M Ű E G Y E T E M 1 7 8 2

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Szélessávú Hírközlés és Villamosságtan Tanszék

A SMOG-1 PocketQube műhold redundáns fedélzeti számítógépének hardver és szoftver fejlesztése

TDK dolgozat

Készítette

Kristóf Timur

Konzulens

Dudás Levente

2015. október 26.

Tartalomjegyzék

1. Bevezető **3**
1.1. Küldetés 3
1.2. Mire jó ez? 4

2. Kihívások **5**
2.1. Alkatrészek 5
2.2. Energia 5
2.3. Redundancia 6
2.4. Mechanikai terhelés és termikus viszonyok 7

3. SMOG-1 zsebműhold rendszerterve **8**
3.1. Magas szintű rendszerterv 8
3.2. Funkcionális rendszerterv 8
3.3. Részletes rendszerterv 8

4. Fedélzeti számítógép hardvere **11**
4.1. Mikrokontroller 11
4.2. Háttértár: flash memória 11
4.2.1. Mekkora a kellő tárolókapacitás? 11
4.3. Időzítés: RTCC 12
4.4. Szenzor 12
4.5. Áramköri megvalósítás 13
4.5.1. Funkcionális tartalékolás, redundancia 13
4.5.2. Áramköri tartalékolás 15

5. Fedélzeti számítógép szoftvere **16**
5.1. Event loop megvalósítása 16
5.2. Prioritásos sor 18
5.2.1. Versenyhelyzetekről 19
5.2.2. Versenyhelyzet jelentősége mikrokontrolleren 22
5.2.3. Versenyhelyzet megoldása 23
5.3. Energiatakarékosság 24

5.3.1. Eseményvezéreltség jelentősége az energiatakarékosságban 25
5.3.2. Energiatakarékosság a CPU tehermentesítésével: DMA 26
5.3.3. Alacsony fogyasztás elérése soros port használatakor 26
5.4. Optimalizálás 28

6. Fedélzeti számítógép feladatai 29
6.1. Analógia operációs rendszerekkel 29
6.2. Fájlrendszer, tömörítés 29
6.3. Titkosítás 30
6.4. Hibajavító kódolás 30
6.5. Telemetria gyűjtés 30
6.6. Spektrummérés ütemezése 30
6.7. Kapcsolat a többi alrendszerrel 31
6.7.1. PCU (power control unit) 31
6.7.2. COM (rádiókommunikáció és spektrumanalizátor) 31
6.7.3. Áramkorlátozó kapcsolók 31

7. Összefoglaló 33
7.1. Köszönetnyilvánítás 33

Első fejezet

Bevezető

A PocketQube a CubeSat típusú műholdak egy új osztálya, amelyet a Morehead State University és a Kentucky Space szabványosított. Fő jellemzője, hogy mérete nem haladhatja meg az 5×5×5 centimétert, és tömege legfeljebb 180 gramm.

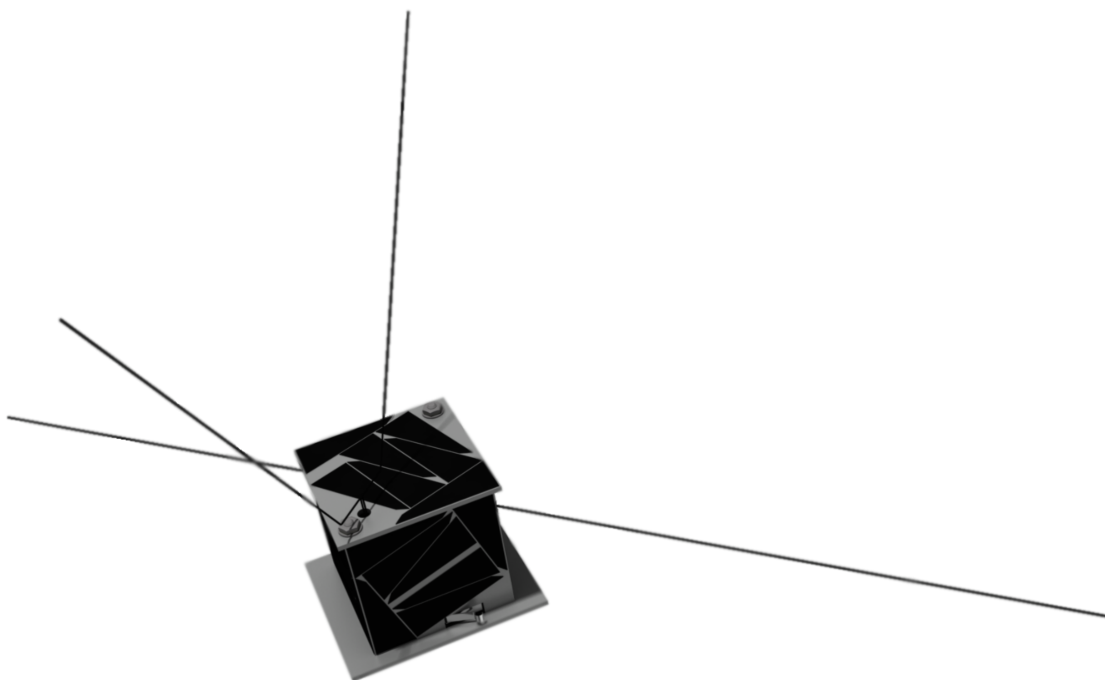
Jelen dolgozat a BSc önálló laboratórium feladatom [6] folytatásaként született. Saját feladatom a SMOG-1 projektben a műhold fedélzeti számítógépének megtervezése, beleértve a hardvert és a rajta futó szoftvert is. Ezen kívül részt vállaltam a műhold rendszertervének kidolgozásában is. Ezek a feladatok remekül igazodnak a korábbi szoftverfejlesztői tapasztalataimhoz és jó lehetőséget adtak nekem arra is, hogy elmélyüljek a komplex hardvertervezésben.

Magyarország első műholja a MASAT-1 volt, amely az eredeti CubeSat szabvány szerint 10×10×10 cm-es volt. Ennek egyik utódprojektje a SMOG-1, amely már az újabb, kisebb méretű szabvány szerint készül a Budapesti Műszaki és Gazdaságtudományi Egyetemen a Villamosmérnöki és Informatikai Kar és a Gépészmérnöki kar együttműködésével. [1] [7] A műhold elektronikáját a Szélessávú Hírközlés és Villamosságtan Tanszék hallgatói tervezik meg és rakják össze. [2] [4] [3] [5] [6] A projekt 2014-ben kezdődött, és jelenlegi terveink szerint 2015 decemberében várható a mérnöki példány elkészülte. Ez korántsem jelenti azonban azt, hogy ezzel a műholdat befejeztük; ezt a példányt alapos tesztelésnek fogjuk alávetni és a tesztek eredménye szerint fogjuk továbbfejleszteni, esetleg egyes kritikus részeket újratervezni, ha úgy ítéljük szükségesnek. A mérnöki példányt egy kvalifikációs és egy repülő (végleges) példány fogja követni.

1.1. Küldetés

Műholdunk fő küldetése az elektromágneses spektrum monitorozása a Föld körüli pályáról DVB-T sávban. Vagyis azt mérjük, hogy a földi DVB-T műsorszóró adók jele mennyire vehető a világűrből. [1] (A SMOG-1-et kb. 500-600 km-es magasságú pályára tervezzük.)

A SMOG-1 a magyarországi CubeSat műholdak oktatási vonalát képviseli. Ez azt jelenti, hogy nagy részét a Műegyetem hallgatói tervezik, és a költségvetése is az egyetemi kasszához mért, vagyis drága, űrminősített eszközök helyett a piacon széles körben elérhető ipari alkatrészekből építkezünk.



1.1. ábra. SMOG-1 3-dimenziós terve

1.2. Mire jó ez?

Természetesen adódik a kérdés, hogy miért érdekes bárki számára is az az információ, hogy a földi digitális műsorszórás vajon vehető-e az űrből. Hiszen odafönt senki sem szeretne tévét nézni. Pontosan ez a válasz a kérdésre. Ugyanis a műsorszóró adók célpontja a földi lakosság, ezért minden olyan elektromágneses jel, ami vehető ezektől az adóktól az űrben, valójában elpazarolt, kidobott teljesítmény.

Küldetésünk tehát alkalmas lesz arra, hogy a méréseinkből arra lehessen következtetni, pontosan melyik földi adó mennyi felesleges teljesítményt sugároz fölfelé. Ezen információ birtokában pedig az adott adó üzemeltetői továbbfejleszthetik, vagy hangolhatják az antennáik iránykarakterisztikáját olyan módon, hogy a földi lakosság felé jutó teljesítmény a lehető legnagyobb, az űrbe kisugárzott elpazarolt teljesítmény pedig a lehető legkisebb legyen.

A küldetés legfontosabb eleme egy folytonosan hangolható spektrumanalizátor, amely a műhold kommunikációs rendszerének (COM) részét képezi. Ezt a spektrumanalizátort és a felbocsátásához használt magaslégköri ballont Dudás Levente és a Mikrohullámú Távérzékelés Laboratórium munkatársai készítették és már számos légballonos kísérleten bizonyított. [1]

A spektrumanalizátoron kívül néhány egyéb kísérlet fog még helyet kapni a fedélzeten, többek között egy totáldózismérő RAD FET, amely Géczy Gábor munkája [3] valamint egy különleges hőszigetelő anyag tesztje, ami az akkumulátort fogja védeni.

Második fejezet

Kihívások

2.1. Alkatrészek

Általában — mint ahogyan a CubeSat műholdak esetén is — a PocketQube-ok tervezésénél a mérnökök a piacon széles körben elérhető ipari alkatrészekből építkeznek. Ennek fő oka az, hogy ezeket a műholdakat zömében okatási céllal, vagy rádióamatőrök készítik, emiatt pedig az űrminősített alkatrészek használata pénzügyileg megfizethetetlen terheket róna a tervezőkre.

Természetesen nem mindegy, hogy milyen alkatrészeket használunk, hiszen egy 5 cm-es kocka meglehetősen korlátozott méretű. Így az egyik legfontosabb szempont, hogy mindenből a lehető legkisebbet válasszuk. A SMOG-1 csapat jelentős időt fordított arra, hogy különféle katalógusokat böngészett, kutatva azt, hogy egy adott feladatra mik a lehető legkisebb elérhető áramkörök, illetve, hogy az adott alrendszert hogyan lehet a lehető legkisebb méretűre zsugorítani. (Ennek kiemelkedő példája az RTCC, a szenzorika és az EPS-ben használt áramkorlátozó kapcsolók.)

Másik fő szempont — amely sajnos kissé ellentmond az előzőnek — az, hogy olyan alkatrészeket válasszunk, amik „már jártak az űrben”, vagyis volt már olyan csapat, akik sikeresen használták az adott alkatrészt a saját amatőr műholdjukban. Sajnos, mivel a CubeSat műholdak még mindig nem terjedtek el széleskörűen, nehéz ilyen alkatrészeket találni, főleg a méretkorlátok miatt, hiszen a SMOG-1 a korábbi CubeSat-ok térfogatának mindössze egy nyolcadával rendelkezik. A fentiek miatt többnyire megelégszünk azzal is, ha olyan alkatrészt találunk, ami valamilyen ipari minősítéssel rendelkezik (pl. a szokásosnál szélesebb hőmérséklettartományban ígér jó működést a gyártó).

2.2. Energia

Elsődleges energiaellátásunkat napelemek biztosítják, amelyek a kocka 5 cm-es oldallapjain helyezkednek el. Ez azt jelenti, hogy a napelemek számára rendelkezésre álló felület kb. egynegyede a MASAT-1 felületének, vagyis negyedakkora bejövő teljesítményre számíthatunk, amely előzetes becslésünk szerint kb. 0,3 watt (körátlag). Ekkora teljesítménnyel kell gazdálkodnunk, és a fedélzeti alrendszereknek is ebből kell működniük. A napelem oldalak működtetése és a bejövő

teljesítmény optimalizálása Herman Tibor munkája. [4] [5]

Természetesen a SMOG-1-ben is lesz egy akkumulátor, azonban a hely szűke miatt ez is lényegesen kisebb lesz, mint a MASAT-1 akkumulátora volt. Optimális esetben az akkumulátort a napelemek töltik, amikor a műhold a Föld körüli pályája napos részén tartózkodik, és amikor földárnyékban van, akkor pedig teljes egészében az akkumulátorról működnek majd a fedélzeti rendszerek.

