Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Méréstechnika és Információs Rendszerek Tanszék

Door Attila

Újrakonfigurálhatóság szenzorhálózatban

BSc szakdolgozat

Konzulens

Orosz György

Budapest, 2015

Tartalomjegyzék

Hallgatói nyilatkozat 1
Kivonat 2
Abstract 3
1 Bevezetés 4
2 Rendszerterv 6
3 A hardveres és szoftveres környezet bemutatása 9
3.1 Hardver 9
3.1.1 Az AVR mikrovezérlő kártya 10
3.1.2 atmega128 flash memóriájának felépítése 11
3.1.3 Az atmega128 bootloader funkciója 15
3.1.4 ISM sávú rádiós kártya 17
3.1.5 A DPY-LED perifériakártya 18
3.1.6 UART bemutatása 18
3.1.7 I2C bemutatása 19
3.1.8 Külső EEPROM chip bemutatása 21
3.2 Szoftver 22
4 Az alkalmazás strukturális áttekintése 25
4.1 Topológia 25
4.2 Vezetéknélküli kommunikáció protokolja 26
4.2.1 Adatstruktúra és üzenetformátum áttekintése 26
4.2.2 A cím mező kezelése 27
4.3 A program végrehajtásának rövid áttekintése 29
4.3.1 PC-master adat átvitel 29
4.3.2 Master slave adat átvitel 31
4.3.3 A program írása a flashbe 32
5 A modulok implementálása 33
5.1 PC oldali szoftver 33
5.1.1 Felhasznált könyvtárak 33
5.1.2 A program működésének rövid áttekintése 33
5.1.3 UART port használata 34
5.2 A master mote működése 35
5.2.1 Felhasznált függvénykönyvtárak 35

5.2.2 A program működésének rövid bemutatása....................................36
5.3 Slave mote működése..39
5.3.1 Felhsznált függvénykönyvtárak..39
5.3.2 A program működésének rövid áttekintése....................................39
5.3.3 A bootloader a programozói gyakorlatban......................................40
6 Eredmények..42
7 Összefoglaló értékelés...47
8 Irodalomjegyzék...48

Hallgatói nyilatkozat

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

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

Kelt: Budapest

Kivonat

A beágyazott rendszerekben gyakran szükség van az eszközöket működtető programok frissítésére. Ez a frissítési folyamat különösen nehézkes vezeték nélküli rendszerek esetében, ahol például az eszközök nehezen megközelíthető helyen találhatóak (például vezetéknélküli szenzorhálózatok).

Az alább tárgyalt alkalmazás a BME-MIT által fejlesztett mikrokontrolleres kártyán került megvalósításra. Az alkalmazás célja, hogy lehetővé tegye a felhasználó számára, hogy távolról újrakonfigurálhassa az adot eszközt, egy adott topológiában. A szükséges alkalmazás függ a hardveres környezettől amiben megvalósítani kívánjuk, ezért ebben a dolgozatban egy egyszerű demó alkalmazáson keresztül kerül bemutatásra a módszer, amivel a feladat végrehajtható.

A dolgozat célja bemutatni a tervezés során fellépő szempontokat amik alapján készült, illetve ismertetni az ezzel járó megkötéseket, részletesen bemutatni a tervezés műszaki paramétereit, eljárásokat, az ehhez felhasznált eszközöket és a megvalósításban szereplő elemek funkcionális működését. A dolgozat végén sor kerül a munka összegzésére, a fejlesztésből és az ennek következtében létrejött alkalmazás teszteléséből levont következtetések összegzésére. Továbbá a dolgozat célja kitekintést nyújtani az alkalmazás jövőli felhasználására és a további fejlesztések lehetséges irányára.

Abstract

Sometimes, it is very complicated to reconfigurate sensors within industrial sensor systems after they were built in. This problem, werewith a developer faces pretty often, inspired me on the basic idea of making an application which helps to modify the program on my sensor. First, a device that would give a platform for the application is needed. At Department of Measurement and Information Systems (MIT) of Budapest University of Technology and Economics, a microcontroller based platform called „MIT MÓT" is used. Therefore it was convenient to use this particular platform to implement my application. This MIT MÓT has an industrial radio frequency interface, which allows to build up a sensor system from these devices.

The purpose of this thesis is the following. First, it is supposed to provide a description to the reader about basic embedded system design concepts. Second, it describes the mechanism of this reconfiguration process through an easy example. And finally, it may serve as a hint for designing a future application. In this thesis, I worked out an application which is able to reconfigurate a microcrontroller via a radio frequency interface and take part in the realization of a sensor system.

1 Bevezetés

Nem csak az iparban, hanem a hétköznapokban is gyakran előfordul, hogy egy tetszőleges platformon egy már kész program fut, és mi ezt szeretnék utolólag módosítani, akár frissítés akár egyéb célból. Gondoljunk csak a mobiltelefonok firmware frissítésére.

Hagyományosan ez elég körülményes, mivel az eszközünket csatlakoztatnunk kell a programozó egységhez, és azzal a memóriájába égetni az új programkódunkat. Sajnos sok esetben ez vagy egyáltalán nem vagy nagyon körülményesen kivitelezhető. Gondoljunk csak bele, a sok millió kiadott mobiltelefont sem lehet visszahívni, vagy a beépített szenzorunkat kibontani, felprogramozni és visszaépíteni.

Ebből kifolyólag adott az igény, hogy valami más módot találjunk a szoftver frissítésére. Még ha ez bizonyos megkötésekkel is jár, illetve további követelményeket is kénytelenek vagyunk felállítani a megoldással szemben.

- Például, hogy stabil és biztonságos legyen, mivel ha elrontjuk a szoftver módosítását könnyen használhatatlanná tudjuk tenni a készülékünket.
- Ezenfelül könnyen alkalmazhatónak kell, hogy legyen, vagyis ez a keret amit mi létrehoztunk ne zavarja, nehezítse meg az eszközön futó, célalkalmazás futását, vagy fejlesztését.
- Ne foglaljon el sok helyet a célalkalmazásunk elől a memóriában.
- Lehetőleg platformfüggetlen legyen, vagyis a megoldás könnyen átültethető legyen más mikrokontrolleres platformokra is.
- Flexibilis legyen, vagyis később könnyen módosítható, továbbfejleszthető legyen.

A hardver pedig, amin megvalósításra került az alkalmazás a Budapesti Műszaki és Gazdaságtudmányi Egyetem, Méréstechnika és Információs rendszerek tanszék által fejlesztett mikrokontrolleres kártya ún. MIT MÓT, ami első sorban oktatási célokra készült. Jellegénél fogva ez egy nagyon sokoldalú és könnyen használható eszköz, ezért kézenfekvő volt, hogy rá essen a választás, és mivel eddig is széles körben használva volt különböző projektek, feladatok során. Ebből kifolyólag elég jól dokumentált, illetve a fejlesztők a rendelkezésünkre bocsájtanak számos, már előre megírt függvénykönyvtárat ami nagyban elő tudja segítani a fejlesztést.

1 Bevezetés

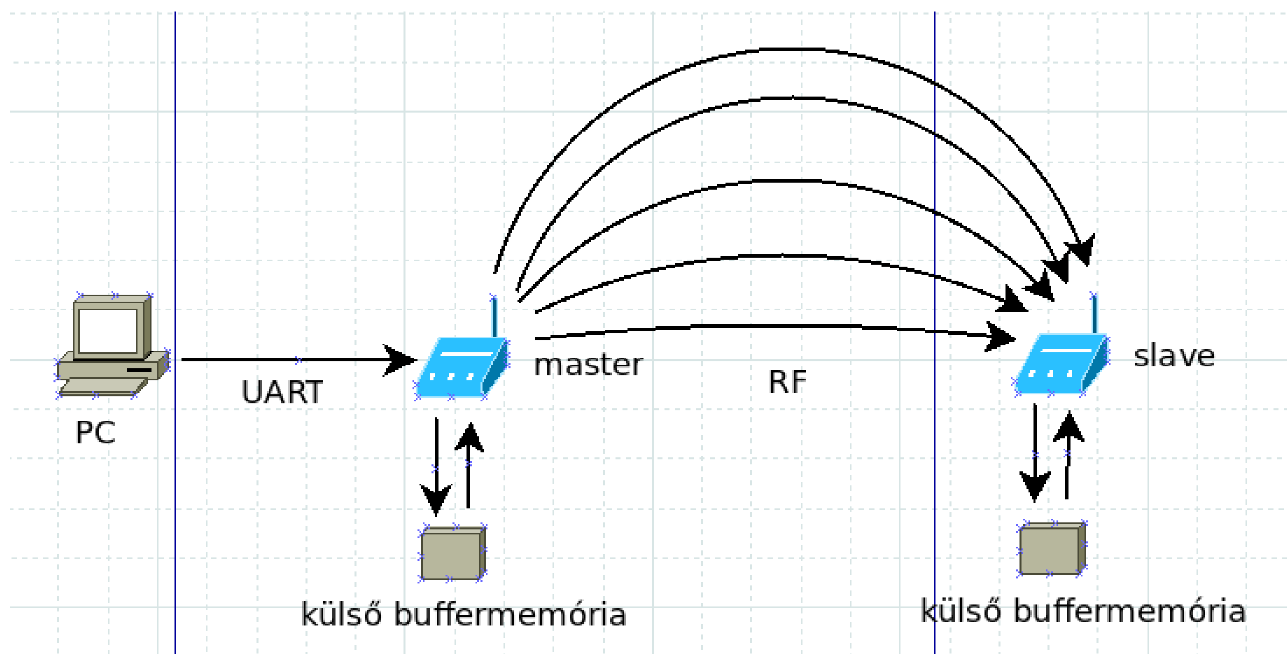
Az alább tárgyalt dolgozat betekintést kíván nyújatni a tervezés lépéseibe, illetve egy példa alkalmazást is be fogok mutatni, ami segítséget nyújthat a későbbiekben egy konkrét alkamazás fejlesztéséhez.

A dolgozat a következőképpen épül fel. A második fejezetben a rendszer felépítéséről adok átfogó képet. A harmadik fejezetben a felhasznált eszközök, illetve pár a beágyazott rendszerekben gyakran előforduló fogalom bemutására kerül sor. A negyedik fejezetben az olvasó egy rövid áttekintést kap a program felépítéséről, illetve bemutatásra kerül a program futásának folyamata is. Az ötödik fejezetben a rendszer moduljainak részletes leírása található, ami egyben bemutatja a rendszer demonstrálását szolgáló példa alkalmazás is. A hatodik fejezetben bemutatom a létrehozott példa alkalmazás működését. Az utolsó fejezetben összegzem a dolgozatban tárgyalt alkalmazás használhatóságát, illetve kitekintést nyújtok a lehetséges jövőbeli fejlesztések irányára is.

2 Rendszerterv

Az alkalmazás ötlete onnan ered, hogy a szenzorhálózatokban sokszor problémát okoz az szenzorbázisaink utólagos felkonfigurálása, amire sokszor nagy szükség van. Ennek a megvalósításához természetesen olyan eszközre van szükség ami ezt lehetővé teszi, én erre a tanszéken már rendelkésre álló MIT MÓT-ot (lásd. 3.1. fejezet) választottam, mivel ez a problémakör még nem lett vizsgálva ezen a platformon, és emellett az ezen található atmega128-as mikrokontroller rendelkezik is az ahhoz szükséges funkcióval, ami lehetővé teszi, hogy a mikrokontroller saját magát felprogramozza (lásd 3.1.3. fejezet), de a probléma még így is több rétű.

Egyrészt valahogyan létre kell hoznunk az új programunkat amit a szenzor egységünkre (továbbiakban csak mote) fogunk rátölteni, ezért valamilyen általunk kezelhető formátumban kell tárolnunk a legenerált programkódot. Mivel a MIT MÓT rendelkezik egy rádiós kártyával, és a dolgozat témája alapvetően egy vezetéknélküli hálózat felől megközelíteni a problémát, így ezen keresztül szeretnénk a programot átvinni. Ebből kifolyólag kell egy illesztő eszköz aminek a segítségével a program fejlesztéséhez használt PC és a felkonfigurálni kívánt mote között rádiós kapcsolatot tudunk teremteni. Ekkor felmerülhet bennünk az a kérdés, hogy pontosan milyen hálózatról beszélünk. Hány mote-ot tartalmaz? Milyen funkciót látnak el ezek a mote-ok? Hogy csatlakoznak egymáshoz? Természetesen az alkalmazás megvalósítása függ a hálózat struktúrájától, így ez esetenként más-és más.



2.1. Ábra A hálózat alkotóelemei

Ebben a dolgozatban én egy két mote-ból álló, pont-pont kapcsolatú hálózatra szorítkoztam, amely lehetőséget nyújt a feladat demonstrálására, illetve a fejlesztés során felmerülő problémák bemutatására. Ebben a pont-pont hálózatban master mote-nak hívjuk azt az egységet ami a PC-re csatlakozik és továbbítja a programkódot a konfigurálni kívánt mote-nak, és slave mote-nak hívjuk amit fel szeretnénk konfigurálni, vagyis ez fogadja a programkódot és a saját ún. flash memóriájába (lásd 3.1.2. fejezet) írja ahol is a végrehajtandó progamot tárolja. A master mote un. Universal asynchronous receiver/transmitter-en (továbbikban csak UART, lásd 3.1.6. fejezet) keresztül csatlakozik a PC-hez, és ezen keresztül történik a legenerált programkód átvitele a két egység között. Ezért az általam használt Windows 7 operációs rendszerre írtam egy kis alkamazást, ami kezeli ezt a kommunikációs formát, és átküldi az adatokat a master mote számára.

Sajnos a rendszer tulajdonságai nem teszik lehetővé, hogy a mote-unk folyamatosan fogadja az adatot, és közben a saját EEPROM memóriájába írja, illetve nem célszerű a PC-s alkalmazással addig várni, amíg végig megy az egész folyamaton egy adatcsomag, amit kiküldünk, ezért mindkét egységhez csatlakoztattam egy buffert

amiben a mote-ok el tudják tárolni a programkódot a folyamat egyik-és másik lépése között.

Ennek a buffernek több kritériumnak is meg kell feleljen, például elég nagynak kell, hogy legyen, hogy egy programkód elférjen rajta, mivel az atmega128-as flash memóriája (vagyis ahol programkódot tud tárolni) 128 kbyte így kézenfekvő volt, hogy ekkora buffert válasszak. Ezentúl permanensnek kell lennie, hogy egy esetleges újraindítás során sem vesszen el az adat. Erre a feladatra egy EEPROM memóriát választottam, amit I2C buszon (lásd 3.1.7. fejezet) keresztül csatlakoztattam a mikroprocesszorhoz (lásd 3.1.8. fejezet).

