmazását a repülő orientáció becslésére. Rávilágítottam arra, hogy a szenzorok önmagukban nem alkalmasak az irányításhoz szükséges információk előállítására, ehhez bonyolult algoritmusokra van szükségünk.

Láttuk, hogy vannak méréseink szenzorokból, ezek adatait be kell gyűjteni és algoritmusokat futtatni. A negyedik fejezetben bemutattam egy szoftver keretrendszert, ami implemetálásra került. A fejezet elején bemutattam azokat a nehézségeket, amik egy ilyen real-time rendszer fejlesztés közben felmerülnek a használt Linux operációs rendszer alatt. Ezután bemutattam az egyes funkciókat megvalósító taskok működését.

Végül bemutattam az elkészült rendszerrel elvégzett néhány mérést. Mivel a dolgozatnak nem célja sem a szenzorok részletesebb elemzése, sem az algoritmusok pontos ismertetése, az eredmények tárgyalása pusztán a Linuxbot rendszer helyes működésére szorítkozik.

A rendszer fejlesztése folyamatban van. Habár a főbb funkciók működnek, egy komplett és megbízható, piacképes UAV fedélzeti szoftver rendszerhez még sok fejlesztés szükséges.

Irodalomjegyzék

[1] Lapack. http://www.netlib.org/lapack/.

[2] Linux kernel development second edition. http://www.makelinux.net/books/lkd2.

[3] J. L. Crassidis and F. L. Markley. Unscented filtering for spacecraft attitude estimation. *Journal of Guidance, Control and Dynamics*, 26(4):536–542, 2003.

[4] T. Gausz and B. Gáti. Merevszárnyú és többrotoros légi eszközök modellje precíziós repülési feladatok szimulációjához, 2013. AMORES projekt.

[5] Warren S Flenniken Iv, John H Wall, and David M Bevly. Characterization of Various IMU Error Sources and the Effect on Navigation Performance. 2000.

[6] T. Lajos. *Az áramlástan alapjai*. Műszaki Könyvkiadó, 2008.

[7] Sebastian O H Madgwick. An efficient orientation filter for inertial and inertial / magnetic sensor arrays. 2010.

[8] F. L. Markley and D. Mortari. How to estimate attitude from vector observations. In *AAS/AIAA Astrodynamics Specialist Conference*, Girdwood, Alaska, August 1999.

[9] Midé. Air Pressure at Altitude Calculator. `http://www.mide.com/products/slamstick/air-pressure-altitude-calculator.php`. [Online elérhető: 2015 10. 26].

[10] William Premerlani and Paul Bizard. Direction Cosine Matrix IMU : Theory. (Dcm):1–30, 2009.

[11] H. Schaub and J. L. Junkins. Stereographic orientation parameters for attitude dynamics: A generalization of the Rodrigues parameters. *Journal of Astronautical Sciences*, 44(1):1–19, 1996.

[12] N. Trawny and S. I. Roumeliotis. Indirect Kalman filter for 3D attitude estimation: A tutorial for quaternion algebra. Technical Report 2005-002, Rev. 57, Department of Computer Science & Engineering University of Minnesota, March 2005.

[13] G. Wahba. *SIAM Review*, 8(3):384–386, 1966.