Az akkumulátor és napelemek által szolgáltatott elektromos teljesítményt az energiaelosztó rendszer szabályozza. Ez a rendszer stabil feszültséget állít elő jó hatásfokkal a fedélzeti számítógép és annak perifériáinak számára. Az energiaelosztó rendszer Géczy Gábor munkája. [2] [3]

Akkumulátorunk kiválasztásánál ügyelnünk kell rá, hogy bizonyos fajták nem működnek jól az űrben. (pl. a lítium-polimer akkumulátorok néhány kivételtől eltekintve alkalmatlanok a feladatra, mert fagypont alatt túlzott mértékben megnő a belső ellenállásuk.) Nem feltételezhetjük azt sem, hogy az akkumulátor a SMOG-1 küldetésének teljes időtartama alatt hibátlan lesz, mint ahogy azt sem, hogy már a pályára állításkor egyáltalán működni fog. (ld. később.) Emiatt a műholdat úgy tervezzük meg, hogy olyan esetekben is működjön (legalább a pályája napos oldalán), amikor az akkumulátor tönkrement.

2.3. Redundancia

Hogy ellenálljon a világűr viszontagságainak, a SMOG-1 alrendszereit redundáns módon kell megtervezni, mégpedig úgy, hogy bármely pont meghibásodása esetén a teljes rendszer zökkenőmentesen működhessen tovább.

Lehetséges meghibásodások az alábbiak:

- Ellenállás: szakadássá változhat, ezért kritikus helyekre legalább két párhuzamos ellenállást helyezünk el.
- Kondenzátor: rövidzárrá vagy szakadássá is változhat, ezért kritikus helyre négy kondenzátort kell elhelyezni. Ha ezek közül bármelyik bárhogyan elromlik, a kapcsolás még mindig kondenzátorként működik, csak a kapacitása változhat meg.
- Diódák, tranzisztorok: rövidzárrá vagy szakadássá változhatnak
- Integrált áramkörök és egyéb aktív eszközök: bármelyik alkatrészláb rövidre záródhat akár a föld (logikai 0) vagy a tápfeszültség (logikai 1) felé, vagy tirisztorhatás jöhet létre. Ezt általában funkcionális redundanciával küszöböljük ki, vagyis az adott funkcionális egységből, amelynek része az adott áramkör, legalább kettőt helyezünk el a fedélzeten.

Ez azt jelenti, hogy a tervezés során különös gondot kell fordítani arra, hogy bármelyik alkatrész, ha elromlik, azzal ne veszélyeztesse a többi alrendszert. (Pl. Ha egy integrált áramkör valamelyik kivezetése rövidre záródik a földponttal, az ne okozhassa más áramkörök tápellátásának megszűnését, és ne akadályozza azok között a kommunikációt.)

A redundanciáról bővebben ld. később.

2.4. Mechanikai terhelés és termikus viszonyok

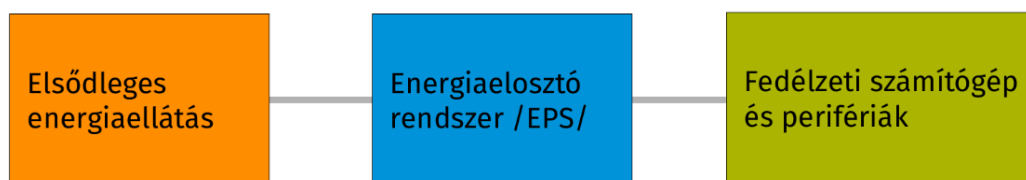
A műholdat rázópad és termovákuum kamra segítségével fogjuk tesztelni, hogy ellenálljon az űrben uralkodó ellenséges viszonyoknak. Gépészmérnök kollégáink foglalkoznak ezekkel a kérdésekkel és végeztek termikus számításokat a 2014-es TDK konferencia keretében. [7]

Harmadik fejezet

SMOG-1 zsebműhold rendszerterve

3.1. Magas szintű rendszerterv

Műholdunk alapvetően három jól elkülöníthető egységből épül fel, amelyeket a 3.1 ábrán láthatunk. [6] Az elsődleges energiaellátásért a kocka oldalain levő napelemek felelősek, melyek MPPT áramkörrel ellátva táplálják a SMOG-1-et. [4] Ezt az energiát kezeli, raktározza és átalakítja az energiaelosztó rendszer (EPS). [2] Ez állítja elő azt a feszültséget, amiről a fedélzeti számítógép és annak perifériái működnek.



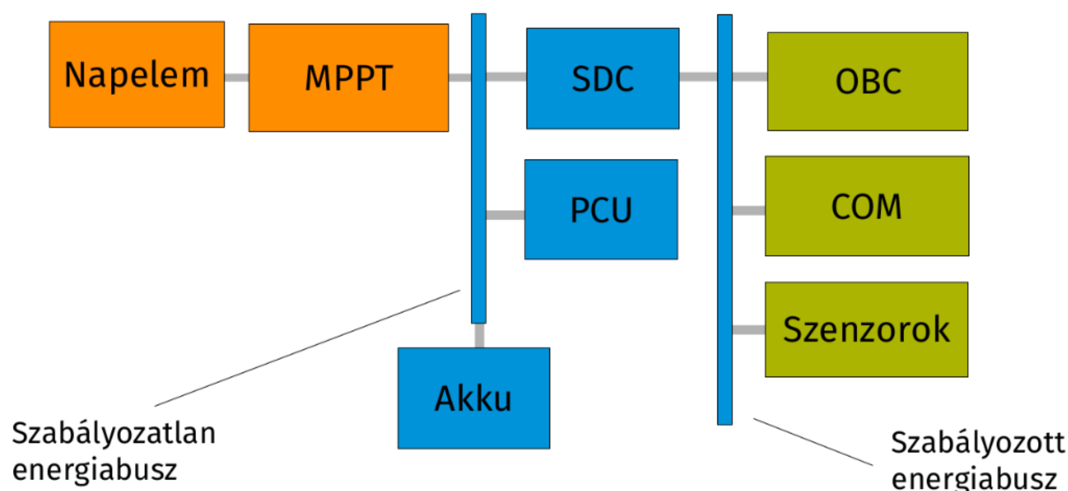
3.1. ábra. Magas szintű rendszerterv

3.2. Funkcionális rendszerterv

Az energiaelosztó rendszer tartalmaz egy akkumulátort az energia tárolására, egy PCU (power control unit) egységet a programozott logikák megvalósítására és egy step-down konvertert, amely a fedélzeti számítógép és perifériái részére állít elő megfelelő feszültséget. Ezen áramkörök mindegyike megfelelő védelemmel (túláram és túlfeszültség, redundancia, stb.) vannak ellátva. a 3.2 ábrán (a redundáns párjuk nélkül) látható a műhold blokkvázlata a lényeges funkcionális egységekkel.

3.3. Részletes rendszerterv

A napelemek és az akkumulátor egy ún. szabályozatlan energiabuszra (ahol akár 5V is lehet) vannak rákötve. Erről üzemel a PCU és az SDC, amely egy ún. szabályozott energiabuszt hajt



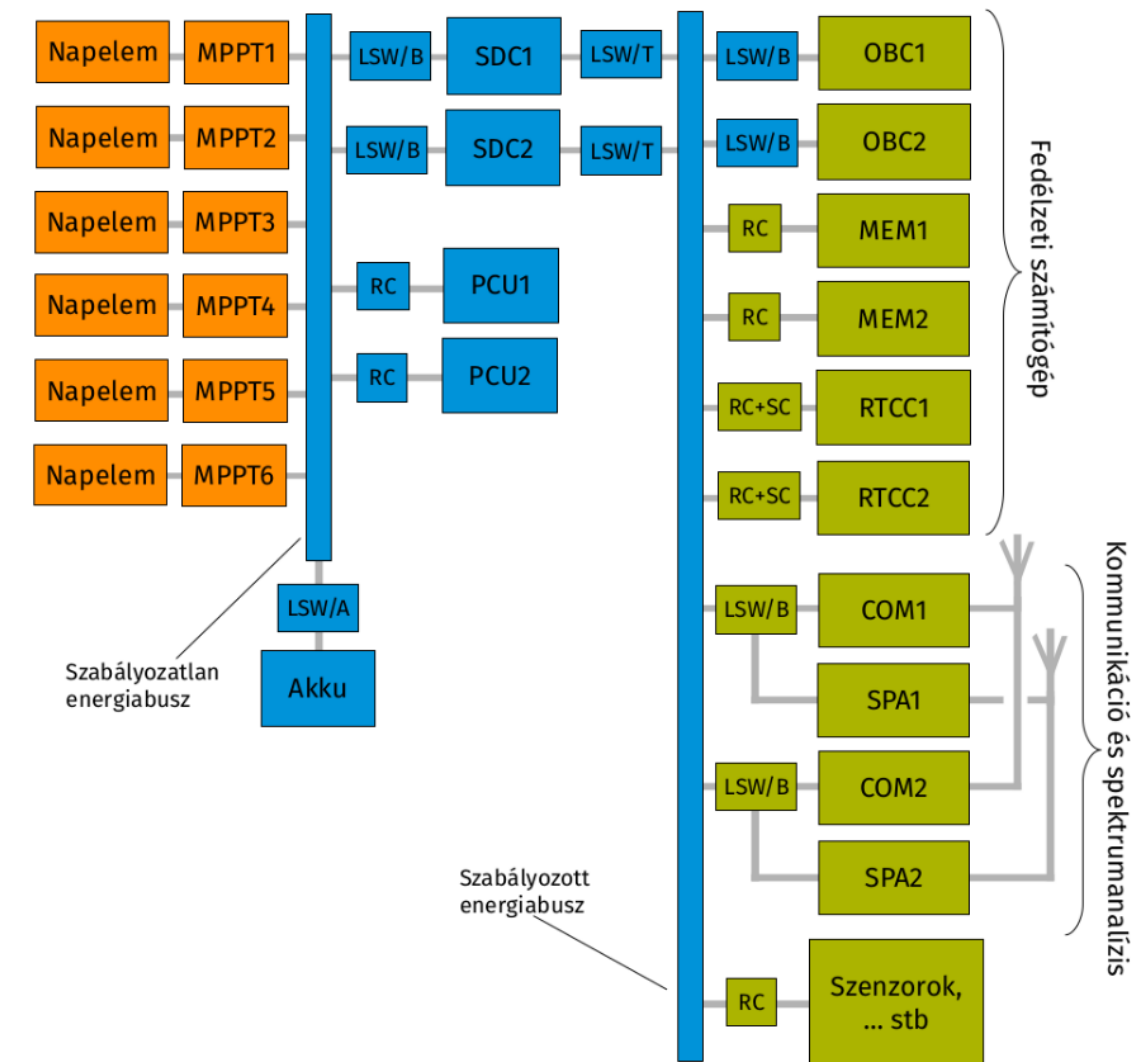
3.2. ábra. Funkcionális rendszerterv

meg a kimenetével (1,8V – 3,6V). Ezen a buszon olyan feszültség van, amiről már az OBC és perifériái működni tudnak.

Természetesen a 3.2 ábra még tovább finomítható, ha az egységeket további részekre bontjuk és figyelembe vesszük a redundáns párokat és a védelmi kapcsolásokat is. Többféle van: az LSW/A az akkumulátor túl- és alulfeszültség és -áram védelmére szolgál, az LSW/B egy áramkorlátozó kapcsoló, az LSW/T pedig egy túlfeszültségtől védő kapcsoló. [2] Az alacsony fogyasztású alkatrészeket áramkorlátozó kapcsoló helyett egyszerű RC tagokkal kötjük a szabályozott energiabuszra, az RTCC (real time clock and calendar, vagyis gyakorlatilag egy pontos kvarcóra) áramkör számára pedig szuperkapacitás biztosítja a redundáns energiaellátást. (Ezt a részletes rendszerterven az „RC+SC" rövidítés jelöli.)