Miután az új programunk bele került a slave egységhez csatlakoztatott EEPROM memóriába, gondosan meg kell terveznünk az ún. bootloader folyamatot (lásd 3.1.3. fejezet), amikor is a mikroprocesszor flash memóriájának egy bizonyos szegmensében található függvények módosíthatják a flash többi részét. A feladat végrehajtása során ügyelnünk kell arra, hogy a programkódunkat maradéktalanul vigyük át, mivel a legkisebb hiba is könnyen a felhasználói program használhatatlanná tételét eredményezheti, ami pedig manuális beavatkozást igényelhet, és a célunk, hogy ezt elkerüljük.

Összefoglalva a program:

1. A legenerált programkódot a PC-ről UART-on keresztül a masternek küldi.
2. A master a fogadott adatot a külső EEPROM-ba írja.
3. A külső EEPROM-ból kiolvasott adatot rádión keresztül a slave egységnek küldi.
4. A slave a fogadott adatot a saját külső EEPROM-jába írja.
5. A slave bootloader alkalmazás a külső EEPROM-ban tárolt programkódot a saját flash memóriájába írja.

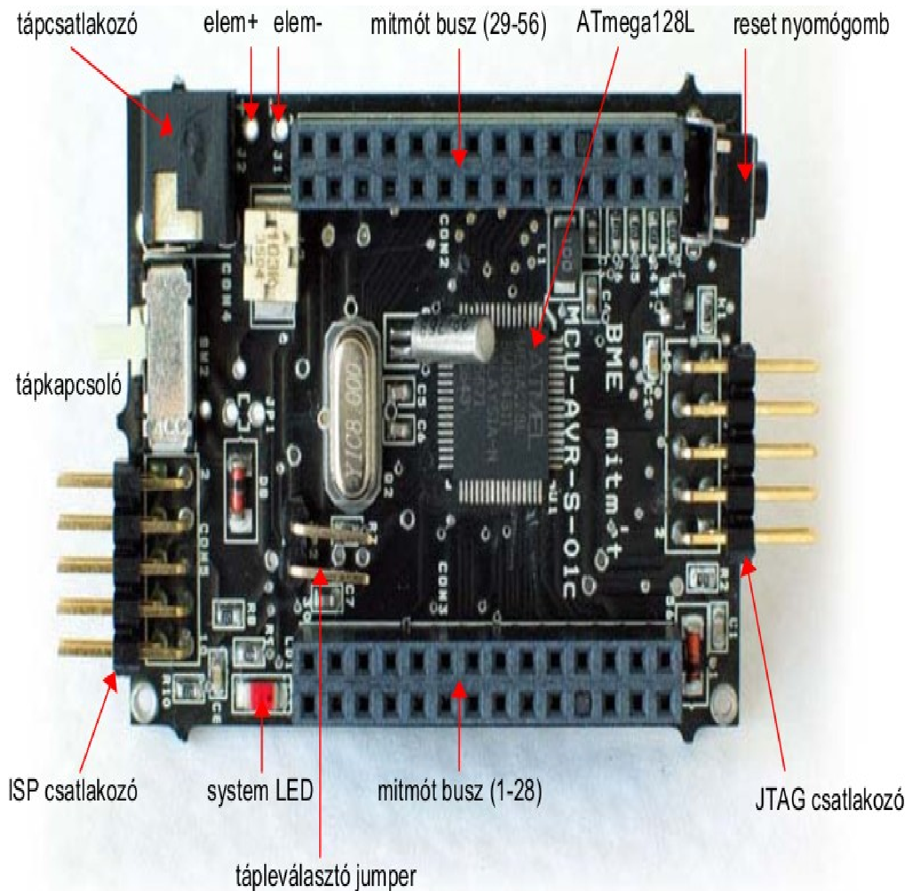
3 A hardveres és szoftveres környezet bemutatása

3.1 Hardver

A MIT MÓT a Budapesti Műszaki és Gazdaságtudományi Egyetem, Méréstechnika és Információs Rendszerek Tanszék által fejlesztett platformja aminek esődleges célja, hogy egy univerzális felületet nyújtson a diákok számára a beágyazott szoftverfejlesztés lépéseinek elsajátításához. A mitmót moduláris felépítésű kártyákból áll, melyeknek meg van a saját funkciójuk. Az általam használt kiépítésben a MIT MÓT a következő kártyákat tartalmazza:

- AVR mikrovezérlő kártya
- ISM sávú rádiós kártya
- DPY-LED periféria kártya

3.1.1 Az AVR mikrovezérlő kártya



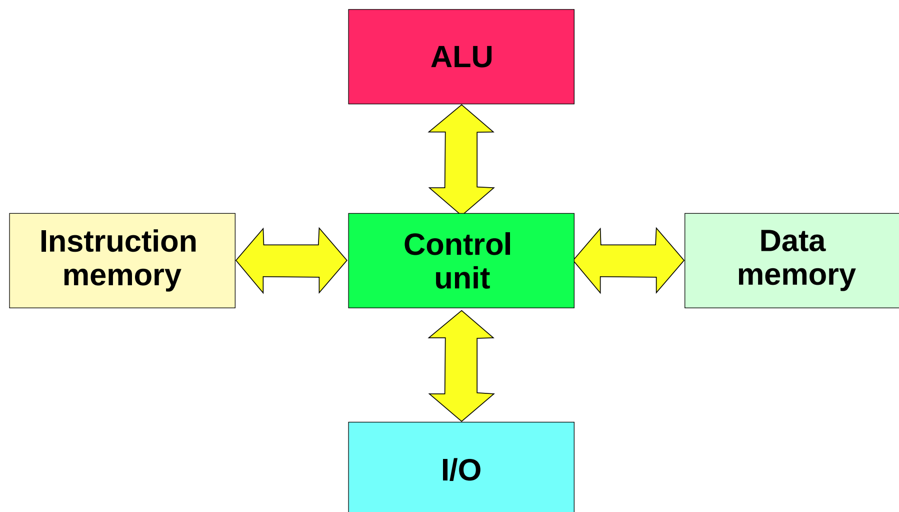
3.1. Ábra: AVR mikrovezérlő kártya [1]

A kártya központi egysége egy 8bites Atmel Atmega128L típusú mikroprocesszor, amely a következő fontosabb tulajdonságokkal rendelkezik: [1]

- 128 kbyte on-chip flash, opcionális boot code szekcióval, önprogramozási lehetőséggel (lásd 3.1.2. és 3.1.3. fejezet)
- 4 kbyte on-chip adat SRAM
- 4 kbtye on-chip EEPROM
- Vektoros interrupt, fix IT prioritások és vektorcímek, de a vektortábla átállítható a boot szekció elejére.
- JTAG támogatás (programozás, debuggolás, Boundary-scan)
- 2 db UART, SPI, I2C kommunikációs interfészek (lásd 3.1.6. és 3.1.7. fejezet)
- A kártya nem használja ki a mikrovezérlő összes lehetőségét.
- 16 bites címzés

3.1.2 Az atmega128 flash memóriájának felépítése

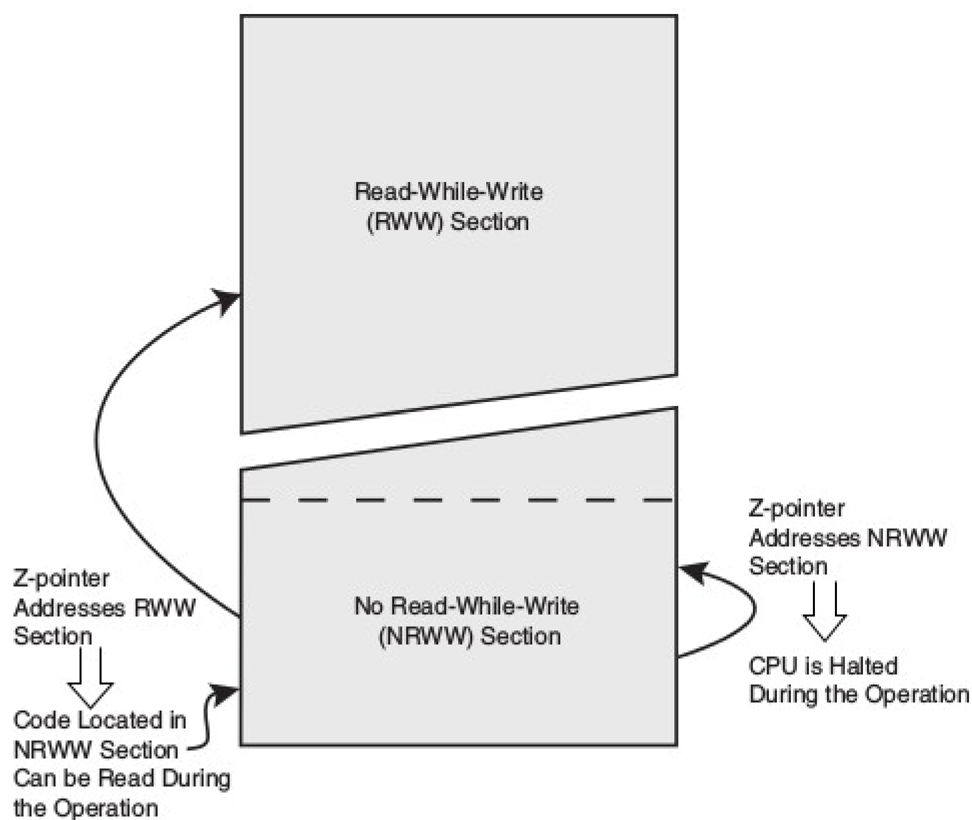
Az atmega128 a PC-kel elentétben nem Neuman architektúrájú, hanem Harward architektúrájú, vagyis a programkód és a program által felhasznált adatok, változók külön memóriában találhatóak. [2]



3.2. Ábra: Harward architektura [2]

Az atmega128 flash memóriája két részre oszlik az egyik az ún. Read-While-Write memória (továbbiakban RWW), ezt tudja a bootloader szoftver módosítani (lásd 3.1.3.fejezet) az újrakonfigurálás során. Illetve a másik a No Read-While-Write (továbbiakban NRWW), ezt csak a mikroprocesszor halt állapotában tudjuk módosítani. Vagyis:

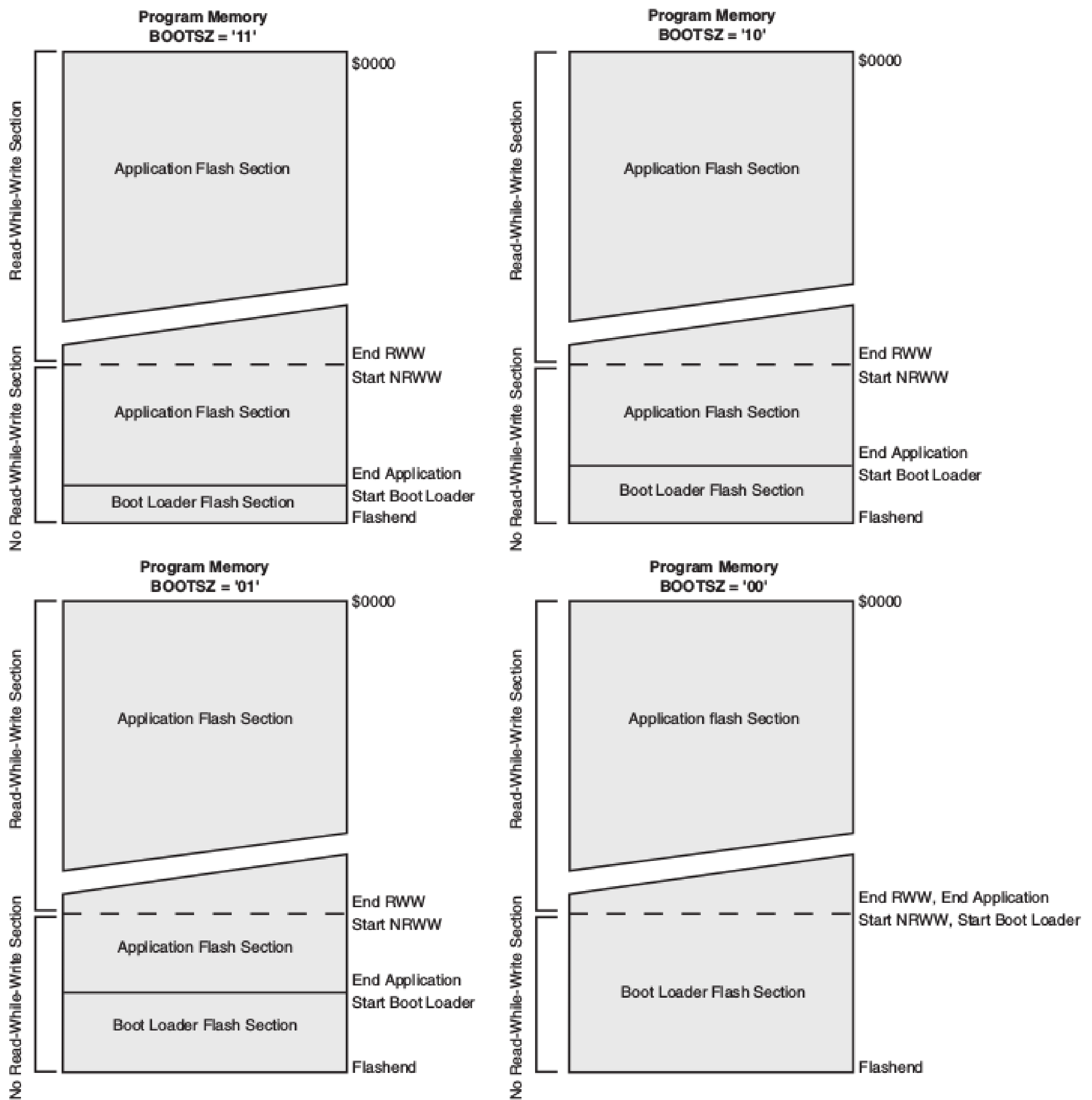
- A NRWW memória olvasható, miközben az RWW memóriát írjuk vagy törölük
- Mialatt a NRWW memóriát töröljük, a processzornak halt állapotban kell lennie
- A bootloader szoftver nem tud olyan programkódot olvasni ami az RWW memóriában található.
- A RWW memória azt a területet jelöli amit a bootloader szoftver felül ír, és nem azt amiből az új kódot másolja
- Az RWW memória mérete állandó, 124 kbyte, és a 0x00000-0x1F000 memória cím között található



3.3. Ábra: atmega128 flash memóriája [3]

- A NRWW memória mérete 4 kbyte, és ebben található az ún. bootloader section ami a bootloader funkciót látja el. Ebből látszik, hogy ennek mérete maximálisan 4 kbyte. Mérete a „BOOTSZ” fuse bitekkel állítható. Ezeket a hardver beállító biteket csak programozóval, a processzor halt állapotában tudjuk módosítani.
- Ha a bootloader méretét nem a maximális 4 kbyte-ra választjuk, akkor a fentmaradó helyen a NRWW-be, szintén tudunk programkódot helyezni, de módosítani csak a fent említettek szerint tudjuk.