Az OBC-t szétbonthatjuk háromfelé: egy mikrokontrollerre, a hozzá tartozó flash memóriára és az RTCC-re. (ld. a fedélzeti számítógép hardvere c. fejeztetet.) A COM rendszert még szétbonthatjuk a földi állomással kommunikáló egységre és a spektrumanalizátorra. A részletes rendszertervet szemlélteti a 3.3 ábra.

Megemlítendő még, hogy a két spektrumanalizátor és a két rádiókommunikációs rendszer egy-egy közös antennával rendelkezik. Ezen antennákat Pápay Levente készíti, és egymásra merőlegesen helyezkednek majd el a műhold külsején. Az antennanyitó mechanizmus is redundáns és annak működését is a fedélzeti számítógép vezérli. (Ennek a mechanizmusnak a működése jelen dolgozat írásakor még nem kellően kidolgozott, ezért nem szerepel a rendszertervről szóló ábrán.)



3.3. ábra. Részletes rendszerterv

Negyedik fejezet

Fedélzeti számítógép hardvere

4.1. Mikrokontroller

A fedélzei számítógép „lelke” egy EFM32 családba tartozó „Wonder Gecko” típusú EFM32WG mikrokontroller, melyet a Silicon Laboratories (Silabs) gyárt. Ez tartalmaz egy saját kategóriájában jónak mondható ARM Cortex-M4 processzort (CPU) és megfelelő mennyiségű operatív memóriát (RAM).

A tervezés korai fázisában egy PIC24 családba tartozó mikrokontrollert választottam a feladathoz, azonban világossá vált, hogy annak számítási kapacitása alkalmatlan lenne a kódolási és titkosítási feladatok megvalósítására.

A mikrokontroller feladata, hogy az OBC fő szoftverét futtassa. Ez a szoftver fogja vezérelni az összes többi alrendszert (az energiaellátó rendszer kivételével), legfontosabbként említendő a kommunikációt, valamint gyűjt majd telemetriaadatokat az egész műhold összes alrendszere felől (beleértve az EPS-t és a napelem oldalakat is). Lásd bővebben a szoftver felépítéséről szóló fejezetet.

4.2. Háttértár: flash memória

Küldetéséből kifolyólag a SMOG-1 mérőrendszere folyamatosan adatokat generál. Egyrészt a spektrumanalizátor kimenetét (ld. bevezető / küldetés), másrészt a műhold különféle alrendszereinek telemetriaadatait (ld. összeköttetések).

Fontos tehát, hogy legyen egy megbízható háttértár, ami kellő kapacitással rendelkezik. Szóba jönnek az EEPROM modulok és a flash memóriák. Lényeges szempont, hogy a memória ne fogyasszon sokat és lehetőleg a mikrokontrollerrel azonos (vagy tágabb) feszültségtartományban üzemeljen.

4.2.1. Mekkora a kellő tárolókapacitás?

Először számoljuk ki, hogy a spektrumanalízátor méréseiből mekkora adatmennyiség keletkezik egy nap alatt. Becsléseink szerint egy mérés $M = 2000 byte$ adatmennyiséget jelent és $T = 20 sec$

ideig tart. [1] Az alábbi egyszerű számítással megkaphatjuk, mekkora sebességgel keletkeznek új adatok:

$$\frac{1nap}{T}M = \frac{24 \cdot 3600}{T}M\frac{byte}{nap} = 8640000\frac{byte}{nap} = 8.24\frac{Mbyte}{nap}$$

Tehát egy nap alatt kicsit több, mint 8 Mbyte adat keletkezik (ha nem számítjuk a telemetriaadatokat). Vegyük figyelembe azt, hogy a SMOG-1 egy nap alatt körülbelül 4-6- szor halad el Magyarország felett, tehát ennyi lehetőségünk van rá, hogy az adatokat letöltsük. Arra nincs szükség, hogy hosszú ideig tároljuk őket a műholdon, de a biztonság kedvéért a szükségesnél kicsit nagyobb tárhelyet helyeztem el a fedélzeten. (Kisebb memóriájú chip fizikailag nem kisebb és nem is fogyaszt kevesebbet.) Így esett a választás az Adesto által gyártott AT45DB641E modulra, amely 64 Mbit (8 Mbyte) tárterülettel rendelkezik és képes az 1,7 - 3,6 V-os tartományban működni. [30] További modulok, amiket találtunk, csak lényegesen nagyobb feszültségen (többnyire 3 V körül) kezdtek működni, vagy lényegesen szűkebb tartományban (pl. 1,65 - 1,95 V) működtek.

4.3. Időzítés: RTCC

A műholdon pozíció meghatározására való szenzor sajnos nem kaphat helyet, mert a kereskedelmi forgalomban kapható GPS áramköröket bizonsági okokból úgy készítik, hogy korlátozzák a maximális magasságot, sebességet és gyorsulást, amelyen működnek. Vagyis bizonyos magasság (altitude) felett „levágják" a mért értékeket és nem szolgálnak pontos adatokkal.

Ezen probléma azért jelentős, mert hiába mérünk a spektrumanalizátorral, ha nem tudjuk semmilyen módon megállapítani, hogy a mérést hol végezzük.

Erre ad megoldást az a megközelítés, hogy ha tudjuk az egyes mérések pontos idejét és a Földről követjük a műholdat, akkor a pályájának kiszámolásával és a pontos idő figyelembevételével mégis meg tudjuk majd mondani, hogy melyik mérést hol végezte a műhold. Ezt a célt szolgálja az RTCC áramkör, amely SPI protokollon kommunikál a fedélzeti számítógép mikrokontrollerével, valamint bizonyos időközönként egy interrupt segítségével felébreszti azt.

Felmerülhet a kérdés, hogy miért nem használtuk valamely mikrokontrollerbe épített RTCC megoldást. [31] Ennek oka az, hogy ezek a megoldások korántsem elég precízek, használatuk esetén a kontrollert nem lehet alvó módban üzemeltetni, valamint redundancia okokból ilyenkor a fedélzeti számítógép mindkét redundáns egységének egyszerre kellene működnie, hogy az egyikük meghibásodása esetén a másik se veszítse el a pontos időt. Ehelyett inkább egy rendkívül alacsony fogyasztású áramkört használunk, amelyet Géczy Gábor épített és amelynek redundáns energiaellátást is bizosítunk 11 mF-os szuperkapacitások segítségével. [2]

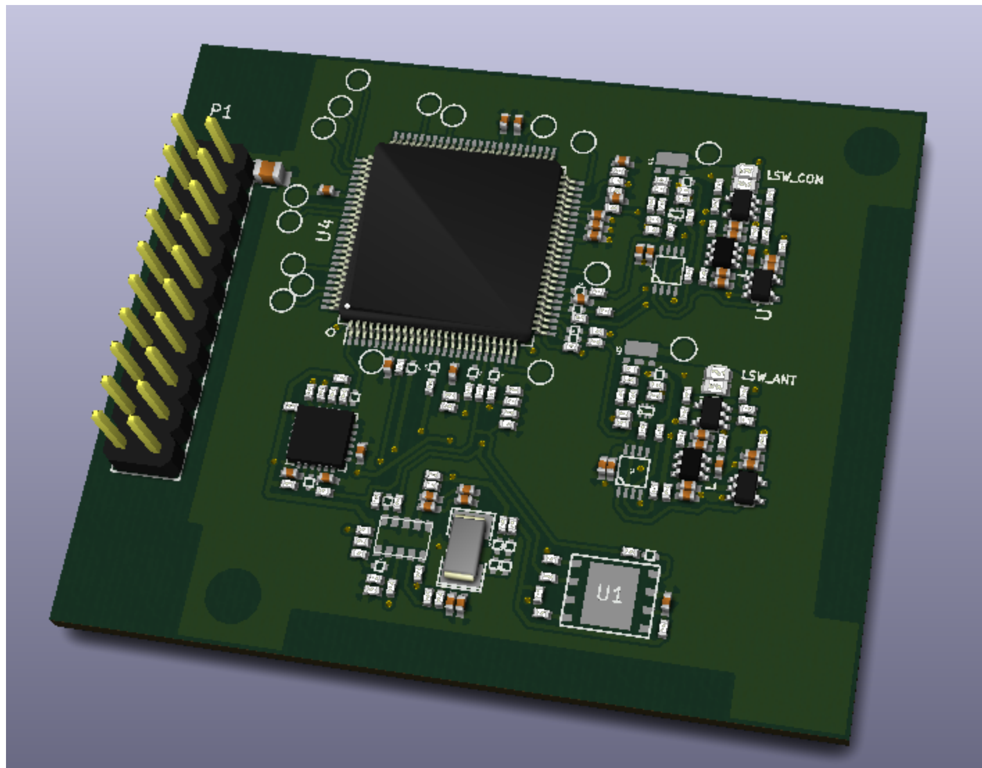
4.4. Szenzor

Helyet kap a fedélzeti számítógép részei között egy mozgásérzékelő szenzor is, amely képes három tengely mentén gyorsulást, szögsebességet és mágneses térerősséget mérni.

Választásunk az Invensense MPU-9250 [32] termékére esett. Főbb szempontjaink a szenzor minél kisebb mérete és ehhez képest minél nagyobb tudása voltak. Ezt a szenzort használjuk többek között arra, hogy adatokat gyűjtsünk arról, mikor milyen mozgásállapotban van a műhold. Ez az információ jelterjedési szempontokból is érdekes. (Pl. segítségünkre lehet az ún. „spin fade” jelenség vizsgálatában.)

4.5. Áramköri megvalósítás

A teljes fedélzeti számítógép az összes komponensével (mikrokontroller, flash memória, RTCC, mozgásérzékelő szenzor) és az általa vezérelt áramkorlátozó kapcsolókkal együtt (amelyek a kommunikációs alrendszer és az antennanyitó mechanizmus energiaellátásáért felelnek) egy kétoldalas, négy rétegű nyomtatott huzalozású lemezen foglalnak majd helyet. Ennek az első prototípusát mutatja a 4.1 ábra.

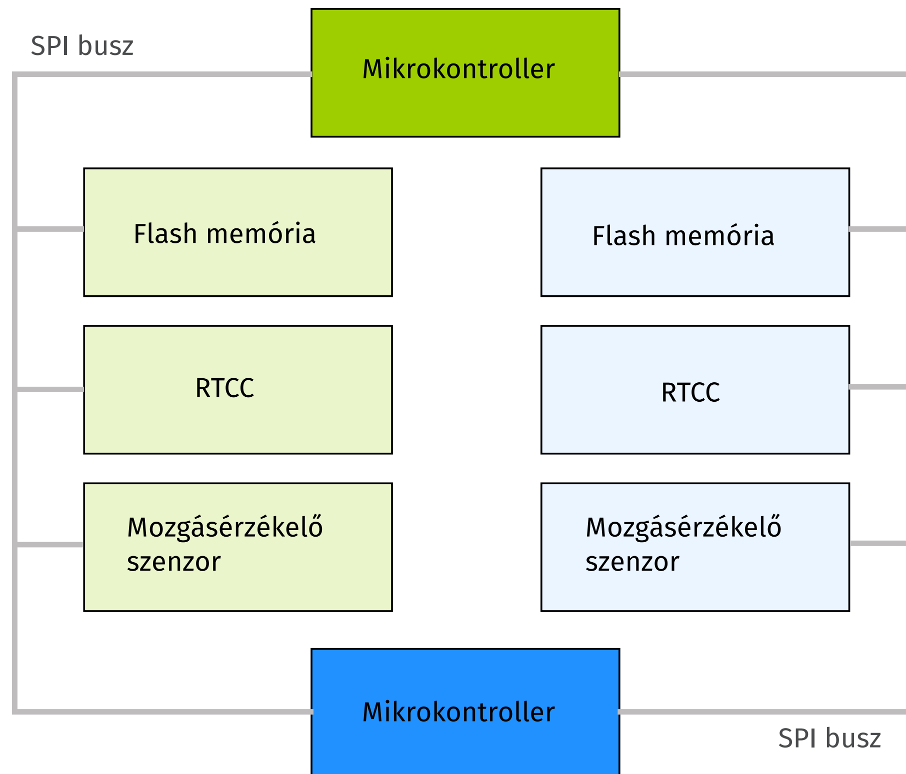


4.1. ábra. OBC első prototípusának 3D képe

A fedélzeti számítógép teljes kapcsolási rajzának pontos ismertetése túlmutat jelen dolgozat keretein. Felépítésének blokkvázlata megtekinthető a 4.2 ábrán.