A flash felépítése a BOOTSZ fuse bitek különböző értékei esetén:



3.4. Ábra: a BOOTSZ különböző beállításai [3]

Ekkor a section-ök mérete:

BOOTSZ1	BOOTSZ0	Boot Size	Pages	Application Flash Section	Boot Loader Flash Section	End Application section	Boot Reset Address (start Boot Loader Section)
1	1	512 words	4	$0000 - $FDFF	$FE00 - $FFFF	$FDFF	$FE00
1	0	1024 words	8	$0000 - $FBFF	$FC00 - $FFFF	$FBFF	$FC00
0	1	2048 words	16	$0000 - $F7FF	$F800 - $FFFF	$F7FF	$F800
0	0	4096 words	32	$0000 - $EFFF	$F000 - $FFFF	$EFFF	$F000

3.5. Ábra: a flash section-ök mérete, a BOOTSZ fuse bitek függvényében [3]

A flash memória page-kbe van rendezve melyek mérete 128 szó, ahol egy szó 2 byte hosszúságú, így 256 byte méretű egy page. A cím bitek a Z-pointer illetve a RAMPZ regiszterekbe vannak tárolva.

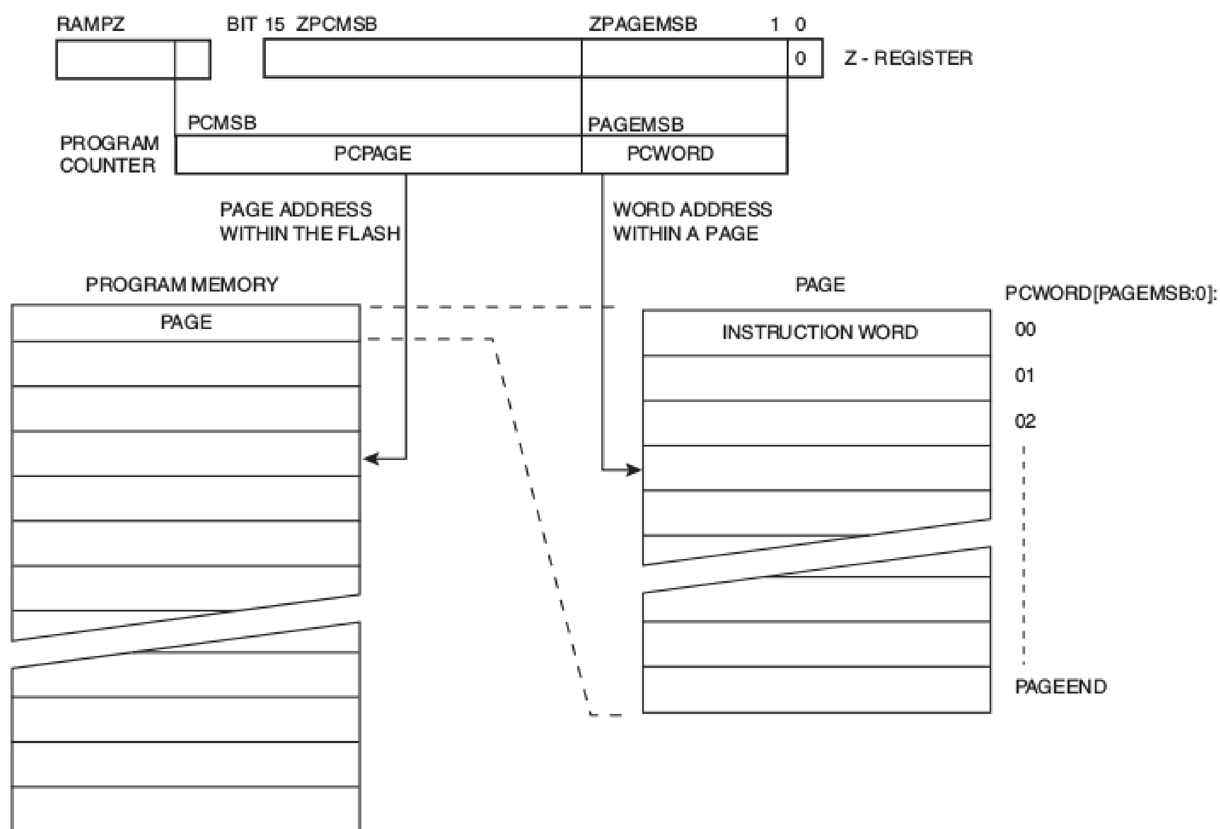
Bit	15	14	13	12	11	10	9	8
ZH (R31)	Z15	Z14	Z13	Z12	Z11	Z10	Z9	Z8
ZL (R30)	Z7	Z6	Z5	Z4	Z3	Z2	Z1	Z0
	7	6	5	4	3	2	1	0

3.6. Ábra: cím regiszterek[3]

Mivel a memória page-ekbe van rendezve, ezért a cím is két részből áll. Ahogy az alábbi ábrán is látszik a cím alsó 7 bitje a page-n belüli szót jelöli, míg a fentmaradó bitek a page-t a flash-en belül.

Flash Size	Page Size	PCWORD	No. of Pages	PCPAGE	PCMSB
64K words (128 Kbytes)	128 words	PC[6:0]	512	PC[15:7]	15

3.7. Ábra: page és szó címzés [3]



3.8. Ábra: page-k, és az azon belül található szavak címzése [3]

3.1.3 Az atmega128 bootloader funkciója

A mikroprocesszor bootloader funkciója lehetővé teszi, hogy a fent említett bootloader section-ben található alkalmazás programkódot írjon az RWW memóriába. Mivel ezt a memória területet szeretnénk írni, ezért nem olvashat a bootloader szoftverünk olyan prograkódot ami az RWW memóriában található, mivel ez a program nem várt működéséhez vezetne. Az RWW memória olvasásást a Store Program Memory Control and Status Register-ben (SPMCSR) található RWW Section Busy bit (RWWWSB), logikai 1-es értékre való beállításával tudjuk biztosítani. Miután a programozással végeztünk ezt logikai 0 szinte kell állítani, hogy újra használhatóvá váljon.

Ezen felül ajánlott az interrupt-okat is letiltani. Ezek olyan függvények amik egy előre meghatározott ún. trigger esemény bekövetkezésekor hívódnak meg. Ez azért fontos, nehogy valamilyen jump, vagy call utasítás következtében valamilyen ismeretlen állapotba kerüljön a mikroprocesszorunk.

Amennyiben nincs szükségünk erre a bootloader funkcióra, lehetőségünk van a

teljes flash memóriát az alkalmazásunkhoz felhasználni. Lehetőség van:

- Az egész flash memóriát védetté tenni a szoftver frissítésével szemben.
- Csak a bootloader section-t védetté tenni.
- Csak az alkalmazás flash-t védetté tenni.
- Engedélyezni az egész memória frissítését.

Ezeket a BLB01,BLB02,BLB11,BLB12 fuse bitekkel lehet beállítani. (Bővebben lásd: [3])

Lehetőség van ezen túl a reset esetén beálló programszámáló értékének beállítására. Ezt a BOOTRST fuse bit beállításával tudjuk megtenni. Amennyiben a fuse bit értéke programozatlan, egy esetleges reset esetény a programszámláló a flash memória elejére, a 0x0000-s címre ugrik. Ellenkező esetben pedig a BOOTSZ bitektől függő, bootloader section elejére.

A flash memória mivel ún. page-kbe van rendezve, ahogy az már előzőleg említésre került, ebből kifolyólag a cím két részből áll, és a Z-pointer, illetve a RAMPZ regiszter tárolja.

Az első fele, az MSB-től (Most Significant Bit) kezdődően, a page-et címzi meg, a cím második fele pedig a page-n belüli szót (1 szó= 2 byte), ahogy ez a 3.7. ábrán is látható. Az EEPROM-mal ellentétben, ahol byte-onként volt megcímezve a memória nincs szükség 17. címbitre.

Meg kell jegyeznünk, hogy a page törlés, és page írás egymástól függetlenül van címezve, ez fontos, hogy ugyan azt a page-t címezze meg a bootloader szoftver, de ha egy programozási folyamat elkezdődött, a Z-pointer/RAMPZ beolvasásra kerül, és onnantól más műveletekre is felhasználható. Egyedül a Boot Loader Lock bit-ek állításának nincs hatása a Z-pointer/RAMPZ-re.

Mivel a flash page-ekbe van rendezve, ezért csak page-enként tudjuk módosítani. Mielőtt egy page-be a flash memóriában adatot töltünk az átmeneti tárolóból (Temporary Buffer) törölnünk kell annak tartalmát. Az átmeneti tárolóba egyszerre egy memória írási műveletet hajthatunk végre, de ezt megtehetjük a page törlése előtt is. Két lehetséges módon vihető véghez a page írás.

1. Alternatíva:
    - átmeneti tároló feltöltése
    - page törlés végrehajtása
    - page írás végrehajtása

2. Alternatíva:
    - page törlés végrehajtása
    - átmeneti tároló feltöltése
    - page írás végrehajtása

Abban az esetben ha csak egy szó módosítása a cél, az első alternatívát kell használni. Ebben az esetben a page tartalma a törlés előtt betöltődik az átmeneti tárolóba, és a kívánt módosítások után vissza írható.[3]

3.1.4 ISM sávú rádiós kártya

A mitmóthoz tartozik egy rádiós kártya is, ami a rendszer egyszerű kommunikációs modulja, melynek alapja egy IA4420 típusszámú ráió adó-vevő. Ez egy könnyen illeszthető IC, amely a 433-868 MHz-es európai, illetve az Európán kívüli 315-915 MHz-es sávban használható. Az aktuális elemek 433 MHz-re lettek optimailizálva. Maximális sebessége ekkor 115,2 kbps. Ezen kívül a NYÁK kialakításakor pad-ek lettek kialakítva külső EEPROM csatlakoztatására. Erre mindkét móton forrasztottam egy Microchip 24FC1025 típusú EEPROM memóriát, ami I2C-n keresztül csatlakozik a microprocesszorhoz. Ezen felül egy szintillesztés nélküli UART csatlakozás is található a kártyán.

3.9. Ábra: rádiós kártya [4]

IA4420 legfontosabb tulajdonságai:[4]

- 0,6–115,2 kbps tartományban változtatható kommunikációs sebesség.
- Gyors frekvenciaugrási lehetőség.
- Pufferelt adatkommunikáció.
- Negyedhullámhosszú antennával hozzávetőlegesen 100–200 m- es távolság.
- Finoman hangolható vivőfrekvencia.

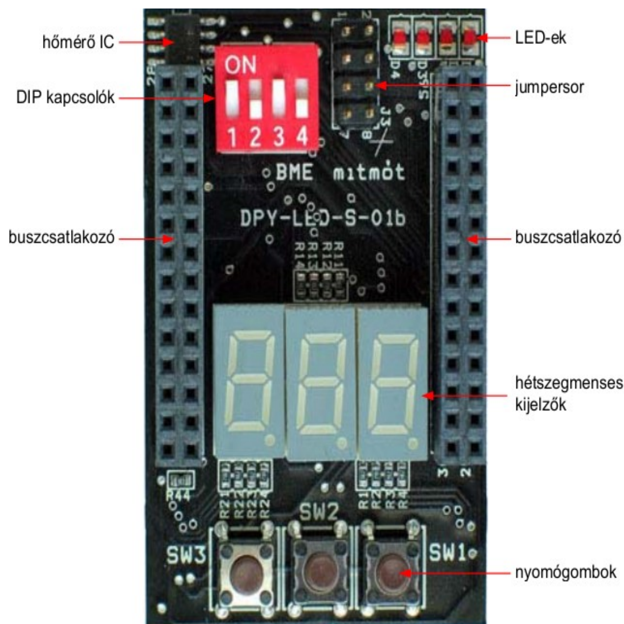
UART csatlakozás:

- RS-232/RS-485/RS-422 szintillesztési, illetve USB konverter lehetőség külső

modul segítségével

3.1.5 A DPY-LED perifériakártya

Az alkamazás fejlesztéséhez szorosan nem kapcsolódik a perifériakártya, de mégis a fejlesztés során is segítséget tud nyújtani, illetve szemléletesebbé teszi az alkamazás működését. Például az általam készített próba alkalmazásban a master mote megjeleníti a hétszegmensű kijelzőkön a slave egységtől lekérdezett kapcsolók álllását, de ez is csak bemutató szerepet tölt be.



3.10. Ábra: DPY-LED perifériakártya [5]

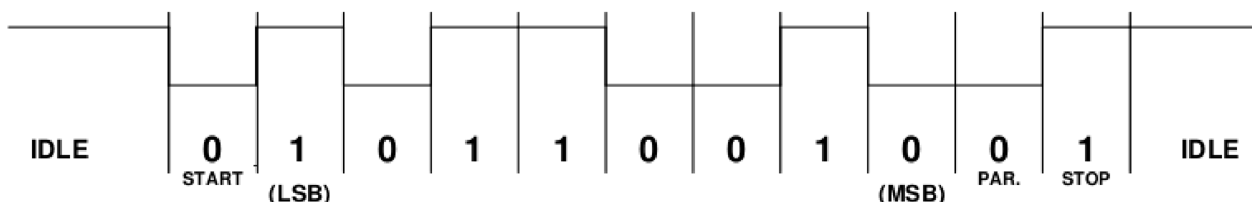
3.1.6 UART bemutatása

Az UART vagyis Universal asynchronous receiver/transmitter, vagyis Univerzális Aszinkron Soros Adó/Vevő, melynél ahogy a neve is sugalja az adatátvitel egy irányba mindössze egy vezetéken történik, pont- pont kapcsolattal az egységek között. Főleg a távközlésben használatos, mivel kicsi a vezetékigénye, de egyszerűsége és rugalmassága miatt a beágyazott rendszerekben is előszeretettel használják.