4.5.1. Funkcionális tartalékolás, redundancia

A fedélzeti számítógép mikrokontrolleréből két redundáns példány is helyet kap az áramkörben, amelyek egymás *funkcionális tartalékai.* Közöttük *hideg redundancia* (cold redundancy) valósul meg, ami azt jelenti, hogy egyszerre csak egyikük lesz bekapcsolva. Annak eldöntése, hogy



4.2. ábra. OBC blokkvázlata

melyiket kell bekapcsolni, a PCU (power control unit) feladata, amely az EPS (energiaelosztó rendszer) része.

Úgy valósul meg a funkcionális tartalékolás, hogy az OBC ún. *heartbeat* („szívdobbanás") jeket küld periodikusan a PCU-k felé. Ha ez a jel megszűnik, vagyis a PCU nem látja a heartbeat jelet bizonyos ideig, akkor újraindítja (reseteli) az aktuálisan bekapcsolt OBC példányt úgy, hogy az energiaellátását lekapcsolja, majd rövid idő múlva visszakapcsolja. Ha a heartbeat jel ezután sem áll helyre, akkor ezt a példányt a PCU elkönyveli „halottnak" és helyette a másikat indítja el.

A mikrokontrollerekhez fizikailag is legközelebb eső elemek (ezek vele azonos nyomtatott huzalozású lemezen helyezkednek el) is természetesen funkcionálisan tartalékoltak. Ez azt jelenti, hogy mindegyikből két példány fog elhelyezkedni az áramkörben, amelyek két SPI buszon fognak a mikrokontrollerrel kommunikálni. Ez úgy értendő, hogy egy mikrokontroller két különböző SPI buszon is master módban működik, és a redundáns párja is rajta lesz mindkét SPI buszon szintén master módban. Fontos megemlíteni, hogy habár mindkét OBC nem lesz egyszerre bekapcsolva (kettejük között ún. hideg redundancia valósul meg), a busz el lesz látva védődiódákkal és ellenállásokkal, hogy a kettejük egyidejű bekapcsolt állapota biztosan ne okozza a slave eszközök meghibásodását.

A fentiekben memória, RTCC és szenzorok egyik példánya az egyik, másik példánya a másik SPI buszon foglal majd helyet slave módban. Így, ha akár mindegyik fajta egység közül egy megsérül, a teljes rendszer még mindig üzemképes marad. Az RTCC egyúttal megszakítást is

tud küldeni az OBC felé, hogy így segítse az egyes feladatok időzítését. [2] Erről bővebben ld. a fedélzeti számítógép feladatairól szóló fejezetet.

4.5.2. Áramköri tartalékolás

Fontos figyelembe venni, hogy nem csak az integrált áramkörök, hanem a passzív áramköri elemek is tönkremehetnek. Ezen hibalehetőségek elkerülése érdekében minden ilyen alkatrész áramköri tartalékolással rendelkezik. (Ezt részletesebben ld. a kihívások c. fejezet redundanciáról szóló részében.)

Ötödik fejezet

Fedélzeti számítógép szoftvere

A fedélzeti számítógépet működtető szoftver az „event loop” mintát követi, ami az eseményvezérelt programozás egyik alappillére. [16] Ennek lényege a következő: az OBC (on-board computer, fedélzeti számítógép) egy prioritásos sort (queue-t) tart fenn az elvégzendő feladatok számára. Valamely esemény bekövetkeztekor mindössze csak rögzíti, hogy újabb dolog vár feldolgozásra, aztán folytatja az addigi teendőit. Amikor pedig éppen kiürült a prioritásos sor, akkor alvó üzemmódba helyezi magát a következő esemény bekövetkeztéig.

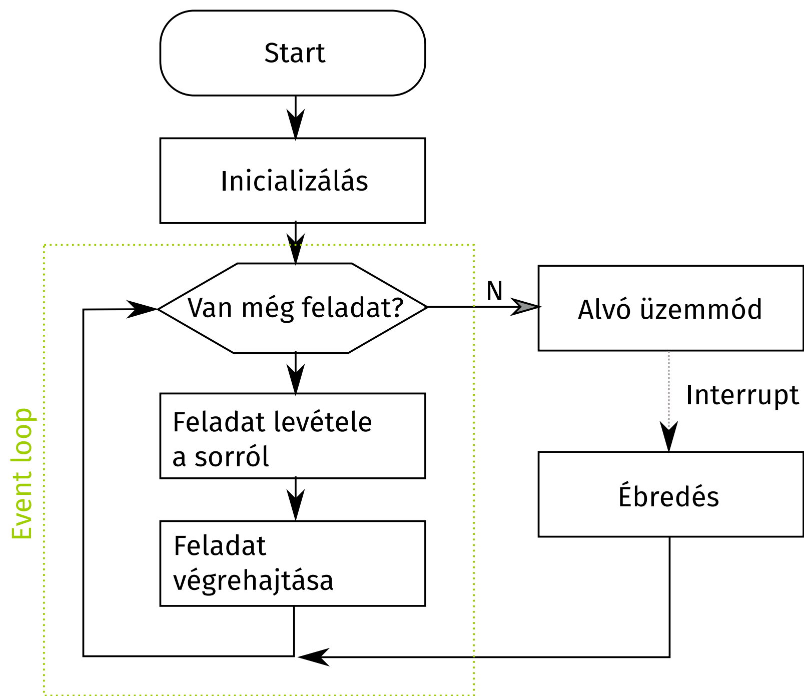
5.1. Event loop megvalósítása

A gyakorlatban az event loop elv azt jelenti, hogy a processzor folyamatosan egy végtelen ciklust futtat, ami ellenőrzi, hogy van-e elvégzendő feladat. Ha nincs, akkor a processzor átvált alvó üzemmódba és így várakozik egészen addig, amíg valamilyen számára érdekes esemény be nem következik. Ez úgy valósul meg, hogy valamelyik periféria megszakítás kérést (interrupt request) küld a processzornak [17], amitől az felébred. Perifériák alatt most nem csak a mikrokontrollerbe integráltakat értem, hanem az összes további egységet, amely kapcsolatban áll még a fedélzeti számítógéppel.

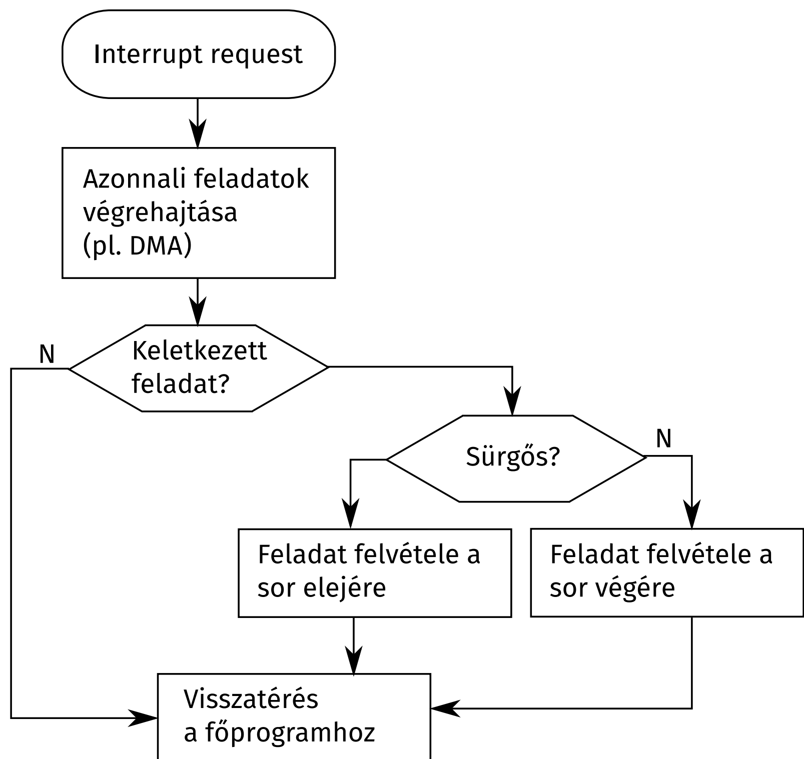
A főprogram egy prioritásos sorban (priority queue) tartja nyilván az elvégzendő feladatokat. Amikor bekövetkezik egy esemény, a processzor, ha aludt, felébred és lefut az adott eseményhez tartozó megszakításkezelő rutin (interrupt service routine, ISR), és jelzi a főprogram számára az adott esemény bekövetkeztét, oly módon, hogy felvesz egy feladatot a már említett prioritásos sorba. A főprogram algoritmusa tehát az 5.1 ábra szerint működik, egy megszakításkezelő rutin pedig az 5.2 ábra szerint.

Előfordulhat, hogy az ISR-nek „azonnali” feladatot kell végrehajtania. Ezalatt olyan tennivalókat értek, amiknek muszáj az interrupt bekövetkeztekor rögtön megtörténniük és nem érnek rá addig, amikor a végrehajtás visszatér főprogramba. Ilyen például a DMA folyamatok vezérlése (ld. később), amelyek elmulasztása hibás működést von maga után.

Az event loop tehát a fentiek alapján egy egyszerű prioritásos soron alapul, amibe kétféle prioritású feladatot lehet felvenni: „normál” esetben a feladat a sor végére kerül, míg a „sürgős”



5.1. ábra. Event loop vázlata



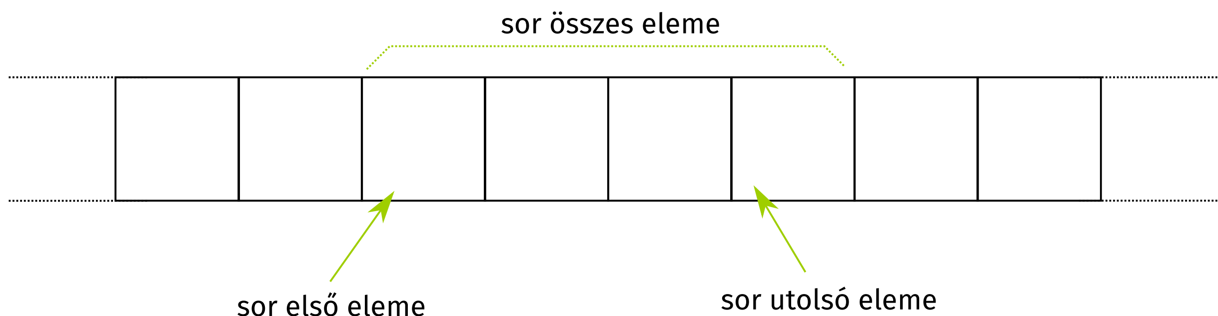
5.2. ábra. Megszakításkezelő rutin vázlata

feladatok a sor elejére. Az event loop egy-egy feladat befejezte után minden esetben a sor elejéről veszi le az éppen következő feladatot és azt hajtja végre. Ha viszont nincs több elvégzendő feladat, akkor a program alvó üzemmódba állítja a processzort. Ez egy olyan működés, amikor a központi feldolgozóegység le van kapcsolva, viszont a mikrokontrollerbe épített többi periféria még működik. (Lásd bővebben az energiatakarékosságról szóló fejezetet.) Ezzel a módszerrel elérhető például többek között az is, hogy az UART periféria, amikor külső egységtől üzenetet kap, felébressze a processzort, ami elkezdheti feldolgozni a beérkező üzenetet. (Erre részletesen kitérek a fedélzeti számítógép többi egységgel való kapcsolódásáról szóló fejezetben.)