Az UART adatávitel keretekben értelmezett, vagyis minden üzenet csomag egy előre definiált formában érkezik, és általában egy karaktert tartalmaz. Egy keret a következő elemeket tartalmazza:

- 1 START bit, ez egy logikai 0-s értéknek felel meg
- 5-9 bit között előre definiált számú adat bit

- Paritásbit, ez lehet páros, páratlan, esetleg mindig 1, vagy mindig 0
- STOP bit, ami lehet 1, 1.5, 2

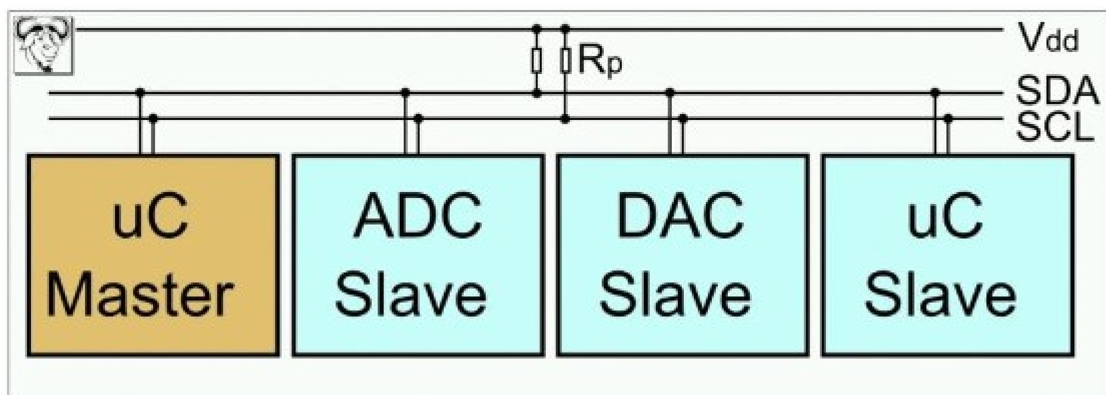


3.11. Ábra: Az UART adatáviteli keret felépítése [6]

A START, STOP, PARITÁS bitek szinkronizációs és hibaellenőrzéső szerepet töltenek be. Ezek beállításait a kapcsolat inicializálásakor kell megadni csak úgymint a sebességet amit általában baudrate-ben adnak meg (szimbólum/másodperc). Ez a szabványban definiált lépésekben szabad választható, 110 baud/s-től akár megabaud/s -ig terjedhet. Jellemzően 115200 baud/s körüli értéket szokott felvenni. Mivel a fogadott jelet a fogadó a saját órajele szerint mintavételezi, így nem igényel dedikált órajelet, ezt általában 16x-osan szokta a baudrate-hez képest. Azonban érzékeny az egységek belső órajele közötti hibára, ez nem lehet nagyobb 1%-nál. Az UART PC-s beállításait lásd a 5.1.3. fejezetben.[6]

3.1.7 I2C bemutatása

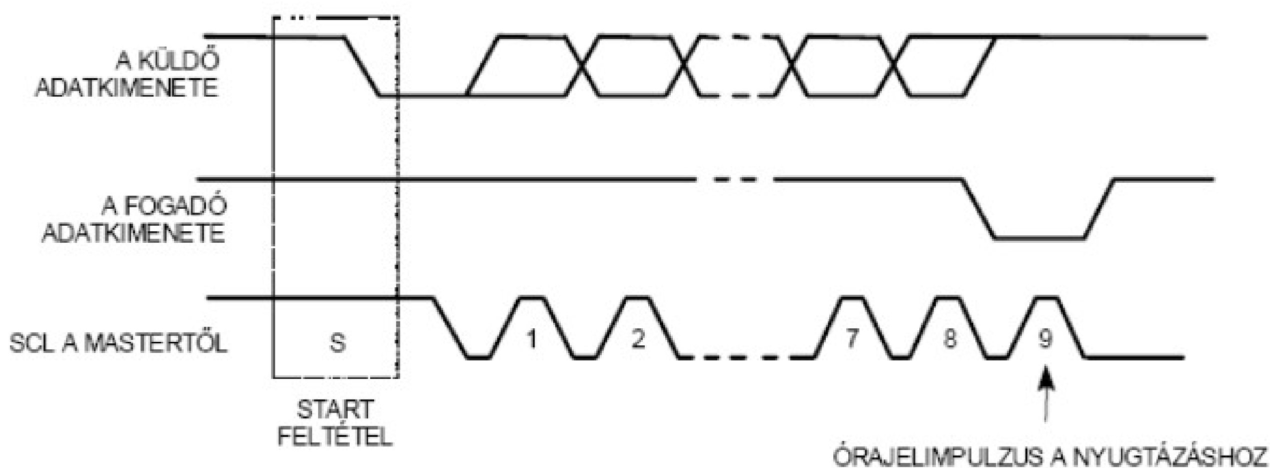
Az I2C vagyis Inter-Integrated Circuit egy kétirányú, szinkron, félduplex adat interfész, vagyis két irányba folyik az adatátvitel, azonban egyszerre csak egy irányba, de ez akár buszra csatlakoztatva is megvalósítható, amin egyszerre maximum 128 különböző egységet lehet megcímezni. Az I2C két vezetéket igényel, az egyik az adatvezeték (SDA), a másik busz órajel (SCL). Alapvetően kis mennyiségű adat átvitelére tervezték, mivel a busz órajele néhány 10 kHz-től néhány 100 kHz-ig terjed, így az adat átvitel sebessége is néhány 10 kbit/s körüli. A topológiában megkülönböztetünk master, illetve slave egységeket. A master vezérli a kommunikáció keretrendszerét, (kezdet vég), és ő szolgáltatja az órajelet is.



3.12. Ábra: I2C kapcsolat felépítése [7]

A mi esetünkben mindig az atmega 128-as a master, és a külső EEPROM chip a slave (lásd 3.1.8. fejezet), bár a rendszer lehetővé tenne, hogy egyszerre több master legyen jelen a buszon, ezt hívják multimaster-nek.

Értelemszerűem az adatávitel során mind a master, mind a slave lehet fogadó, illetve küldő. Gondoljunk csak bele, nem csak írni szeretnénk az EEPROM-ba, hanem olvasni is belőle, de az órajelet minden esetben a master hajtja meg. Ha a slave esetleg nincs kész az adatátvitelre, akkor lenttarthatja az órajelet, és ezzel várakozásra bírhatja a master-t. Az adatvezetéket (SDA) mindig az éppen küldő egység hajta meg, és rakja ki rá az átküldendő adatot. Ha az adatátvitel sikeres volt, a fogadó egység egy nyugtázó bittel (ACK) jelzi ezt a küldő felé.



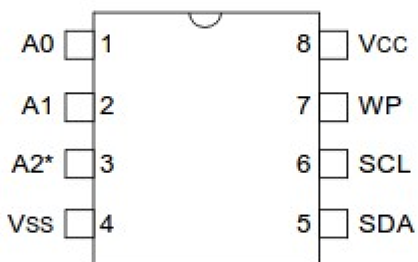
3.13. Ábra: Egy adatbájt átvitele az I2C buszon[7]

Az I2C buszon, csak úgy mint az UART esetében keretekben (frame) történik, amit a START (S), illetve STOP (P) bitek határolnak. Ezek specialitása, hogy ezek alatt az SDA adatvonal az SCL magas értéke alatt vált állapotot, egyéb esetben az SDA adatvonal az órajel magas értéke alatt stabil. Míg új értéke az SCL alacsony állapota alatt állítja be. [7]

Az atmega128 gyárilag rendelkezik az I2C kezeléséhez szükséges interface-szel, erre a dokumentációjában Two-wire Serial Interface-ként hivatkozik.

3.1.8 Külső EEPROM chip bemutatása

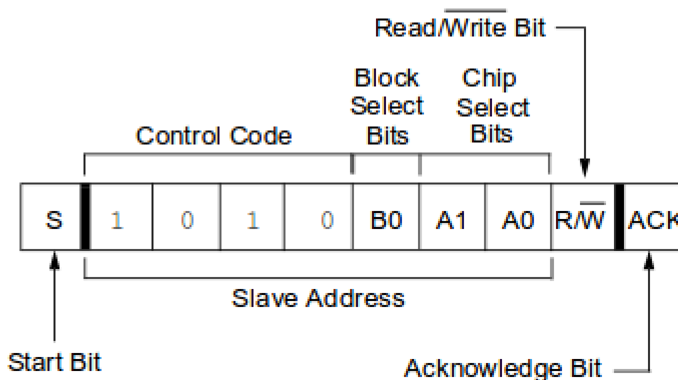
A MITMÓT fejlesztői a mikrokontroller kártyára kialakítottak úgynevezett PAD-eket, amin keresztül csatlakoztatni lehet egy Microchip 24FCxxxx típusú EEPROM memóriát. Mivel mi atmega128-ast használunk amelynek programmemóriája 128 kbyte méretű, így kézenfekvbő volt, hogy ebből az elérhető legnagyobbat válasszam, a 24FC1025-öst aminek mérete 128 kbyte.



3.14. Ábra: 24FC1025-ös lábkioszása [8]

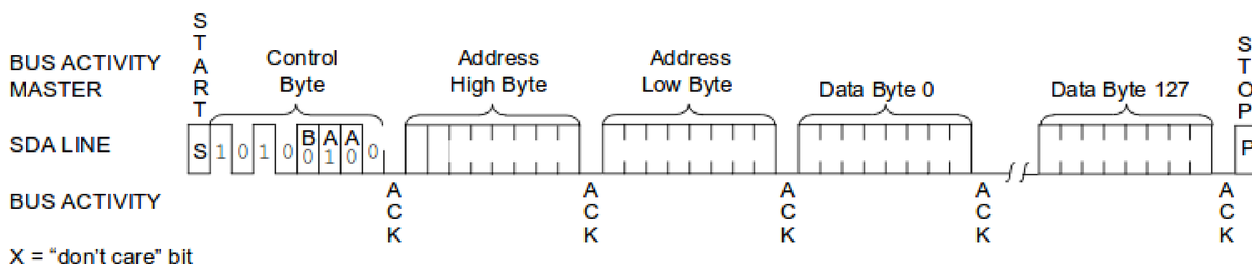
A 24FC1025 memóriája 128 byte-os pagekbe van rendezve, amiből ebből kifolyólag 1024 darab található benne. A chippet I2C-n keresztül tudjuk elérni, amihez az atmega128 rendelkezik a szükséges port-tal. Az I2C buszon egyszerre négy ugyanolyan típusú EEPROM-ot tudunk megcímezni, erre szolgál a 3.14. ábrán látható A0, A1 láb, illetve az A2 aminek minden esetben logikai 1-es szinten kell lennie, ha használni szerernénk a chippet.

A 24FC1025 memóriája két részre van osztva, egy 0x0000-0xFFFF-es tartományra, illetve egy 0x10000-0x1FFFF-es tartományra. Ezek logikailag elkülönülnek egymástól így külön kell őket megcímezni. Az I2C buszon a (3.15. ábra) szerint történik a



3.15. Ábra: I2C buszon történő címzés [8]

címzés. Az EEPROM chip A0,A1,A2 lábat rendre logikai 1-es szintre kötöttem egy ellenálláson keresztül, és így az EEPROM alsó szegmensét a 0x53-as control byte-al tudom megcímezni, a felső szegmensét pedig a 0x57-es control byte-al. A címzéskor meg kell adnunk, hogy írást vagy olvasást szeretnénk kezdeményezni.



3.16. Ábra: 24FC1025 írási művelete [8]

A 24FC1025-ös támogatja a szekvenciális írást, amivel maximálisan egy page-be írhatunk egyszerre, utána újra kell kezdenünk az írási folyamatot. Olvasásnál hasonló képpen, csak a két cím byte-ot írás kezdéssel kell kezdenünk, utána pedig egy olvasást kell kezdenünk. Az utolsó olvasott byte után pedig meg kell mondanunk, hogy nem várunk visszajelzésre az EEPROM-tól. [8]

3.2 Szoftver környezet

Az almazás megírásához az Atmel nyújtotta AVR studio 4.19-et illetve a szintén az Atmel által kiadott AVR stúdió 6.2-t, amelyek az AVR típusú mikrokontrollerek hivatalos, a gyártó által is ajánlott fejlesztőkörnyezete, amely egyébként ingyenesen elérhető a gyártó honlapján. Ez hivatalosan csak Windows operációs rendszer alatt támogatott.

A program fordításához a WinAVR 20100110 számú verzióját használtam, amely egy gcc alapú nyilt hozzáférésű fordító program, ami a GNU projekt részét képezi. Ez széles körben használt, nem csak beágyazott rendszerek fejlesztéséhez. A fordító használatához a fejlesztők a rendelkezésünkre bocsájtanak egy kézikönyvet, ami tartalmazza a fordítóban használt direktívákat, jelöléseket a memória felépítésére, illetve fordítás menetéről adnak tájékoztatást. Később erre konkrét példát is látni fogunk.

Szintén a WinAVR tartalmaz nagyon sok, a fejlesztők által a rendelkezésünkre bocsájtott függvénykönyvtárat ami nagyban elősegíti a fejlesztést, mivel a

legalapvetőbb funkciókat ellátó függvények többnyire már implementálva vannak ezekben a könyvtárakban.

A mikrokontrollerre való program feltöltéshez a HappyJTAG nevezetű programozót használtam, ami a mikrokontroller JTAG portján keresztül lehetőséget nyújt az eszköz programozására és debugolására. Ehez tartozik egy PC-s alkalmazás amelynek én a v2.45-ös verzióját használtam. Ez létrehoz a Windows-ban két virtuális soros portot, és ezen keresztül égeti fel a programunkat a mikroprocesszorra, de a segítségével akár ki is olvashatjuk onnan. Ezen túl lehetőséget nyújt a mikroprocesszor fuse bitjeinek beállítására is.[9]

Az AVR studioban is lehetőség nyílik arra, hogy ha a HappyJTAG be van kapcsolva, akkor a fejlesztőkörnyezetből csatlakozzunk közvetlenül a mikroprocesszorra, és onnan kényelmesen írjunk bele, és olvassunk belőle. A flash memória kiolvasás nagy segítséget nyújthat a hibakeresésre, illetve egyfajta visszacsatolást nyújt a fejlesztő számára.

Az alkamazásban néhol az egységek közti kommunikáció UART-on keresztül valósul meg, ennek a teszteléséhez a HTerm nevezetű virtuális terminál programot használtam. Ezen felül az UART használata a fejlesztés során is komoly segítséget tud nyújtani a fejlesztő számára, mivel kapcsolatot tud létrehozni a mikroprocesszor és a PC között, és még ha korlátozottan is, de bepillantást nyerhet a a processzor belső állapotaiba, a program futásába.

A PC oldali klienst C programozási nyelvben írtam, a CodeBlock nevezetű fejlesztői környezetben, szintén gcc fordítóval, illetve Microsoft Windows operációs rendszeren, ami annyiból lényeges, hogy a Microsoft a rendelkezésünkre bocsájt egy C függvénykönyvtárat ami tartalmazza a szükséges függvényeket a PC soros portjának az eléréséhez, amivel máris egy rutin feladattá válik a kliens megvalósítása.

A legenerált programkódomat intel HEX fájlokban tároltam, ami a fordító már a felhasználó számára is használható kimenete, ebben minden érték hexadecimális formátumban található. Később ezt a .hex kiterjesztésű fájlt töltjük fel JTAG-en keresztül a mikrokontrollerünkre. A program tervezése során törekedtem arra, hogy később a programkód az átvitele során megmaradjon az elején létrejött formájában.

Az intel HEX fájl minden sora egy előre meghatározott struktúra szerint épül fel:

1. Start kód:ez egy kettőspont
2. Bájtok száma:megadja ,hogy a sorban hány adat bájt található, ez maximálisan 16

byte

3. Cím: a következő adatbájtok milyen címtől kezdődően helyezkednek el, maximálisan 2 byte.
4. Adat típus, ez lehet:
   - adat:0x00
   - fájl vége:0x01
   - kiterjesztett segmens cím:0x02
   - szegmens kezdete cím:0x03
   - kiterjesztett lineáris cím. 0x04A mi hex fájlunkban is 64 kbyte után található egy ilyen.
   - Lineáris cím kezdés 0x05
5. Adat
6. Ellenörző összeg

Példa az intel hex fájlra

:10010000214601360121470136007EFE09D2190140

:100110002146017E17C20001FF5F16002148011928

:10012000194E79234623965778239EDA3F01B2CAA7

:100130003F0156702B5E712B722B732146013421C7

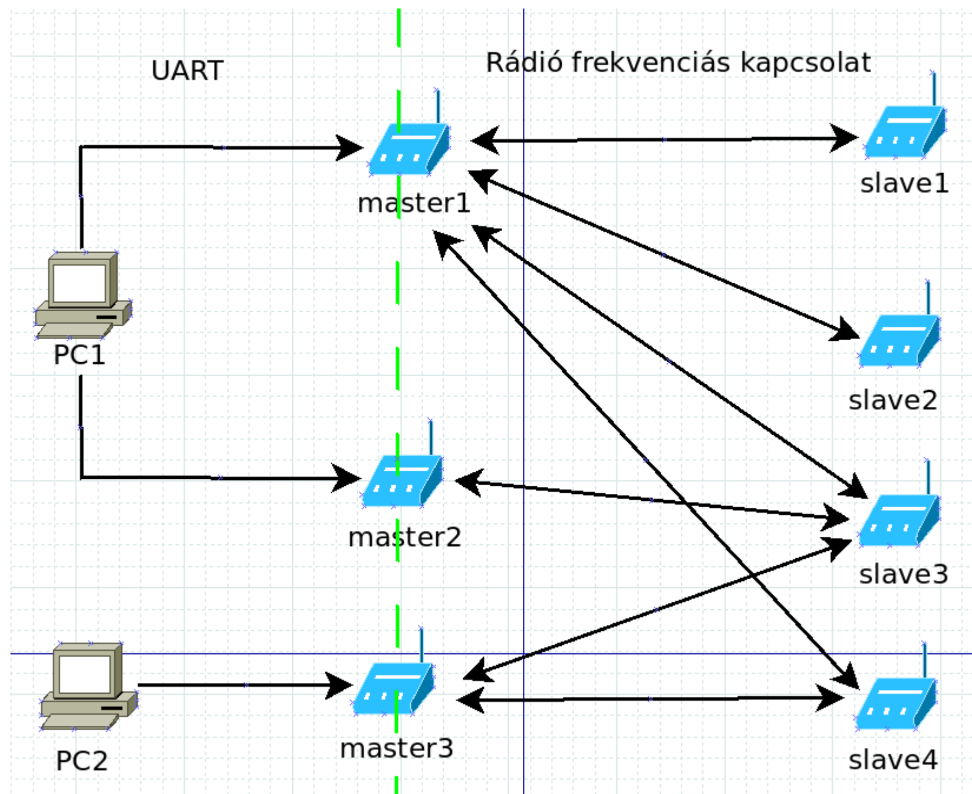
:00000001FF

Később JTAG-en keresztül, a processzor halt állapotában ki is tudjuk olvasni a mikrokontroller flash memóriájában található HEX fájlt, ahol azokon a helyeket ahova mi nem helyeztünk programkódot, a mikrokontroller 0xFF-ekkel tölti fel. A kód kiolvasása egy jó vissza csatoltált tud nyújtani a fejlesztő számára, a flash memória állapotáról.

4 Az alkalmazás strukturális áttekintése

4.1 Topológia

Az alkalmazás alapvetését a szenzorhálózatok adták, amiben sokszor problémás a már használatban lévő szenzorokon futó program utólagos módosítása. Ebben a



4.1. Ábra: egy lehetséges elrendezés

dolgozatban, én most csak egy pont-pont kapcsolatú master-slave hálózatban mutatom be az eljárás menetét, de ez a hálózat könnyen bővíthető további slave, vagy további master egységekkel, mivel minden egység rendelkezik az saját azonosítóval és csak azokra válaszol a megcímzett slave, amiben a saját azonosítója szerepel. Persze ennek a kialakítása nagyban függ a megvalósítandó feladattól, és mivel ennek a dolgozatnak a célja a módszer bemutatása így ebben a dolgozatban csak egy master-slave kapcsolatra szorítkoztam az egyszerűség kedvéért, de a később kitérek a bővités lehetőségeire is.

4.2 Vezetéknélküli kommunikáció protokolja

A vezetéknélküli kommunikáció a BME-MIT álltal fejlesztett, és feljebb már tárgyalt ISM sávú rádiós kártyán keresztül történik. Ehhez a fejlesztők a rendelkezésünkre bocsájtottak a függvényeket amivel már könnyen egy egyszerűbb adatvitelt tudunk megvalósítani.

4.2.1 Adatstruktúra és üzenetformátum áttekintése

A rádiós interfészen keresztül történő adatátvitel egységessé tételét és jobb tervezhetőségét hivatott szolgálni az a megoldás amit már az UART, illetve I2C protokolnál megismerhettünk, vagyis az, hogy az üzeneteink egy úgynevezett keretbe vannak foglalva. Ez csupán annyit jelent, hogy minden üzenet hasonló hosszúságú, és közel azonos részekből áll. Mivel a C programozási nyelv lehetőséget nyújt a struktúra által lefoglalt memóriaterület tetszőleges felbontására, így az előre definiált üzenet keret módosítható.

sDataStruct		
Típusa	Neve	Célja
uint8_t	moteID	A slave mote saját ID-ja
uint8_t	msgID	Az üzenet csomag ID-ja
uint8_t	data_field[16]	Itt lesz az átvitt programkód
uint16_t	address	A 16 byte-os tömb címe

- moteID: Minden motenak saját ID-ja van ami lehetővé teszi, hogy több slave esetén mindegyik mote csak a neki címzett csomagokkal foglalkozzon, a többbit hagyja figyelmen kívül.
- msgID: Minden üzenetnek is saját ID-ja van, mivel amikor a slave visszajelzést küld a masternek az adat fogadásáról, akkor időzítési problémák miatt könnyen összekavarodhatnak az üzenetek, és ez veszélyezteti az adat biztonságos átvitelét.
- data_field[16]: Mivel az atmega128 byte-onként van címezve, és az intel hex fájlban soronként maximum 16 byte található így kézenfekvő volt ekkora csomagokban átküldeni az adatot. Ekkor sor címe is adott az intel hex fájléból.

- Address: A továbbított adat címét adja meg a memóriában, illetve a csomag jellegét is hivatott jelezni.

Ezen túl a MITMÓT hivatalos, fejlesztői által kiadott rádiós függvények tartalmaznak egy CRC kód képzőt, ami biztosítja, hogy az adat fogadás csak akkor legyen sikeresként megjelölve, ha ténylegesen azt a csomagot fogadtuk, amit elküldtünk.

4.2.2 A cím mező kezelése

Az üzenet struktúrban a cím mező kezelése több megfontolást is igényelt. Mivel az atmega128 128 kbyte-os flash memóriája 0x1FFFF-ig van címezve, a Non-Read While Write section (lásd 3.1.2. fejezet) pedig 0x1F000-nál kézdődik, ide nem tudunk futás közben programkódot írni, csak ha a processzor halt állapotban van. Ebből látszik, hogy bizonyos címek nem fordulhatnak elő, ugyanis oda nem tudnánk adatot írni, és az ilyen címekre eső adatok továbbítása már a PC oldalán ki lettek szűrve, így felhasználhatjuk ezeket a címeket arra, hogy másfajta adat csomagkat is definiáljunk, amik rendszerüzenet szerepét töltik be, vagy pedig az felhasználói programban játszanak szerepet.

Az is jól látszik, hogy 0x1FFFF 17 biten fér csak ki, vagyis a struktúrában definiált 16 bites címváltozó kevés lenne, de ha jobban ismerjük az atmega 128-as működését akkor észre vehetjük, hogy 16 bites utasításai vannak (2 byte-os a szóhossz), így a legenerált hex fájlban nem fordulhat elő olyan sor amiben páratlan számú kód byte lenne, vagyis az LSB nem hordoz számunkra hasznos információt. Íme egy rövid részlet egy HEX fájlból:

```
:103480008FF706E69DD414C5ABB122A0B992308364
:1034900000C77B4E6AD5585C49E33D6A2CF11E780F14
:1034A00028F91C58C94A0AF80848FFDFF728F91C10
:0E34B00058C94A0AF80848880B2E190E8E00DB
```

Mint ismeretes az intel hex fájl első byte-ja megadja az adott sorban lévő kód byte-ok számát, míg az utána következő két byte pedig a címét a flash memóriában, az utána található egy byte-on pedig az adat típusa található. Vagyis mivel a cím a HEX fájlban is csak két byte-on van tárolva így látszik, hogy nem férne el rajta mind a 17 bit. Ezért a

HEX fájlban található egy sor aminek a típusa Extended Segment Address, és típusa 0x02. Ez alapján a hex fájl olvasásakor tudjuk, hogy mikor vált a 16. bit nulláról egyre, és mivel az LSB úgysem hordoz ebben az esetben számunra hasznos információt, kézenfekvő eggyel jobbra shiftelni a címet.

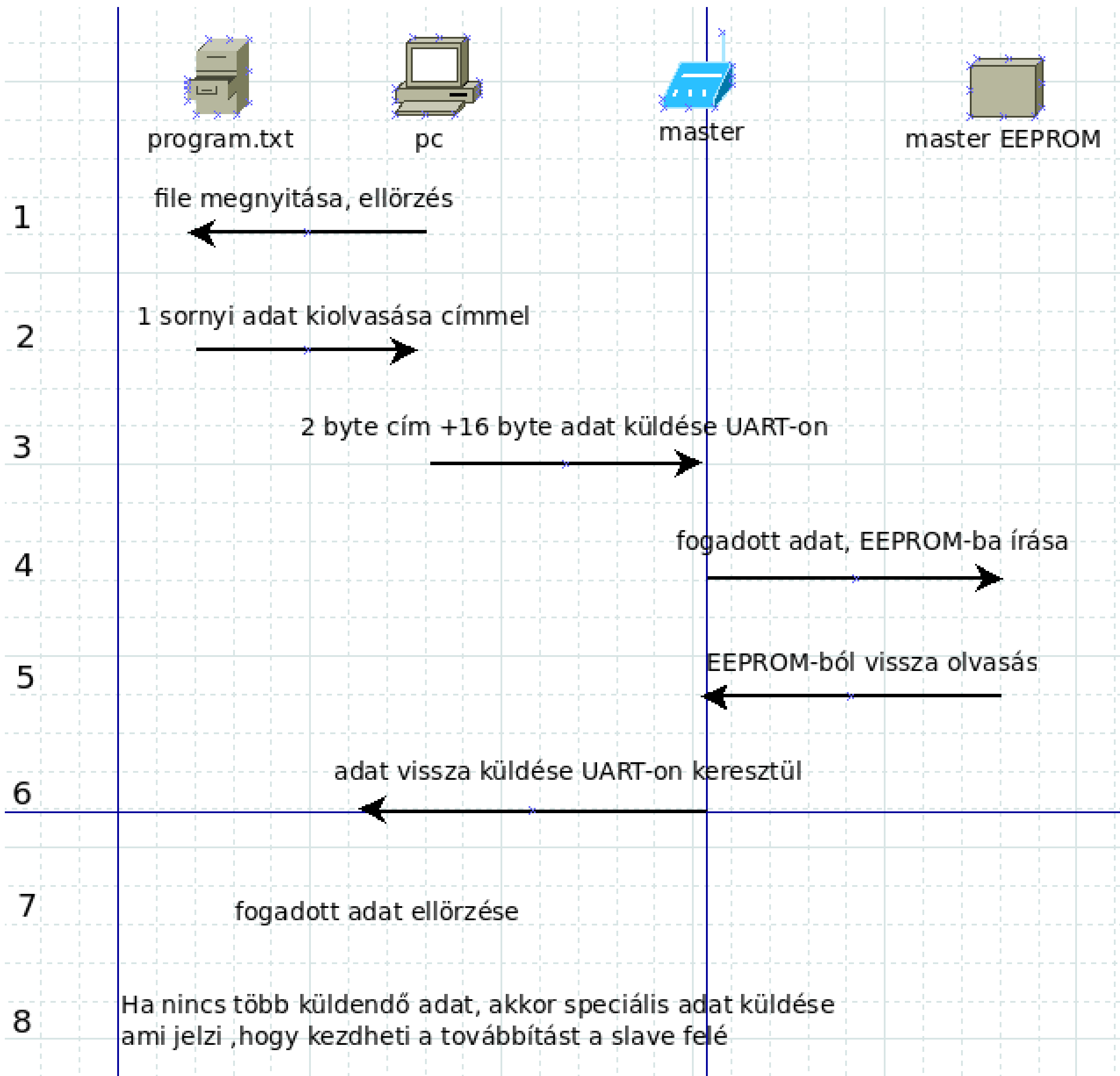
Ekkor a Non-Read While Write memória címe vagyis az itt előfordulható cím határa 0x1F000-ről 0xF800-ra vált, és marad 0x800 helyünk ahova nem kerülhet kód byte. Ez a kommunikációs protokolban szűrve van, vagyis ha olyan csomag jön aminek az address mezőjében 0xF800-nál kisebb érték van, akkor az kód csomag lesz, és menti az EEPROM-ba, míg ha valami más akkor az már a definiált programtól függ. Az én példa kódomban az address=0xFB00 jelenti, hogy a mote állapotát szeretnénk lekérdezni, vagyis ez a felhasználói alkalmazáshoz kapcsolódik, míg acknowledge üzenet esetén address=0xFB00, és ha azt szeretnénk jelezni, hogy nem küldünk több kódcsomagot és a mote kezdheti az újra programozást, akkor az address=0xFD00.

Ha esetleg egy sokoldalúbb üzenetstruktúrát szeretnénk alkalmazni, akkor megtehetjük, hogy programban lefoglalt struktúra memóriájának a helyére egy vele megegyező méretű valamilyen más struktúrát írjunk, és ezt továbítsuk a rádión keresztül. A megkötés egyedül az address változóra vonatkozik mivel az alapján szűrjük a csomagokat a vevő oldalon. Az intel HEX fájlról bővebben, a 3.2. fejezet utolsó bekezdésében olvashatunk.