5.2. Prioritásos sor

A korábban említett prioritásos sor egy klasszikus megvalósítása lehet az alábbi algoritmus. Ez az adatszerkezet — mivel csak kétféle prioritást ismer — programozástechnikai szempontból megfelel egy kettős sornak (double-ended queue, vagy röviden deque).

Egy lehetséges implementáció a következő. Inicializáláskor foglalunk egy folytonos memóriaterületet, amelyben levő részek a sor elemeit reprezentálják. Egy mutató jelzi a sor első elemét ezen memóriaterületen belül, egy másik pedig a sor végét. Ezt az elrendezést figyelhetjük meg az 5.3 ábrán.



5.3. ábra. Deque implementáció

Amikor új elemet adunk hozzá a sor végéhez, akkor eggyel növeljük az utolsó elemre mutató pointert és az általa mutatott memóriaterületre elhelyezzük az új elemet. Amikor új elemet adunk hozzá a sor elejéhez, akkor eggyel csökkentjük az első elemre mutató pointert és az általa mutatott memóriaterületre elhelyezzük az új elemet. Természetesen ha valamelyik pointer már „kilógna" az eredetileg foglalt folytonos memóriaterületből, akkor új, nagyobb területet kell foglalni és a korábbi elemeket oda átmásolni.

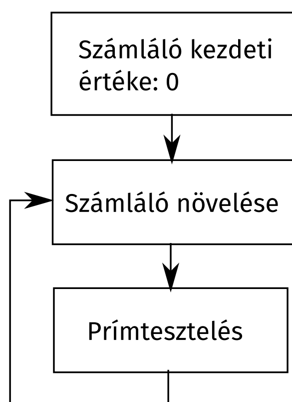
Különlegesség azonban, hogy a megszokott deque implementációk nem használhatóak a fedélzeti számítógépen, mert - megvalósításukból fakadóan - használatuk kritikus versenyhelyetet idézhet elő, hiszen adatkorrupció következik be olyankor, amikor a főprogram éppen levesz egy elemet a sor elejéről, és ezen művelet közben kap a processzor egy megszakításkérést, amelynek a megszakításkezelő rutinja szintén módosítaná a sort. Fogalmazhatunk úgy is, hogy a legtöbb deque implementáció nem szálbiztos (thread-safe).

5.2.1. Versenyhelyzetekről

A versenyhelyzet egy tipikus probléma a szoftveriparban, amely leggyakrabban hagyományos számítógépek párhuzamos programozásakor fordul elő. [8] [9] (Természetesen találkozhatunk vele a digitális technikában is.)

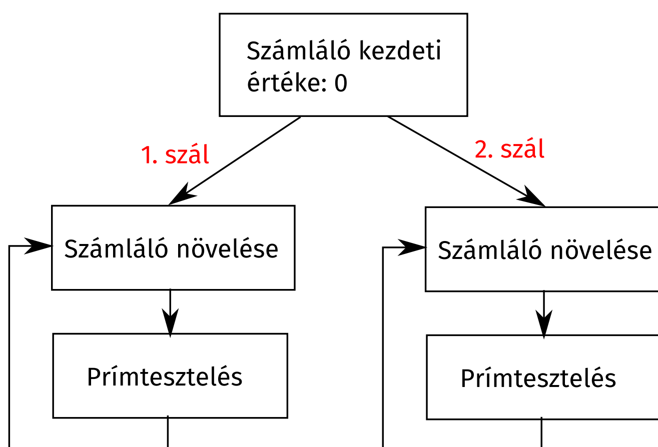
Analógia

Szeretnék egy analógiát bemutatni a probléma jobb megértéséhez az asztali számítógépek világában előforduló konkurenciaproblémákkal. Illusztrálásképpen tekintsünk először egy részletet egy egyszerű egyszálú programból, ami prímszámokat keres, például az 5.4 ábrán látható módon.



5.4. ábra. Analógia illusztrálásához prímszám kereső algoritmus

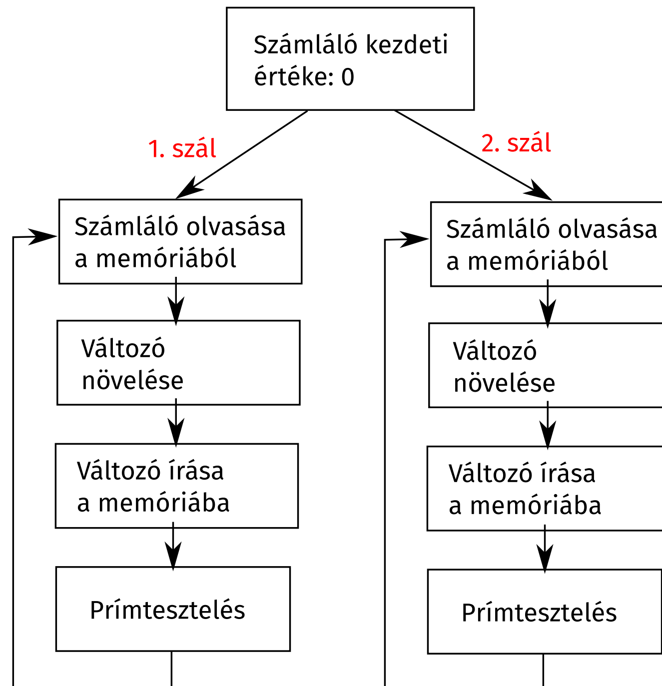
A fenti algoritmus természetesen helyesen működik, tekintsük most azonban azt az esetet, amikor két szál futtatja ugyanezt az algoritmust, ezt az 5.5 ábrán lehet megnézni.



5.5. ábra. Analógia illusztrálásához kétszálú prímszám kereső algoritmus

Ez szintén jónak tűnik, azonban valójában nem az. Sajnos ugyanis a legtöbb processzorarchitektúrán az inkrementálás (eggyel növelés) nem atomikus művelet, hanem három különböző részműveletből áll: a változó betöltése a memóriából, növelése eggyel, majd az eredmény visszaírása a memóriába. [10]

Tehát a ténylegesen a hardverben végrehajtódó működést az 5.6 ábrán ismertetem.



5.6. ábra. Analógia illusztrálásához kétszálú prímszám kereső algoritmus valódi működése

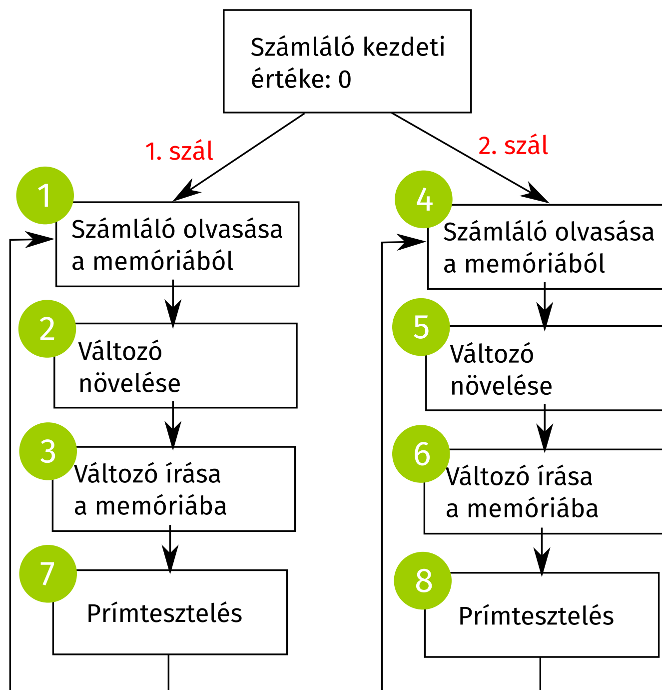
Ezen az ábrán már jól látszik, hogy az atomicitás illúziója nem tartható fenn, mert ha a két szál egymással párhuzamosan fut, akkor ugyan van olyan eset, amikor a program helyesen működik, de az elgondolás alapvetően nem helyes. [10]

Az algoritmus helytelenségének oka az, hogy nincs befolyása arra, hogy a két szál hogyan ütemeződik, ezt az aktuális futtató környezet dönti el. Így sajnos a program futásának eredménye függ attól, hogy az egyes műveletek végrehajtása milyen sorrendben kezdődik meg. [8] [9] [10]

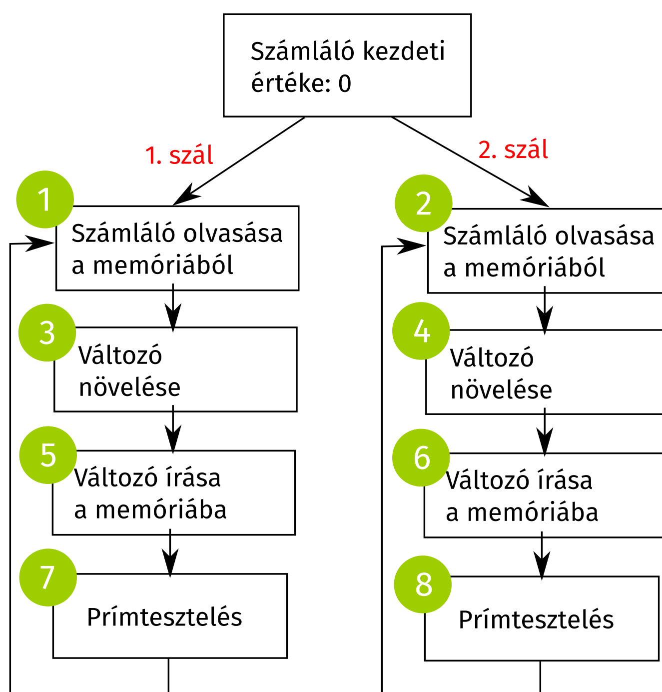
az 5.7 ábrán és az 5.8 ábrán láthatunk két példát a fenti algoritmus futására, melyek egyike jó eredménnyel, másika pedig rossz eredménnyel zárul. Természetesen olyan algoritmust, amelynek futása rossz eredménnyel is járhat, helytelen algoritmusnak tekintünk.

Mivel a SMOG-1 zsebműhold fedélzeti számítógépe felé alapvető elvárás a nagy megbízhatóság, ezért az ilyenfajta problémákat komolyan kell venni, hiszen, amikor a műhold már fent lesz a világűrben, nem fogjuk tudni az ilyen jellegű problémákat hibakereséssel megoldani.