4.3 A program végrehajtásának rövid áttekintése

4.3.1 PC-master adat átvitel

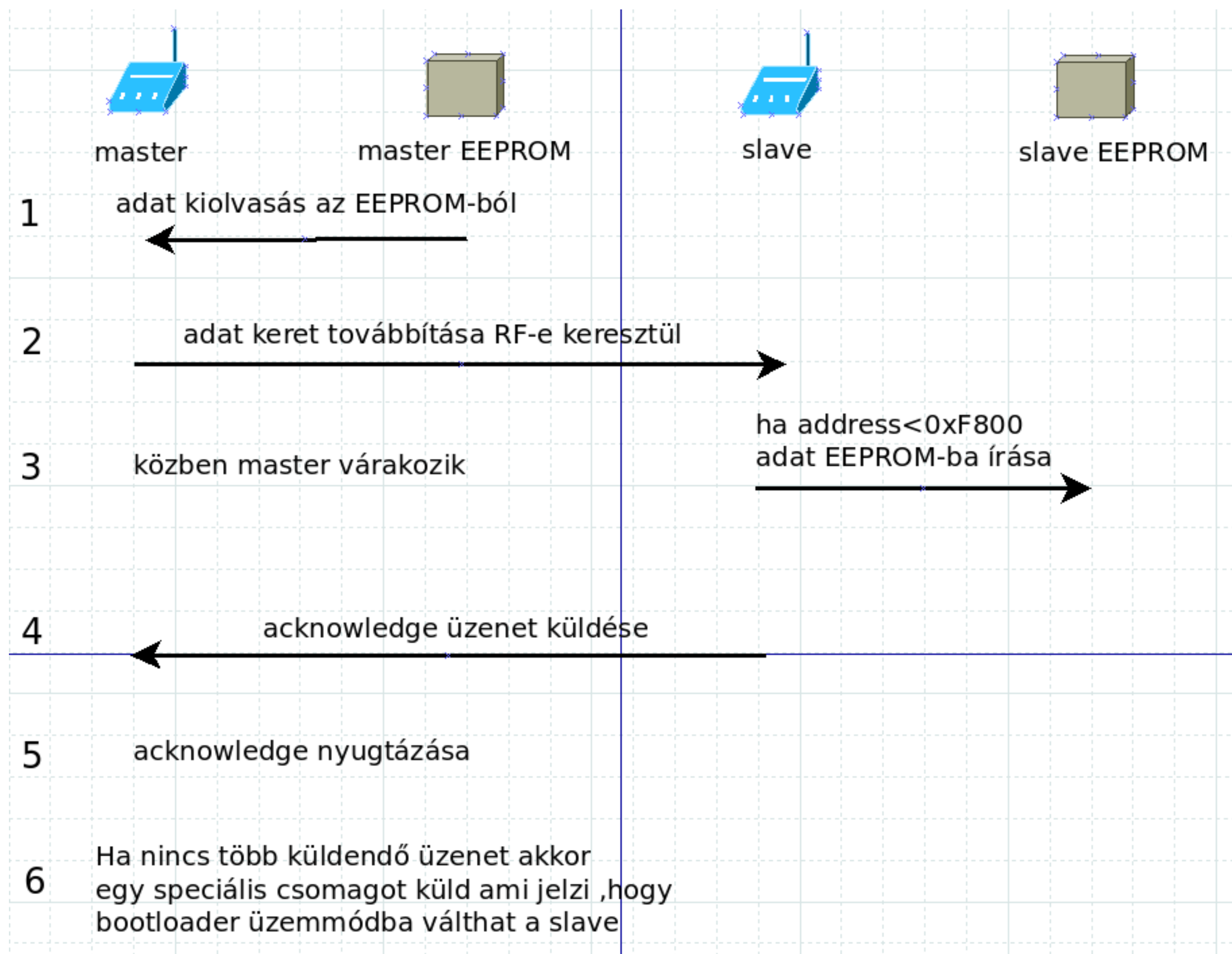
A PC és a master közötti kommunikációt a 4.2. ábra mutatja, részletesen pedig a következő módon jellemezhető:



4.2. Ábra: PC-master mote adatátvitel

- Ha a PC oldali programnak nem sikerül megnyitni a fájlt vagy az UART-ot inicializálnia hibaüzenetet dob, és leáll.
- A PC oldali program lefutás után kiírja, hogy hány hibás csomagok kapott vissza, és csak akkor ad parancsot a masternek az adat továbbításra a slave felé, ha ez nulla.
- A master oldali UART-on történő adat fogadást egy interrupt aktiválja, így ez független a mote-on éppen futó programtól.
- Amikor a master adatot ír az EEPROM memóriájába, az 0x1F000-as cím után elkezd egy memóriatérképet vezetni, ahol szekvenciálisan minden bit egy 16 byte-os cellát jelöl, aminek értéke 1 ha írtunk abba a cellába. Vagyis ha a 0x1F000-as címen található byte értéke 0x82=10000010 akkor a 0x00000 és 0x00060 címen található 16 byte-os blokkban van adat.

4.3.2 Master slave adat átvitel



4.3. Ábra master-slave RF kommunikáció

- A master a memóriatérkép alapján kiolvassa azokat a memóriarekeszeket ahova korábban írt adatot. Ebből adatstruktúrát készít, amihez generál üzenetazonosítót, bele írja a címzett slave mote ID-jét, illetve az adat címét. (lásd 4.2.2. fejezet)
- A master miután elkülde a slave-nek rádión keresztül az adatot, vár az acknowledge üzenetre.
- Ez alatt a slave beírja az EEPROM memóriájába, de ő már nem generál memóriatérképet, mivel csak a rádión keresztüli adatávitellel érdemes spórolni, mivel a mote-on belüli memória műveletek viszonylag gyorsak.
- A slave bizonyos időközönként hallgatja csak a rádiós csatortnát, mivel ez

blokkolja a program végrehajtást. Csak azokkal a fogadott csomagokkal foglalkozik ami tartalmazza a saját ID-ját, a többit eldobja.

- Ha a master nem kap egy elküldött csomagra választ, akkor azt újra küldi, így biztosítva azt, hogy minden csomag maradéktalanul megérkezzen.
- A master külső EEPROM-ját töröljük minden alkommal, amikor sikeresen átvitte az adatot a slave-nek, nehogy a következő adatküldés során benne maradjon, és azt is továbbítsuk.

4.3.3 A program írása a flashbe

A bootloader section az én alkalmazásomban a 0x1F000 címtől kezdődik, itt található minden olyan függvény, ami szükséges a külső EEPROM eléréséhez, illetve a programkód flashbe írásához. A flash-ből 256 byte-os pagenként (ami az atmega128 flash memóriájának page mérete) szekvenciálisan kiolvassuk az adatot az EEPROM memóriából, és beírjuk a page bufferbe, a buffer tartalmát pedig az flash memóriába. Mivel az EEPROM 128 byte-os page-ekbe van rendezve, így abból két page-t kell kiolvasni a buffer feltöltéséhez. Ez idő alatt, mivel flash memóriát írjuk, nem hívhatunk meg olyan függvényt ami ott található.

Ha az EEPROM teljes tartalmát (0x0000-tól 0x1F000-ig) bele írtuk a flash-be egy watch dogos restart-tal a 0x0000 címre ugrunk és ezzel el is indul az ott található programunk. Mivel automatikusan az egész külső EEPROM-ot a flash-be másoljuk így, ajánlott újraindítás után törölni a tartalmát, nehogy valami programkód benne maradjon, és a következő alkalommal azt is bemásoljuk a flash-be, és ezzel tönkre tegyük a programunkat.

5 A modulok implementálása

Ebben a fejezetben egyenként részletes bemutatásra kerülnek a hálózatban szereplő egységek.

5.1 PC oldali szoftver

5.1.1 Felhasznált könyvtárak

A PC oldali programhoz az alábbi függvénykönyvtárakat használtam fel:

- stdio.h: Standard-Input-Output könyvtár, ez tartalmazza az képernyőre írás, vagy az esetleges billentyűzeten keresztül történő adat bevitelhez szükséges függvényeket.
- conio.h: Ez is input-output függvényeket tartalmaz, de buffer nélkül viszi végbe, bizonyos esetekben célszerű használni.
- string.h: string kezelő függvényeket tartalmazó könyvtár.
- stdint.h: Itt vannak definiálva a különböző hosszúságú integer fajták. Mivel a mikrokontrollerünkben figyelnünk kell ezekre a méretekre, így ajánlott a műveleteket már a PC oldalon is ezekkel a változókkal végezni.
- windows.h: A windows operációs rendszerben ez tartalmazza a sorosport megcímzéséhez és kezeléséhez szükséges függvényeket, így ennek segítségével tudjuk az UART kapcsolatot a PC és a master mote között létrehozni.
- file.h: Ebbe a függvénykönyvtárba raktam az intel hex fájl kezeléséhez szükséges függvényeket
- uart.h: Itt találhatóak az UART kezeléséért felelős függvények, ennek megírásához használtam fel a windows.h könyvtárat.

5.1.2 A program működésének rövid áttekintése

Az alábbiakban nem veszem pontról-pontra a meghívott függvényeket, és módosított változókat, csak a fontosabb lépésekre szorítkozok.

1. A main függvény meghívása után lefut egy függvény, ami megnyitja és megvizsgálja, hogy az intel hex fájlunk megfelel-e a formai követelményeknek,

például tartalmazza a címkiterjesztés sort. Ha hibát talál, akkor leáll a program és egy hibaüzenettél tér vissza.

2. A program újra megnyitja a fájlt és inicializálja az UART-ot. Ha valami probléma lép fel, hibával tér vissza a program. Az UART port inicializálása előtt létre kell hoznunk egy HANDLE típusú változót, amin keresztül később el tudjuk érni a portunkat.
3. Elindul egy ciklus, aminek első elemeként kiolvas egy sort az intel hex fájlból. Egy sorban maximum 1+2+1+16+1=21 byte található így maximumálisan ennyi adatot olvasunk ki egyszerre, de amennyiben a sorban kevesebb adat található, ami mellesleg a sor első byte-jében rögzítve van, akkor persze kevesebbet. Intel HEX fájlról bővebben a 3.2. fejezetben olvashattunk.
4. Ezek után a kiolvasott adatból csak az adat és cím byte-okat tartjuk meg. A kiolvasott címből, és az Intel HEX fájl lineáris címkiterjesztés alapján a 4.2.2. fejezetben tárgyalt módszerrel átalakítjuk a címet, és betöltjük egy 18 byte-os tömb első 2 byte-jába, a fentmaradó helyet pedig a kiolvasott adat byte-tokkal töltjük fel. Amennyiben nincs meg 16 byte, akkor a maradékot 0xFF-el töljük fel. Ez nem okozhat problémát a program végrehajtásában, mivel alapállapotban ezzel van feltöltve a flash memória. Ebben a lépésben egy olyan tömböt kapunk amit máris kiküldhetünk az UART-on keresztül a master mote-nak.
5. Ebben a lépésben kiküldjük az UART-on keresztül a 18 byte hosszú tömbünket a master-nek. Nem várunk azonnali választ, az ellenőrzést később végzi a program.
6. Az adat leküldése után a program egyből várni kezd a master-re, hogy az visszaküldje neki az adatot. Ha minden rendben van, akkor tovább megy, ha nem akkor növeli a hiba számlálót eggyel.
7. Ha vége a fájlnak (EOF), és a hibák száma nulla akkor küld egy speciális csomagot, ami jelzi a master-nak, hogy a küldés hibátlan volt és kezdheti a továbbítást.

5.1.3 UART port használata

Mint azt az előző fejezetben említettem, szükségünk van egy HANDLE változóra, ezt érdemes a könyvtárban globálisan létrehozni, mivel az inicializálás után, az írás, illetve olvasás függvénynek is használnia kell. Ezentúl szükség van egy DCB típusú (ez egy a windows.h-ban deifinált típus) struktúra változóra ami az UART paramétereit tartalmazza, illetve egy COMMTIMEOUTS ami pedig a time out paramétereket.

Az UART paramétereinek beállítása a gyakorlatban:

- A `HANDLE hserial` inicializálása: `hSerial=CreateFile( "\\\\.\\COM5", GENERIC_READ|GENERIC_WRITE, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL );` Ezzel a COM5-ös portot inicializáljuk fel. Ekkore ügyelnünk kell rá ,hogy más program ne használja, különben hibát fog dobni.
- `DCB dcbSerialParams` struktúra praméterezése:

```
dcbSerialParams.BaudRate = CBR_38400;
dcbSerialParams.ByteSize = 8;
dcbSerialParams.StopBits = ONESTOPBIT;
dcbSerialParams.Parity = NOPARITY;
```

- `COMMTIMEOUTS timeouts` struktúra paraméterezése:

```
timeouts.ReadIntervalTimeout = 1000;
timeouts.ReadTotalTimeoutConstant = 1000;
timeouts.ReadTotalTimeoutMultiplier = 1000;
timeouts.WriteTotalTimeoutConstant = 50;
timeouts.WriteTotalTimeoutMultiplier = 10;
```

- Inicializálás után az UART portra történő `init8_t buffer` írása:

```
DWORD bytes_written;
WriteFile(hSerial, buffer, 1, &bytes_written, NULL);
```

- Inicializálás után az UART portról történő `init8_t buffer[1]`-be olvasás:

```
DWORD read;
ReadFile(hSerial, buffer, sizeof(buffer), &read, NULL);
```

5.2 A master mote működése

5.2.1 Felhasznált függvénykönyvtárak

A master mote programhoz az alábbi függvénykönyvtárakat használtam fel:

- master_com.h: ez tartalmazza a master egység általam írt komunikációs függvényeit. Ebben találhatóak az adat rádión keresztüli átvitelét biztosító függvények, úgy mint: egy csomag elküldése és a hozzá tartozó acknowledge lekezelése, csomag fogadása, csomag visszajelzés.
- eeprom_128.h: Ez tartalmazza a mote külső EEPROM-ját kezelő függvényeit, mint például: egy 16 byte hosszú tömb memóriába írása, memória olvasása.
- io.h: ez tartalmazza a mikrokontroller regisztereit címző változóneveket.
- interrupt.h: a megszakításokat kezelő függvények.

- Inttypes.h: ebben vannak definiálva a nem standard integer típusok, mint például: uint8_t ami egy 8 byte-os integer-t jelöl.
- mcu_avr_atmega128_api.h: ez tartalmazza a MIT MÓT fejlesztői által a rendelkezésünkre bocsájtott mikrokontroller kezelő függvényeket, mint: az UART kezelése.
- dpy_trm_s01.h: ez tartalmazza a dpy kártya perifériáit kezelő függvényeket. Ennek szerepe csupán reprezentatív jellegű, a programban funkcionális szerepet nem tölt be.
- com_r04_s01b.h: az ISM sávú rádiós kártya, rádiós portját kezelő elemi függvények találhatóak benne, mint küldés, fogadás.
- msgDataStruct.h: Ebben a fájlban található az üzenet struktúránk definiálása.
- etc.h: Egyéb, máshova be nem sorolható függvények találhatóak benne, mint például: késleltetés.

5.2.2 A program működésének rövid bemutatása

1. A master mote-on egy példa alkalmazás fut, ami rádión keresztül bizonyos időközönként lekérdezi a slave egységen található kapcsolók állását. Ez csupán csak bemutató szerepet tölt be.
2. Ha a PC UART-on keresztül adatot küld a master mote-nak, akkor egy megszakítás lép érvénybe, és az addig futó programtól függetlenül meghívódik. Ez fogadja az UART interface buffer regiszterébe töltött adatot, és feldolgozza azt.
3. Ha elegendő, vagyis 18 byte-nyi adat összegyűlt, vagyis átküldtük az intel HEX fájl egy sorát (2 byte cím + 16 byte adat) akkor, ezt az EEPROM memóriájába írja. Addig a PC oldali kliens várakozik a master mote-ra, hogy az visszaküldje a fogadot adatot, ellenörzés céljából.
4. A 18 byte-nyi adatot amikor az EEPROM memóriájába írja, kitölt egy memóriatérképet, ami tartalmazza, hogy melyik 16 byte-os memória szegmensben van adat, és ezt az EEPROM 0x1F000-tól kezdődő részébe írja. Itt ugyanis a NRWW memória helyezkedne el (NRWW lásd 3.1.2. fejezet), vagyis ide nem írhatnánk adatot az EEPROM-ba. Ez a memóriatértkép azért fontos, hogy számon tartsuk melyik adatcellákat kell

majd továbbküldenünk, és ezáltal ne kelljen minden esetben, mind a 124 kbyte-nyi RWW memóriát átvinni. A memóriatértképről kicsit bővebben: a 4.3.1. fejezetben.

5. Miután a master az EEPROM-jába írta utána kiolvassa, és visszaküldi a PC oldali kliensnek ami összeveti a kiküldött adattal. Ha ez nem egyezik, akkor a PC oldali program megjegyzi, hogy hibás volt az átvitel.
6. Ha sikeresen átjött minden adat, és hibát sem talált a PC, akkor kiküld egy utasítást a master mote-nak, hogy elkezdheti a továbbítást a rádión keresztül.
7. A master mote a memóriatérkép alapján kiolvas egy 16 byte-os memória szegmenst és az ebből létrehozott üzenetstruktúrát továbbítja (üzenet struktúrát lást 4.2.1. fejezet) rádión keresztül a slave-nek, ami miután megkapta a csomagot vissza jelez a masternek, hogy a csomag küldés sikeres volt.
8. Ha minde adatot sikeresen átvitt, törli a teljes EEPROM memóriáját, hogy az tiszta legyen a következő adat fogadásra.
9. Elkeződik újra a felhasználói program ami egy darabig bizonyára sikertelenül próbálja meg elérni a slave-t, mivel az éppen újraprogramozza magát.

A master mote-nak folyamatosan csatlakoztatva kell, hogy legyen a PC-hez, hogy a PC-ről vezérelve bármikor el tudjon indulni az adat átvitele. Mivel az adat fogadását egy megszakítás következtében meghívódó függvény végzi, így ez nem blokkolja közvetlenül a master-on futó alkalmazást. Ez a megszakítás az UART adat regiszterébe való írásakor aktiválódik.

A master mote újraindítása után, illetve egy sikeres adat továbbküldés után automatikusan törli a teljes küldő EEPROM memóriáját, ami azért fontos, hogy a következő adat fogadásakor már üres legyen a memória, és ne maradhassan benne programkód az előző programozásból, ami esetleg tovább küldve ismeretlen működéshez vezetne.

A master működésének egyik kritikus pontja az adat fogadásakor a fogadott adat külső EEPROM-ba írása, mivel ekkor a PC folyamatosan vár a válaszra, így a master addig

kvázi várakoztatja a PC-t, amin emiatt egy elég nagy time out van beállítva, de mégis ha a master mote nem tudja időben beírni az adatot az EEPROM-ba, akkor újra kell indítani a mote-ot, hogy törölje a memóriáját, és újra kell kezdeni a folyamatot.

A memória végén létrehozott memóriatérkép csupán az adatátvitelt hivatott gyorsítani, mivel az atmega128-as 128 kbyte-os flash memória kiugróan magas a többi 8 bit-es mikrokontrollerhez képest, és vélhetően a felhasználó nem fogja a memória egésszét felhasználni, ezért nincs is szükség a fentmaradó „üres kód” továbbítására. Viszonyításképpen, a példa alkalmazásban, a slave egység RWW memóriájában található programkód mindössze 4.3 kbyte, ami tartalmazza a kommunikációs függvényeket.

A példa alkalmazásban a master mote generálja az ünenetcsomagokat, a slave egység csak válaszol rájuk. Kétféle üzenetet küld:

1. Állapot lekérdezést: ez hivatott a felhaszálói alkalmazást reprezentálni, és ezt szeretnénk módosítani az újraprogramozással. A példában csak a slave egységen található kapcsolók állását olvassuk le. A kérdést a master kezdeményezi, a slave csupán válaszol. Egy ilyen állapotjelentés fogadása után választ küld az üzenet megérkezésének tudomásulvételéről.
2. Programkód üzenet: A programozást is, csak úgy mint az állapotlekérdezést, a master kezdeményezi, és az adott üzenetkeret address szegmensében jelzi a slave felé, hogy most programkód jön.

Az RF átvitel esetén, az UART-al ellentétben a küldő kap visszajelzést az elküldött csomag megérkezéséről, de ha mégsem akkor újraküldi a csomagot. Ezen felül a MIT MÓT fejlesztői gondoskodtak az átvitel során a CRC kódos hibavédelemről, ami az átküldendő adatokból egy ellenörző kódot generál, mind a küldő, mind a fogadó oldalon, és összeveti, hogy ugyanazt generálta-e le, és ha nem, akkor hibát dob, aminek hatására a program újra elküldi a csomagot.

5.3 Slave mote működése

5.3.1 Felhsznált függvénykönyvtárak

- slave_com.h: Ebben találhatóak az általam írt rádiókommunikációs függvények. Úgy mint a csomag fogadása, csomag felismerése, és az azt kiszolgáló függvények hívása.
- slave_eeprom.h: Itt találhatóak a külső EEPROM-ot kiszolgáló függvények, úgy mint: blokk írása, olvasása.
- mcu_avr_atmega128_api.h: A mikrokontroller kártyához tartozó függvényeket tartalmazza, azzal a kiegészítéssel, hogy itt találhatóak a bootloader funkciót megvalósító függvények, mint pl.: ami kiolvassa a külső EEPROM-ot, page-ekbe rendezi, aztán a flash memóriába írja. Azért itt kaptak helyet ezek a függvények, mivel így csak egy section-t kell létrehozni a makefile-ban. (lásd.: 5.3.3. fejezet).
- dpy_trm_s01.h: DPY kártya perifériáinak kezelésére szolgáló függvénykönyvtár.
- io.h: Ez tartalmazza a mikrokontroller regisztereit címző változóneveket.
- interrupt.h: A megszakításokat kezelő függvények.
- boot.h: A gyártó által a rendelkezésünkre bocsájtott, a bootloader funkcióhoz szükséges elemi függvények, úgy mint: page írása, page törlése, szó page-be írása.
- wdt.h: A hardveres restart-hoz szükséges függvények.

5.3.2 A program működésének rövid áttekintése

A slave mote-on futó példaprogram működésének rövid áttekintése:

1. A slave alapvetően figyel a rádiós port-on keresztül, és várja a master üzeneteit, Ha állapotlekérdezést kapott, akkor visszaküldi a kapcsolók állását, és vár az acknowledge üzenetre, míg ha programkódott kapott, akkor beírja a külső EEPROM-ba, és ő küld egy acknowledge-t. A harmadik variáció pedig, hogy utasítást kap arra kezdheti a flash memória átírását, vagyis meghívhatja a

bootloader függvényeket.

2. Amennyiben programkódot kap, azt a saját külső EEPROM memóriájába írja, majd ellenőrzés céljából kiolvassa azt, és összeveti a fogadott adattal.
3. Visszajelzést küld a master mote-nak az adat sikeres átviteléről.
4. Ha minden adat átjött, és megkapta az utasítást az újrakonfigurálás megkezdéséhez a program az NRWW memóriában található bootloader alkalmazáshoz ugrik. Ez az alkalmazás végigmegy az egész RWW memória címtartományán és a külső EEPROM-ból minden adatot a flash memóriába másol.
5. Feltöltjük a flash page buffer-t, és mivel a flash 256 byte-os page-kbe van rendezve, az EEPROM pedig 128 byte-osokba, így kétszer kell page olvasást kezdeményezni egy buffer feltöltés után.
6. A flash-ben található page-t töröli.
7. A helyére írja a buffer tartalmát.
8. Ha végzett az egész memóriával, egy hardveres reset-tel újra indítja magát.
9. Újraindítás után az első dolga, hogy törölje a teljes külső EEPROM memóriáját.
10. Elindul a felhasználói alkalmazás.

A rádiós adatátvitel során előfordulhat, hogy valamelyik adat átvitele sikertelen, ekkor a slave nem küld visszaigazolást, ezért a master újraküldi a csomagot. Ha esetleg az EEPROM írás során lép fel hiba, és az írás utáni kiolvasás során valami eltérést észlel a program, akkor se küld visszajelzést a master részére, jelezve, hogy művelet újrakezdésére lenne szükség. Íly módon nincs megkülönböztetve a két fajta hiba, ebből kifolyólag a hiba detektálására a master oldalán van lehetőség.

Amennyiben a bootloader alkalmazás futása közben lép fel hiba, abban az esetben jó eséllyel használhatatlanná válik az alkalmazásunk, és manuális beavatkozást tesz szükségessé. Az alkamazás futása során a DPY kártya LED-jei adnak visszacsatolást a felhasználónak, amelyek villogásából látni ha a mikroprocesszor éppen „dolgozik”.

5.3.3 A bootloader a programozói gyakorlatban

A bootloader alkalmazáshoz a bootloader-hez felhasznált függvényeinknek az NRWW memóriában kell elhelyezkedniük. Mivel ennek mérete a fuse bitekkel változtatható, így a fordító programunk nem tudja, hogy hova kell raknia ezeket a függvényeket. Ehhez létre kell hoznunk egy úgy nevezett section-t, ami egy területet

takar a memóriában, ahova utána programkódot helyezhetünk. Ilyen section-öket eredendően tartalmaz minden progam, például .text section-nnek hívják azt ahova alapállapotba kerülnek a függvények, .data section-nek pedig azt, ahol az SRAM memória található. Ahogy az látszik, ez egy általános eljárás, vagyis más célokra is használhatjuk, de a bootloader alkalazásunkhoz létre kell hoznunk egy .bootloader section-t az NRWW memóriában. Ehhez a fordításhoz használt makefile-ba, ami a programfordító számára tartalmaz utasításokat az alábbi direktívát kell használnunk:

```
-Wl,-section-start=.bootloader=0xf800
```

Ezzel a fordító létrehoz egy .bootloader nevezetű section-t a 0x1F000 címen, itt ugyanis szavak szerint van címezve a memória, és 1 szó 2 byte.

Ezek után meg kell adnunk a fordítónak, hogy milyen függvényeket helyezzen ebbe a bootloader section-be. Ezt az alábbi módon, a függvény prototípusának létrehozásakor tudjuk megtenni:

```
int boot_fv (void) __attribute__((section(".bootloader")));
```

Ennek hatására a fordító a boot_fv nevezetű függvényt a .bootloader címre helyezi, amit mi előzőleg 0x1F000-nak definiáltunk. Természetesen ha több függvénynek adjuk meg ezt a direktívát akkor ezeket egymás után fűzi. Mint ahogy azt már korábban említettem fontos, hogy minden függvény amit a bootloader alkalmazás meghív, itt a bootloader section-ben legyen elhelyezve. Illetve kerüljük az olyan változók definiálást, amit a fordító a flash RWW section-jben helyez el, ilyet a .noinit section használata. (bővebben lásd: [10])

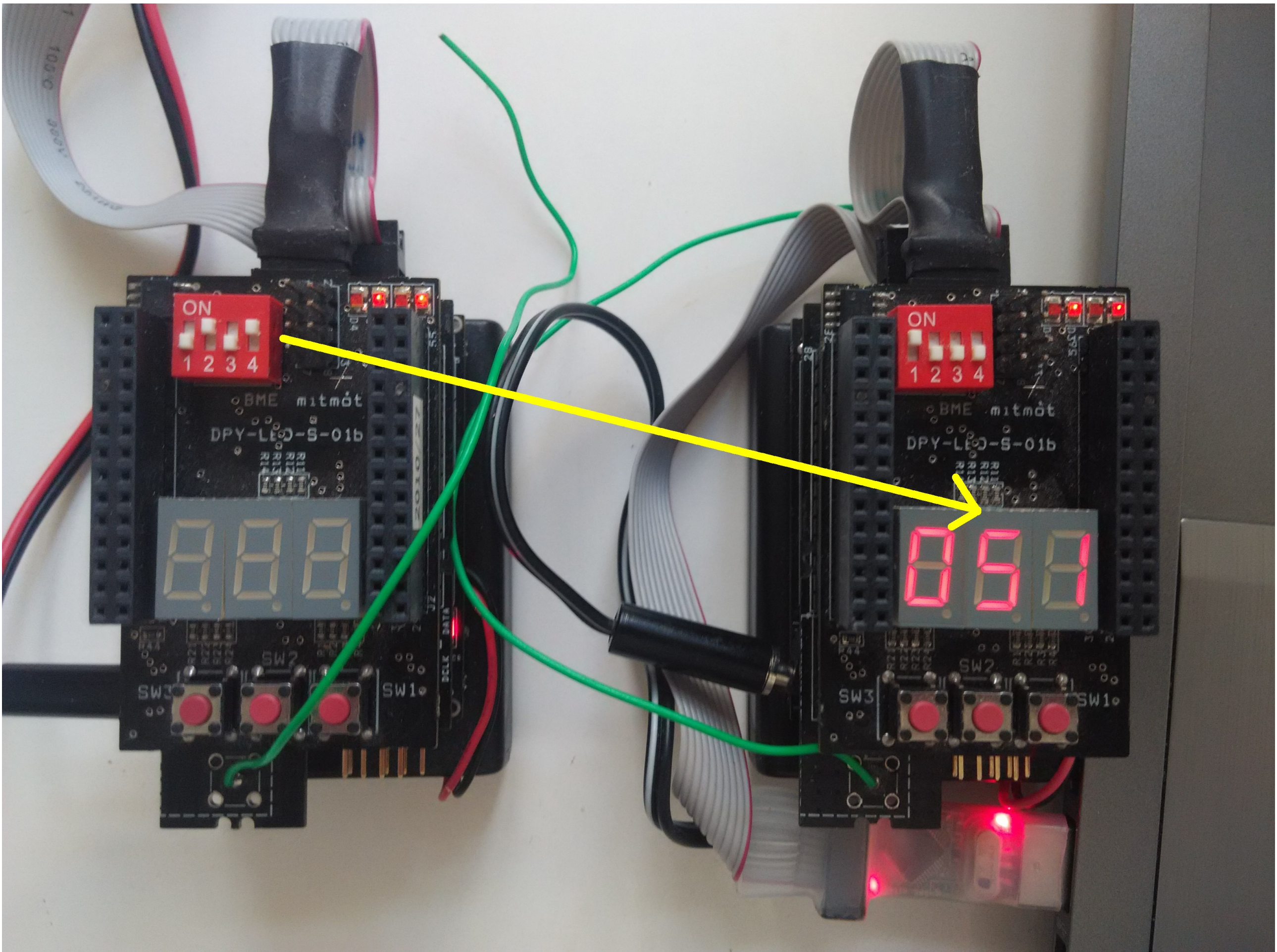
A page írására a gyári boot.h függvénykönyvtár dokumentációjában található példa, ami bemutatja egy page flash memóriába írását. Lásd: [11]