Ebből a klasszikus példából azt láthatjuk, hogy mivel mindkét szál ugyanazt az értéket dolgozza fel és egyszerre próbálják az ugyanazon a memóriaterületen levő változó értékét olvasni és módosítani, a futás eredménye függeni fog attól, hogy a két szál hogyan ütemeződik, vagyis, hogy az egyes párhuzamosan futó műveletek milyen sorrendben kezdődnek meg egymáshoz képest. Példánkban ez azt eredményezheti, hogy bizonyos feladatokat egyáltalán nem végez el egyik szál sem, másikakat pedig mindkettő (feleslegesen) elvégzi. [9] [10]



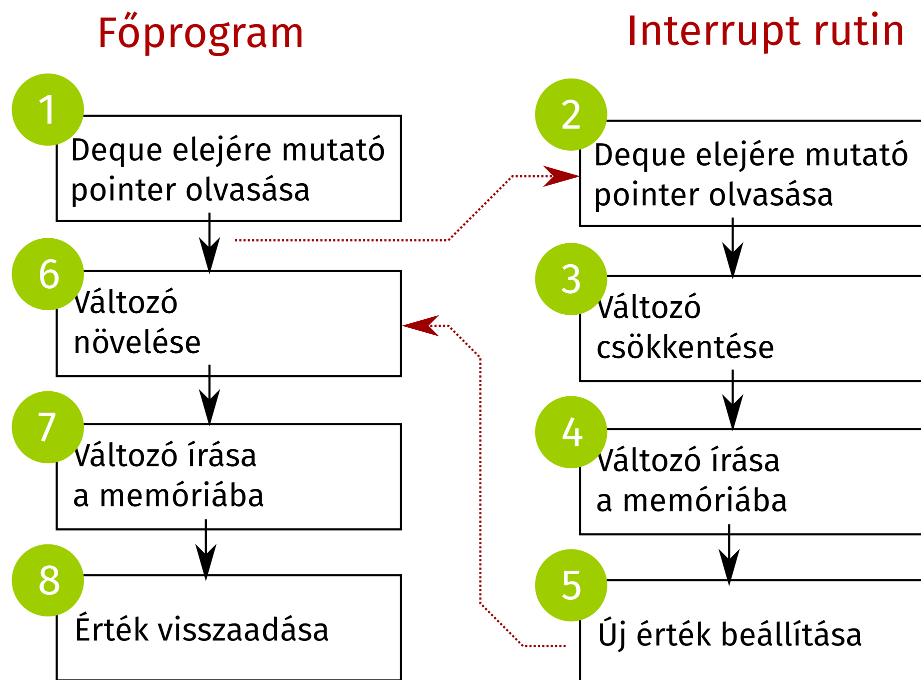
5.7. ábra. Versenyhelyzetet tartalmazó program jó kimenetelű futása



5.8. ábra. Versenyhelyzetet tartalmazó program helytelen kimenetelű futása

5.2.2. Versenyhelyzet jelentősége mikrokontrolleren

Jogos a kérdés, hogy ez a problematika miért és hogyan vetődhet fel, hiszen a SMOG-1 fedélzeti számítógépe mindössze egy processzort tartalmaz, tehát nincs benne valódi párhuzamosság. Ez így is van, viszont a megszakítások révén mégiscsak aszinkronitás kerül a rendszerbe. [17] A mi esetünkre, és a deque implementációra vonatkoztatva például ha a megszakításkezelő rutin egy „sürgős" feladatot ad hozzá, és a megszakítás pont akkor történt, amikor a főprogram épp levett egy elemet a sor elejéről, akkor előállhat egy olyan helyzet, amikor egy feladat elveszik, vagy a program másik feladatot hajt végre, mint amit kellene. Úgy is megfogalmazhatjuk, hogy a deque-hoz való hozzáadás és törlés művelete nem atomikus, hasonlóképpen a korábbi analógiához. Egy ilyen hibalehetőséget mutat be az 5.9 ábra.



5.9. ábra. Ha a megszakítás kérés rosszkor történik, elromolhat a főprogram algoritmusa

A gond itt is abban rejlik, hogy a memóriában megváltozik ugyan a változó tartalma, de a főprogramban levő kód még a régi értékkel dolgozik, mert a változtatás azután történt, hogy ő ezt az értéket már beolvasta.

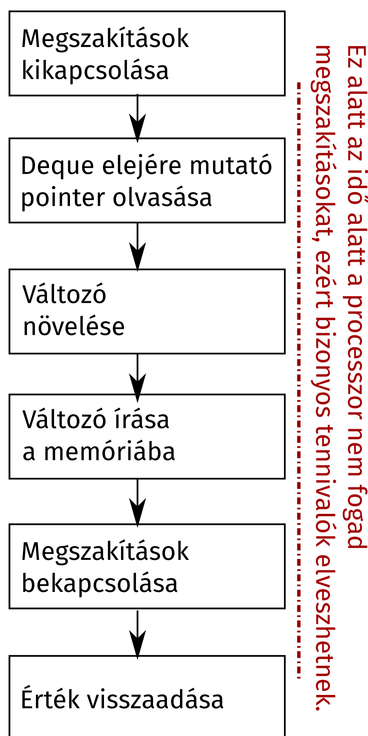
Ebben a példában jól megfigyelhető a konkurencia problémák egyik fő ismérve: sajátos körülmények kellenek az előidézésükhöz és nehezen reprodukálhatóak, ami lényegesen bonyolítja az ilyen hibák felderítését és javítását. Természetesen a műholdon nem szabad megengedni, hogy ilyen előfordulhasson.

A fentiekből tehát leszűrhetjük azt a következtetést, hogy a sztenderd, nem szálbiztos deque implementáció nem ad megfelelő megoldást a fedélzeti számítógép szoftvere számára.

5.2.3. Versenyhelyzet megoldása

Klasszikus körülmények között a szálak összehangolását kölcsönös kizárással valósítják meg, melynek eszközei az ún. szinkronizációs primitívek, mint például a szemaforok vagy zárak. [10] A szálak futása folyamán kritikus szakaszokat határoznak meg, amelyek olyan programrészletet jelentenek, amiknek semmiképpen nem szabadna egyszerre futniuk. [9] Ezek a konstrukciók azonban nem adnak megfelelő megoldást az itteni problémára, hiszen nincs valódi párhuzamosság, vagyis, amennyiben zárakat alkalmaznék a prioritásos sor „megvédésére", akkor az interrupt rutin egyszerűen elakadna, mert miközben ő fut, a főprogram sosem tudja a zárat feloldani. Ezt a jelenséget deadlocknak nevezzük. (A deadlock egy olyan helyzetet jelent, amikor egy szoftver olyan állapotba kerül, hogy megáll és nem tud véges időn belül befejeződni.) [8]

Triviális megoldás lenne a megszakítások teljes kikapcsolása a kritikus szakasz idejére, ez azonban nem megengedhető, mert ekkor csökkenne az eseményekre adott válaszidő, illetve előfordulhat, hogy egyes eseményeket a processzor nem is észlelne. Ezt szemlélteti az 5.10 ábra.



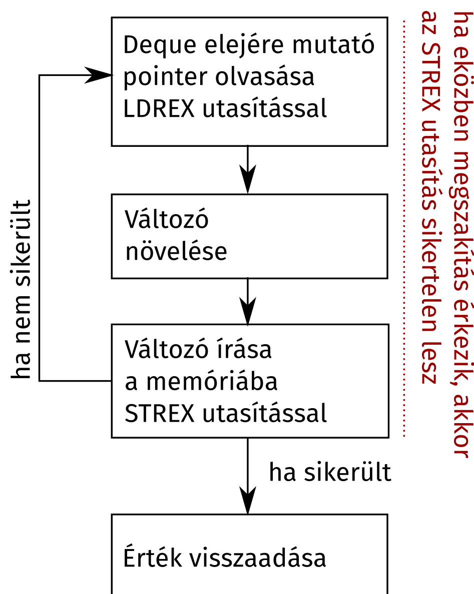
5.10. ábra. Megszakítások kikapcsolásával megvédhetjük a kritikus szakaszt.

Szerencsére a hardvergyártók is észrevették ezt a problémát, és a modern hardvereken találhatóak olyan atomikus utasítások, amelyek „azonnal" végeznek el bizonyos műveleteket. [11] (Természetesen az azonnaliság csak a felhasználó szempontjából tűnik úgy, a processzorok belső működése során nem teljesül.) Egy megoldás például az „összehasonlítás és csere" (compare and exchange), amely a memóriába írás előtt egy olvasást és összehasonlítást végez, és az írást csak akkor végzi el, ha a szóbanforgó memóriaterületen egy elvárt érték szerepel. Ezzel a módszerrel magától értetődően lehet szinkronizációt elérni, ha az algoritmust úgy írjuk meg, hogy sikertelen „összehasonlítás és

csere" esetén újrakezdje az épp végzett funkciót.

Az Intel processzorain a CMPXCHG utasítást [12] lehet erre a célra használni, az ARMv7-M architektúra esetén pedig az LDREX és STREX utasításpárost kínálja az ARM a programozók számára. [13] (Ezen utasítások nem részei az ARMv6-M utasításkészletnek, tehát nem elérhetőek a Cortex-M0 és Cortex-M0+ processzorokon. [13]) A fedélzeti számítógépben levő Cortex-M4 támogatja ezeket az utasításokat, és a GCC fordítóprogram (amellyel a szoftvert lefordítjuk) pedig rendelkezésünkre bocsát olyan beépített függvényeket (intrinsic function), amelyekkel könnyedén használhatóvá válnak ezek az utasítások. [14] [15] A működés egyszerű: ha az LDREX segítségével töltünk be valamit a memóriából, akkor, amennyiben az STREX segítségével tároljuk el, az STREX sikertelen lesz olyankor, ha az adott memóriaterületet a betöltés óta módosították. [13] Ezt a processzor hardveresen támogatja, és úgy működik, hogy az LDREX végrehajtása után figyelni kezdi, hogy az adott memóriaterülethez hozzáférték-e még, és ha igen, akkor az STREX kimenetele sikertelen lesz.

Ekkor, ha a processzor megszakításkérést kap, miközben a deque-t módosítja, akkor az STREX utasítás sikertelen lesz és a főprogram megismételheti a műveletet, így pedig a futás már helyes eredményt hoz. Tehát a versenyhelyzetmentes, helyes prioritásos sor az 5.11 ábra szerint működik.



5.11. ábra. LDREX és STREX utasításokkal megvédett deque

5.3. Energiatakarékosság

Az energiatakarékosságot sokan hardveres kérdésnek tekintik, azonban azonok a hardvereken, melyeken szoftver fut, bonyolódik a helyzet, mert a hardver hiába teremt lehetőséget különféle energiamegtakarító módszerek alkalmazására, ha a szoftver nem használja ki ezeket, akkor semmit nem érnek.

5.3.1. Eseményvezéreltség jelentősége az energiatakarékosságban

Klasszikus példa a szoftvertechnológiában a „busy wait” és a „sleep wait” közötti különbség. A „busy wait” esetben a szoftver, amikor vár valamilyen esemény bekövetkeztére, egy végtelen ciklust futtat, amely folyamatosan ellenőrzi, hogy az esemény bekövetkezett-e már (ez a módszert „polling” néven is ismert). Ellenben a „sleep wait”, amennyiben a várt esemény még nem következett be, alvó üzemmódba helyezi a processzort (vagy ha a szoftver egy operációs rendszer alatt fut, akkor átadja az operációs rendszernek a vezérlést), és egészen addig nem fut tovább egyáltalán, amíg valamilyen módon jelzést nem kap, hogy az esemény bekövetkezett. Hardverközeli esetben ilyen lehet egy interrupt. [17] Ennek pedig magasabb szintű analógiája az, amikor egy program üzenetet kap az operációs rendszertől. [18]

Tehát a busy wait esetben a program folyamatosan utasításokat ad a processzornak, míg a sleep wait esetében a processzor akár alhat, akár más feladatokat hajt végre. A sleep wait megközelítés megfelel az úgynevezett Hollywood elvnek (amely úgy szól: „majd mi hívunk, te ne hívj minket”). Ez az elv igen fontos az eseményvezérelt programozás megértéséhez. [16]

Természetesen olyan helyeken, ahol a fogyasztás másodlagos (például asztali számítógépben) megengedhető a busy wait jellegű megközelítés, de manapság a hordozható eszközök terjedésével az energiakészlet szűkössége miatt egyre inkább előtérbe kerül a fogyasztás. Egy átlagos mai számítógép (és okos eszköz, pl. telefon vagy tablet) processzora normál használat mellett is az ideje nagyrészét valamilyen készenléti vagy alvó üzemmódban tölti. Például UNIX alapú rendszereken, ha Intel processzort használunk a „powertop” program segítségével nyerhetünk igen részletes információt arról, hogy a CPU az ideje hány százalékát milyen üzemmódban tölti. Például jelen dokumentum írásakor (általános felhasználás mellett) a számítógépben levő négy feldolgozóegység közül három az ideje több, mint 95%-át készenléti módban tölti (a negyedik esetében ez az érték 60%).

Egy zsebműhold olyan rendszer, ahol minden egyes milliamper számít [2] [4], tehát a busy wait megközelítés alkalmazhatósága teljességgel kizárt. Szerencsére az eseményvezérelt programozás természetes módon teszi lehetővé a processzor alacsonyabb energiafogyasztású üzemmódjainak használatát.

A fedélzeti számítógépben használt Silicon Laboratories „Wonder Gecko” EFM32WG mikrokontroller is többféle energiatakarékos üzemmóddal rendelkezik. Ezek elnevezése EM1, EM2, EM3 és EM4. [22] (az EM itt az „energy mode” rövidítése.) Ezen üzemmódokba a szoftver a kontroller energiamenedzselő egységének (energy management unit, EMU) használatával juthat el. [22] [19]

- EM1 módban lekapcsol az EFM32WG-ben levő processzormag és csak a perifériák működnek.
- EM2 módban a mikrokontroller lekapcsolja a nagyfrekvenciájú belső oszcillátorát is, így az arról működő perifériák is megállnak.
- EM3 módban már csak az néhány alacsony órajelről működő belépített periféria marad bekapcsolva, ezek azonban továbbra is képesek lehetnek felébreszteni a processzort.

- EM4 módból már csak reset (teljes újraindítás) segítségével lehet kijutni, amelyet például a backup RTC (real time counter) idézhet elő.

Természetesen lehetőség van az EFM32WG órajelének megváltoztatására is, és a processzormag alacsonyabb órajelen kevesebb energiát fogyaszt. Ez a módszer azonban kétélű, hiszen alacsonyabb órajelen ugyanazokat a műveleteket tovább tart elvégezni, tehát a processzor kevesebb időt tölthet energiatakarékos módban. Vagyis hiába csökken a pillanatnyi fogyasztás, az átlagfogyasztás nőhet is. [19] [24]

Egyes mérések [19] szerint jobban megéri futás közben a processzort magas órajelen működtetni, hogy a feladatait hamar befejezze és minél több időt tölthessen alvó üzemmódban, mert így lehetséges az átlagfogyasztást a lehető legalacsonyabbra csökkenteni.

5.3.2. Energiatakarékosság a CPU tehermentesítésével: DMA

Bizonyos esetekben azzal spórolhatjuk meg a legtöbb energiát, ha a processzort úgy tehermentesítjük, hogy bizonyos feladatokat specializált hardverekre bízunk. Ilyenkor a processzormag akár alvó módban is töltheti az idejét, miközben a specializált hardveregység ugyanazt a feladatot gyorsabban és alacsonyabb fogyasztással tudja elvégezni. Ilyen feladat a legtöbb I/O (input / output) művelet is, mint például az adatok mozgatása a memóriában vagy a memória és a perifériák között.

Erre a feladatra találták ki a közvetlen memóriaelérést (direct memory access, DMA), amely lehetővé teszi, hogy különféle hardverelemek egymás között a processzor beavatkozása nélkül mozgathassanak adatokat. Ennek eszköze egy ún. DMA vezérlő. A CPU-nak mindössze közölnie kell a DMA vezérlővel, hogy mennyi adatot, honnan és hová kell mozgatni, az pedig teszi a dolgát, miközben a CPU más feladatokkal foglalkozhat vagy alvó üzemmódba helyezheti magát. A DMA vezérlő, ha elkészült a feladattal, megszakításkérés (interrupt request) segítségével jelzi azt a processzornak. [23]

Mint minden modern rendszerben, az EFM32WG-ben is található DMA vezérlő, amely egyszerre akár több adatmozgatást is képes több csatornán végezni (megfelelő időosztással az egyes csatornák között). [22] [21] Ezt a funkciót használja ki a SMOG-1 fedélzeti számítógépe is a perifériákkal való kommunikáláskor.

5.3.3. Alacsony fogyasztás elérése soros port használatakor

Az EFM32WG soros porti vezérlője képes kihasználni a DMA vezérlőt, tehát az onnan jövő adatok mozgatásához nincs szükség a CPU beavatkozására. [22] [20]

Ezen kívül azonban rendelkezik még egy hasznos funkcióval, ez pedig a frame detection (keret érzékelés) [20], ami azt jelenti, hogy bizonyos előre beállítható adatbájtokat képes érzékelni a hardver, és ezek érkezésekor képes megszakításkérést küldeni a processzor felé.

Két ilyen detektálható bájt állítható be:

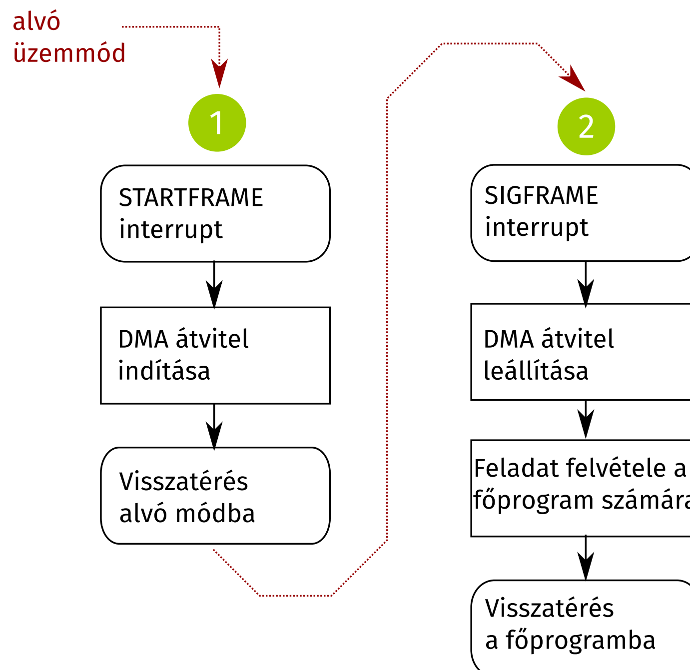
- Start frame: egy üzenet kezdetét jelző bájt.
  Érkezésekor a STARTFRAME interruptot küldi nekünk a soros port.

- Signal frame: egy üzenetben valamilyen jelentéssel bíró bájt, például az üzenet befejeztének felismerésére használható.
  Egy ilyen bájt beérkezésekor a soros porttól SIGFRAME interruptot kapunk.

A fentiek segítségével olyan soros porti kommunikáció valósítható meg, amely a lehető legkevésbé igényli a processzor beavatkozását. Ennek igénybevételéhez az alábbiakra van szükség:

1. Soros port konfigurálása
   Beállítjuk a soros port vezérlőjében a start frame-et és a signal frame-et.
2. DMA vezérlő konfigurálása
   Létrehozunk egy DMA csatornát a soros porti adatok küldésére és még egy csatornát a soros portról érkező adatok fogadására. A DMA vezérlő üzemmódjai [21] közül ebben az esetben a „Basic" üzemmódra van szükség.
3. DMA vezérlő bekapcsolása
   Órajelet adunk a DMA vezérlőnek és engedélyezzük a működését.
4. Soros port bekapcsolása
   Órajelet adunk a soros port vezérlőjének és engedélyezzük a működését.

A kommunikáció beállítása után használatba is vehetjük azt. Ezt az 5.12 ábra mutatja.



5.12. ábra. Alacsony fogyasztás megvalósítása soros porton.

A folyamat tehát a következő:

1. A soros port észleli az üzenet elejét jelző bájt beérkezését, küld egy STARTFRAME megszakítást.

2. A megszakításkezelő rutin elindítja a DMA átvitelt azon a fogadó csatornán, amit az előző oldal leírása szerint beállítottunk.

3. A processzor visszatérhet alvó üzemmódba (vagy folytathatja egyéb elfoglaltságait, ha voltak).

4. A soros port észleli az üzenet végét jelző bájtot, és küld egy SIGFRAME interruptot a CPU felé, ezzel felébresztve azt, ha aludt.

5. A megszakításkezelő rutin leállítja az előzőleg indított DMA átvitelt.

6. A megszakításkezelő rutin feladatot ad a főprogram számára azáltal, hogy felveszi azt a korábbiakban taglalt prioritásos sorba.

Ez a megoldás nagyságrendekkel energiatakarékosabb a szokásos naiv megvalósításnál, amelyben a processzor busy wait jelleggel folyamatosan figyeli a soros porton beérkező adatokat és vár, amíg az összes adat meg nem érkezett. Az energiatakarékos megoldás nem csak azt teszi lehetővé, hogy a CPU aludjon, miközben az üzenetet a soros port fogadja, hanem azt is, hogy közben akár egyéb feladatokat lásson el.

5.4. Optimalizálás

Az OBC szoftverét a Silicon Laboratories által is preferált GCC (GNU Compiler Collection) fordítóprogrammal fordítom le, amely számos lehetőséget ad a készült kód optimalizálására. Az optimalizálás nem csak a sebesség, hanem ennek következményeképp az energiatakarékosság miatt is fontos, hiszen ha a feladatait gyorsabban végzi el a szoftver, akkor a processzor több időt tölthet alvással.

Azonban az optimalizálás veszélyes is lehet. Körültekintőnek kell lenni, mert a modern optimalizációs módszerekkel rendelkező fordítóprogramok olyan lehetőségeket használnak fel a C / C++ nyelvek specifikációiban, amelyek gyakran a programozó által nem várt mellékhatásokkal járnak. [25] Ezek elkerülése érdekében a SMOG-1 mérnöki példányát alapos tesztelésnek fogjuk kitenni.

Hatodik fejezet

Fedélzeti számítógép feladatai

6.1. Analógia operációs rendszerekkel

Párhuzamot találhatunk az OBC szoftvere és egy operációs rendszer között. Az előbb részletezett event loop megvalósítás tulajdonképpen úgy működik, mint egy nagyon kezdetleges ütemező. Ha így nézzük, akkor az elvégezendő feladatok processzeknek, az OBC szoftvere pedig egyfajta operációs rendszernek fogható fel. Hardveres korlátok miatt nem lenne gazdaságos ennél bonyolultabb rendszert kialakítani, mert az EFM32WG processzorában nincs sem MMU (memory management unit), sem elég operatív memória ahhoz, hogy érdemes legyen több folyamatot párhuzamosan futtatni. [22]

Használhatnánk valamelyik ún. valósidejű operációs rendszert (real-time operating system, RTOS), ezt azonban nem tesszük. Mivel a műhold szoftverének nagy megbízhatóságúnak kell lennie, semmilyen olyan szoftverelemet nem kívánunk feljuttatni, amit nem mi fejlesztettünk ki, mert third party szoftverelemek ellenőrzése és hibajavítása nehezen megvalósítható feladat.

Természetesen lehetne preemptív multitaszkingot megvalósítani, úgy, hogy context switch esetén az operatív memória teljes tartalmát átmásoljuk a flash memóriába, de ez mikrokontrolleres hardveren lassú és gazdaságtalan folyamat lenne.

6.2. Fájlrendszer, tömörítés

Az OBC-hez tartozik 8 Megabyte redundáns flash memória. Ezt a tárterületet használjuk majd a mért adatok és a telemetria tárolására. Amikor a műhold a földi állomással kommunikál, akkor innen olvassa majd ezeket az adatokat és küldi le őket.

Ehhez egy egyszerű fájlrendszert kell kialakítani, amely ACID tulajdonságokkal (atomicity, consistency, isolation, durability) rendelkezik majd, valamint a hely kihasználtság hatékonysága érdekében érdemes lesz az adatainkat tömöríteni.

6.3. Titkosítás

Az EFM32WG kontrollerbe épített hardveres AES-128 támogatást fogjuk használni arra, hogy ne tudjon akárki parancsokat küldeni a műholdnak. A földi állomástól érkező parancsok titkosításra kerülnek, és ezeket a kontroller dekódolja.

6.4. Hibajavító kódolás

Természetesen az űreszközökkel való kommunikáció folyamán elengedhetetlen a hibajavító kódolás alkalmazása. Ezen kódolások teszik lehetővé, hogy a korlátozott energia ellenére a SMOG-1 üzenetei eljussanak hozzánk. A hibajavító kódolás az algoritmuselmélet egyik részterülete, és arra alkalmas, hogy zajos csatornán keresztül növelje az átviteli rendszer jel-zaj viszony tűrőképességét, s ezáltal annak valószínűségét, hogy a fogadó oldalra az az üzenet jut el, amit a küldő feladott. [29]

A SMOG-1 projektben kétféle kódolást alkalmazunk: a lefelé irányuló rádiós összeköttetésen a rádióamatőr műholdakban már bevált AO-40 kódot [28], a műhold felé tartó üzeneteket pedig Golay kóddal [27] védjük meg, mert a tapasztalatok szerint kis adatmennyiségű üzenetek esetén ez válik be a legjobban, mert ennek a hibaarány-görbéje (BER görbéje) a legjobb.

A hibajavító kódolók algoritmusainak implementációja Szabó Lóránt és jómagam munkája, de a ezek pontos ismertetése túlmutat jelen dolgozat keretein.

6.5. Telemetria gyűjtés

A fedélzeti számítógép folyamatosan telemetriát gyűjt a műhold állapotáról. Telemetria részei az alábbiak:

- Napelem oldalak állapota és a pillanatnyi bejövő teljesítmény
- Műhold fedélzeti hőmérséklete
- Szabályozatlan energiabusz feszültsége, amelyet a PCU-tól kér le
- Mozgásérzékelő szenzortól gyűjtött adatok a műhold aktuális mozgásállapotáról
- COM rendszer pillanatnyi fogyasztása, rajta eső feszültség és rajta folyó áram, hőmérséklet, RSSI (received signal strength indicator) értékek és a rádiós chip státusz információi

Ezeket az adatokat a flash memóriában tárolja és parancsra a COM rendszerrel leküldi a földi állomásra.

6.6. Spektrummérés ütemezése

Bizonyos időközönként az RTCC jelez az EFM32 mikrokontrollernek (megszakítás kéréssel). Ilyenkor az OBC üzenetet küld full-duplex UART-on a COM rendszernek, amelyben megkéri azt,

hogy indítson el egy mérést a spektrumanalizátorral. Amikor a mérés kész, a COM visszajelez az OBC-nek, amely fogadja, feldolgozza és tárolja a mérési adatokat.

Amikor a SMOG-1 legközelebb elhalad Magyarország felett, akkor pedig mindezen adatokat a földi állomás lekérdezi a műholdtól.

6.7. Kapcsolat a többi alrendszerrel

6.7.1. PCU (power control unit)

A PCU egy PIC24 családba tartozó mikrokontroller, ami az EPS-hez tartozik. Az EPS Géczy Gábor munkája, és a PCU feladatai közé tartozik a szabályozott energiabusz és az akkumulátor felügyelete, valamit az OBC-t ellátó LSW-k vezérlése, és annak eldöntése, hogy a redundáns pár közül melyik OBC üzemeljen. [2]

Az OBC és a PCU között „one-wire" half-duplex UART digitális összeköttetés van, amelyen telemetriaadatokat és egyéb információkat cserélhetnek egymással.

6.7.2. COM (rádiókommunikáció és spektrumanalizátor)

Rádiókommunikációért egy Silicon Laboratories gyártmányú Si1062 integrált áramkör lesz a felelős. Ez a chip tartalmaz a rádiófrekvenciás áramkörön kívül egy beépített 8051-es mikrokontrollert is, amely által „helyi intelligencia" valósítható meg a COM rendszerben. Ez az alrendszer Dudás Levente munkája. [1]

A spektrumanalizátor egy Si4464 típusú IC, amelyet az Si1062-ben levő mikrokontroller fog vezérelni SPI buszon keresztül. Ebben az elrendezésben tehát a földi állomással való kommunikáció és a spektrumanalizátor az OBC szemszögéből egyetlen egységes egészként viselkedik. A COM és az OBC között full-duplex UART digitális összeköttetés lesz, amelyen spektrum mérési adatokat, földről vett, és a földre küldendő információkat cserélhetnek egymással nagy sebességgel. (Terveink szerint akár száz kilobyte / sec sebességgel.)

6.7.3. Áramkorlátozó kapcsolók

A fedélzeti számítógéphez tartozik négy darab limiter switch (áramkorlátozó kapcsoló), amelyeket Géczy Gábor tervezett. Ezek a kapcsolók a COM rendszer egy-egy redundáns egységét, valamint az antenna nyitó mechanizmus két példányát védik és felügyelik. Afféle „okos biztosítóként" egy bizonyos megengedett áramérték felett kioldanak, így megakadályozva, hogy az adott áramkörben egy rövidzárlat energiaveszteséget okozzon vagy a műhold egészét tönkretegye. [2] Ezek a kapcsolók nem csak az automatikus kioldásra jók, hanem ennek megtörténte esetén képesek digitális jelet adni a mikrokontroller felé, valamint áram- és feszültségmérő kapcsolásokkal is el vannak látva, amelyekről az OBC telemetriát gyűjt.

Az OBC-nek feladata két ilyen kapcsoló vezérlése és tőlük áram- és feszültségtelemetria mérése (analóg-digitál átalakítók segítségével) és rögzítése. Természetesen a fedélzeti számítógép

mikrokontrollerei (a redundáns pár mindkét tagja) is ilyen kapcsolókkal vannak megvédve, de azokat a kapcsolókat az EPS vezérli.

Hetedik fejezet

Összefoglaló

A fedélzeti számítógép legfőbb célja tehát az, hogy a műhold egészét vezérelje és nagy megbízhatósággal álljon rendelkezésre. „Bombabiztosnak” kell lennie, mert a műholdon nem engedhető meg az, mint napjaink fogyasztói termékeiben, ahol elterjedt a „ha nem működik, indítsuk újra” szemlélet. (Nem tudunk kimenni és újraindítani, ha nem működik.) Ezt a megbízhatóságot hardveres és áramköri redundanciával, valamint a szoftverelemek körültekintő megtervezésével és implementálásával érem el.

Mivel az OBC minden más alrendszerrel kapcsolatban van, elengedhetetlen számomra, hogy a többi alrendszer működésével is tisztában legyek. Ezért vállaltam részt a műhold rendszertervének kidolgozásában is, és ezért fektettem hangsúlyt a rendszerterve a dolgozatom elején.

7.1. Köszönetnyilvánítás

Szeretném megköszönni elsők között konzulensemnek, Dudás Leventének áldozatos munkáját, amellyel felbecsülhetetlen mértékben hozzájárult szakmai előmenetelemhez. Neki köszönhetem, hogy elmélyültem a hardverek világában, és tőle tanultam a villamosmérnöki szakma gyakorlati oldalát is. Hálával tartozom továbbá Herman Tibornak és Géczy Gábornak, akik tehetségükkel és szakértelmükkel segítettek a fedélzeti számítógép hardverének fejesztésében.

Irodalomjegyzék

[1] Dudás Levente, *The Spectrum Monitoring System of Smog-1 Satellite*, MAREW 2015 konferencia

[2] Géczy Gábor, *SMOG-1 Másodlagos Energiaállátó Rendszere*, MSc Önálló laboratórium beszámoló, 2014 ősz és 2015 tavasz

[3] Géczy Gábor, *SMOG-1 Másodlagos Energiaállátó Rendszere*, MSc diplomaterv, 2015 ősz

[4] Herman Tibor, *SMOG-1 Elsődleges Energiaállátó Rendszere*, MSc Önálló laboratórium beszámoló, 2014 ősz

[5] Herman Tibor, *SMOG-1 Elsődleges Energiaállátó Rendszere*, MSc diplomaterv, 2015 tavasz és ősz

[6] Kristóf Timur, *PocketQube műhold fedélzeti számítógépének tervezése*, BSc Önálló laboratórium beszámoló, 2015

[7] Jáger Dávid és Török Péter, *PocketQube műhold numerikus hőtani szimulációja*, TDK dolgozat, 2014

[8] Microsoft, *Description of race conditions and deadlocks*
(elérés dátuma: 2015. október 26.)
`https://support.microsoft.com/en-us/kb/317723`

[9] David A. Wheeler, *Avoid race conditions*
(elérés dátuma: 2015. október 26.)
`http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/avoid-race.html`

[10] Paul Butcher, *Seven Concurrency Models in Seven Weeks*
(elérés dátuma: 2015. október 26.)
`https://pragprog.com/book/pb7con/seven-concurrency-models-in-seven-weeks`

[11] IBM, *Multithreaded data structures for parallel computing: Part 2, Designing concurrent data structures without mutexes*
(elérés dátuma: 2015. október 26.)

http://www.ibm.com/developerworks/aix/library/au-multithreaded_structures2/index.html

[12] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual*
(elérés dátuma: 2015. október 26.)
http://www.intel.com/Assets/en_US/PDF/manual/253666.pdf

[13] ARM Infocenter, *ARM Synchronization primitives: LDREX and STREX*
(elérés dátuma: 2015. október 26.)
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0008a/ch01s02s01.html

[14] ARM Infocenter, *ARM C Language Extensions*
(elérés dátuma: 2015. október 26.)
http://infocenter.arm.com/help/topic/com.arm.doc.ihi0053c/IHI0053C_acle_2_0.pdf

[15] GCC dokumentáció, *ARM C Language Extensions page*
(elérés dátuma: 2015. október 26.)
https://gcc.gnu.org/onlinedocs/gcc/ARM-C-Language-Extensions-_0028ACLE_0029.html

[16] Stephen Ferg, *Event-Driven Programming: Introduction, Tutorial, History*
(elérés dátuma: 2015. október 26.)
http://sourceforge.net/projects/eventdrivenpgm/

[17] Scott Rosenthal, *Interrupts might seem basic, but many programmers still avoid them*
(elérés dátuma: 2015. október 26.)
http://www.sltf.com/articles/pein/pein9505.htm

[18] Red Hat, *Hardware Interrupts*
(elérés dátuma: 2015. október 26.)
https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_MRG/1.3/html/Realtime_Reference_Guide/chap-Realtime_Reference_Guide-Hardware_interrupts.html

[19] Silabs, *AN0007 - Energy Modes*
(elérés dátuma: 2015. október 26.)
https://www.silabs.com/Support%20Documents/TechnicalDocs/AN0007.pdf

[20] Silabs, *AN0017 - Low Enery UART*
(elérés dátuma: 2015. október 26.)
https://www.silabs.com/Support%20Documents/TechnicalDocs/AN0017.pdf

[21] Silabs, *AN0013 - DMA*
(elérés dátuma: 2015. október 26.)
https://www.silabs.com/Support%20Documents/TechnicalDocs/AN0013.pdf

[22] Silabs, *EFM32WG Reference Manual*
(elérés dátuma: 2015. október 26.)
http://www.silabs.com/Support%20Documents/TechnicalDocs/EFM32WG-RM.pdf

[23] National Instruments, *DMA Fundamentals on Various PC Platforms*
(elérés dátuma: 2015. október 26.)
http://cires1.colorado.edu/jimenez-group/QAMSResources/Docs/DMAFundamentals.pdf

[24] Bhavin Turakhia, *Understanding CPU utilization and Optimization*
(elérés dátuma: 2015. október 26.)
http://careers.directi.com/display/tu/Understanding+CPU+Utilization+and+Optimization

[25] Chris Lattner, *What Every C Programmer Should Know About Undefined Behavior*
(elérés dátuma: 2015. október 26.)
http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

[26] *GCC Options That Control Optimalization*
(elérés dátuma: 2015. október 26.)
https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc/Optimize-Options.html

[27] Golay, Marcel J. E. (1949). *Notes on Digital Coding*. Proc. IRE 37: 657.

[28] Phil Karn, *Proposed Coded AO-40 Telemetry Format*
(elérés dátuma: 2015. október 26.)
http://www.ka9q.net/papers/ao40tlm.html

[29] Györfi-Győri-Varga, *Információ- és kódelmélet*

[30] Adesto Technologies, *AT45DB641E adatlap*
(elérés dátuma: 2015. október 26.)

[31] Micro Crystal Switzerland, *RV-3049-C3 adatlap*
(elérés dátuma: 2015. október 26.)

[32] Invensense, *MPU-9250 adatlap*
(elérés dátuma: 2015. október 26.)