6 Eredmények

Ez a fejetet az elkészült mintaprogram működését hivatott bemutatni. Az alkalmazás működéséhez szükség van egy:

- PC-re
- két MIT MÓT mikrokontroller kártyára és a rajtuk elhelyezett külső EEPROM chipre (lásd: 3.1.8. fejezet)
- két MIT MÓT rádiós kártyára
- egy soros port illesztő kábelre (lásd: 3.2. fejezet)

A példa felhasználói alkalmazásban a master folyatosan állapotlekérdezés parancsokat küld a slave számára (4.2.1), ekkor a master második hétszegmensű kijelzőjén a 6.1. ábrán látható módon a slave kapcsolóin beállitott OFF,ON,OFF,ON állapotnak megfelelő 5-ös kód jelenik meg.



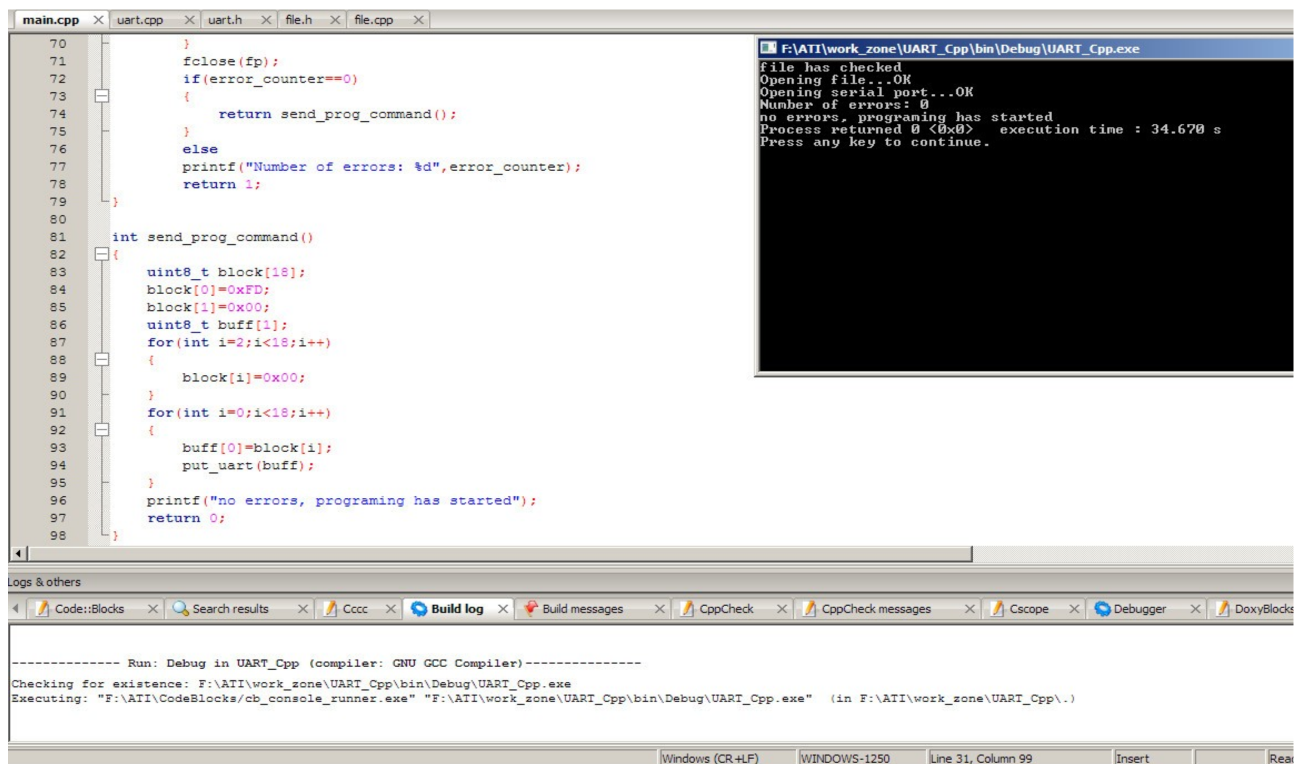
6.1 Ábra: master (jobb oldalon) és slave (bal oldalon)

A jobb oldali kijelzőn a master mote megjeleníti, hogy hanyadik próbálkozásra sikerült elérnie, vagyis ha nem elérhető akkor folyamatasan növekszik a szám, és az első sikeres elérésnél újra egyes értket vesz fel. Ez csupán a lehetséges hibák detektálást segíti.

Ahhoz, hogy a slave egységünkön található programot módosítsuk a megírt programból elő kell állítani egy Intel HEX fájlt (lásd: 3.2. fejezet). Ez a fájl nem tartalmazhat semmilyen, az NRWW memóriában (3.1.2) található programkódot, mivel ezt nem tudjuk az alkalmazással felülírni. Ezt a fájlt be kell másolni a programunk gyökérkönyvtárába, és „slave.hex”-re átnevezni.

A program induláskor lellenőrzi a fájlt, hogy megfelel-e az Intel HEX fájl formátumnak, és nincs-e programkód az NRWW memória területén. Ezek után automatikusan felkonfigurálja az UART virtuális interface-t, és ha mindent rendben

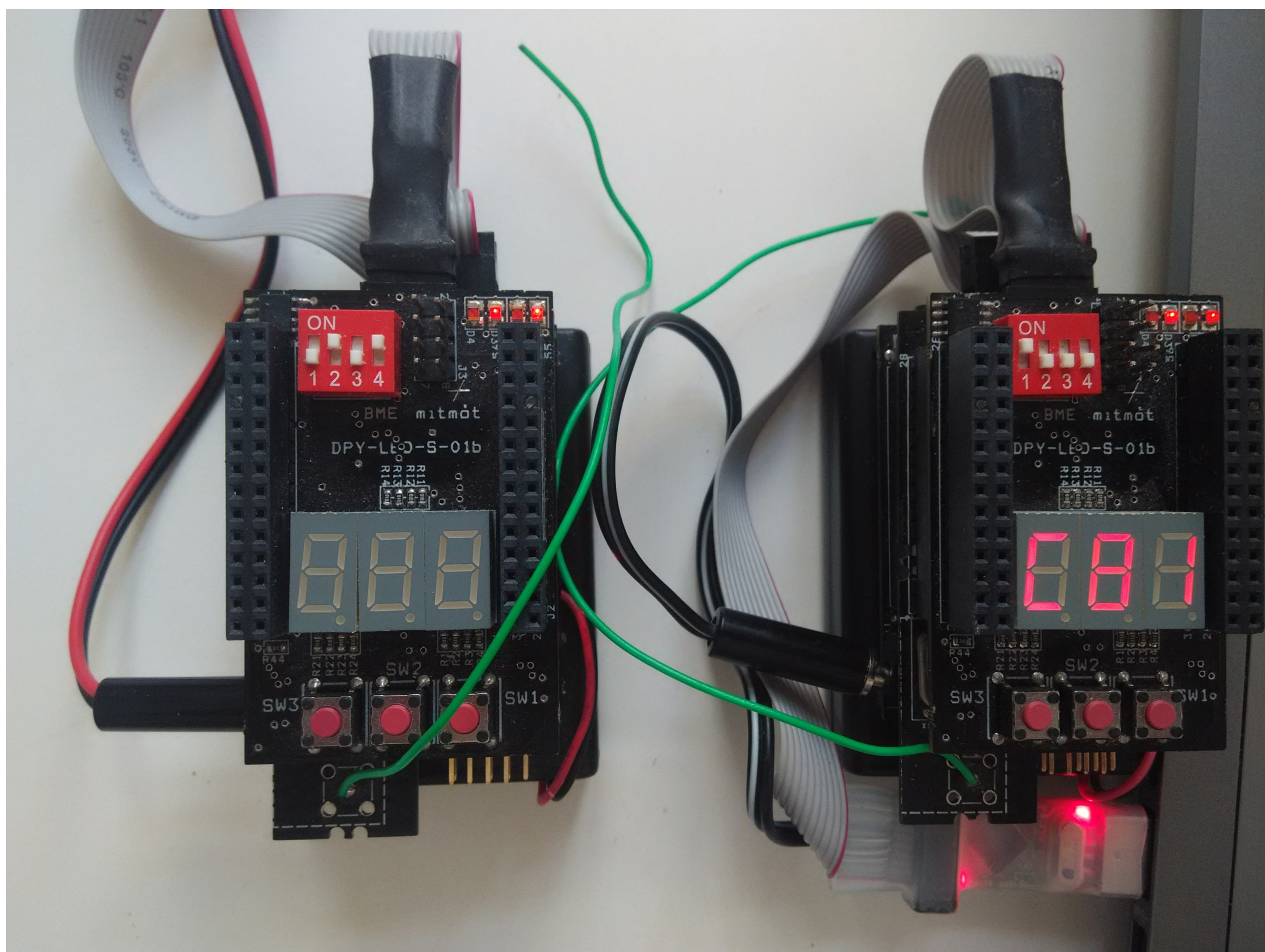
talált, akkor el is kezdi az adat küldést a master felé.



6.2. Ábra: sikeres adatátvitel a PC-ről

Amennyiben az adat sikeresen átment a master felé és az lementette az EEPROM memóriájába, és az adatellenőrzés során a program sem talált hibát, a PC küld egy parancsot a master számára, hogy tovább küldheti az adatot a slave felé.

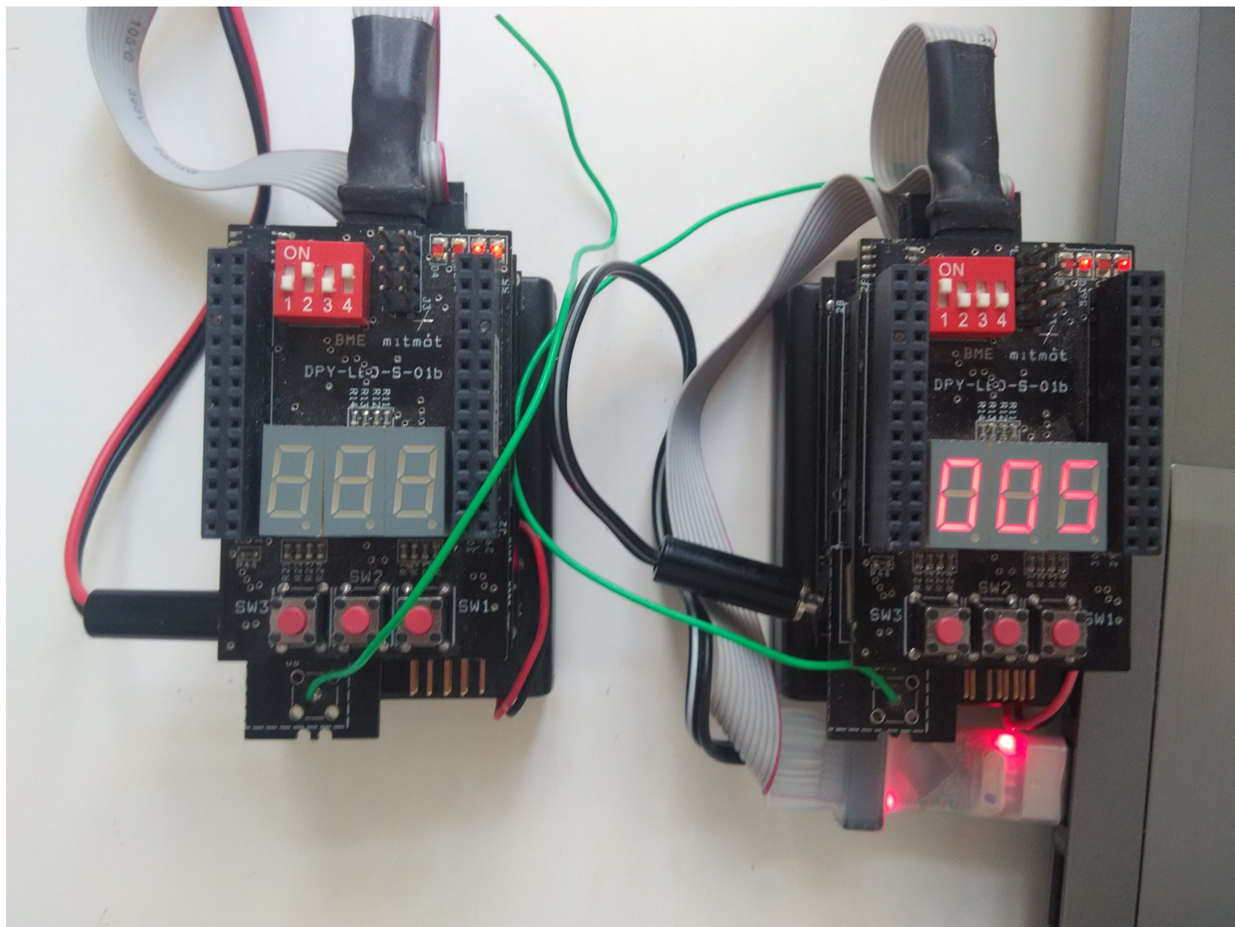
A DPY kártyán található LED-eknek csupán a program futását illusztráló szerepe van, semmilyen más funkciót nem látnak el.



6.3. Ábra: Programkód átvitele a master-ről a slave-re rádión keresztül

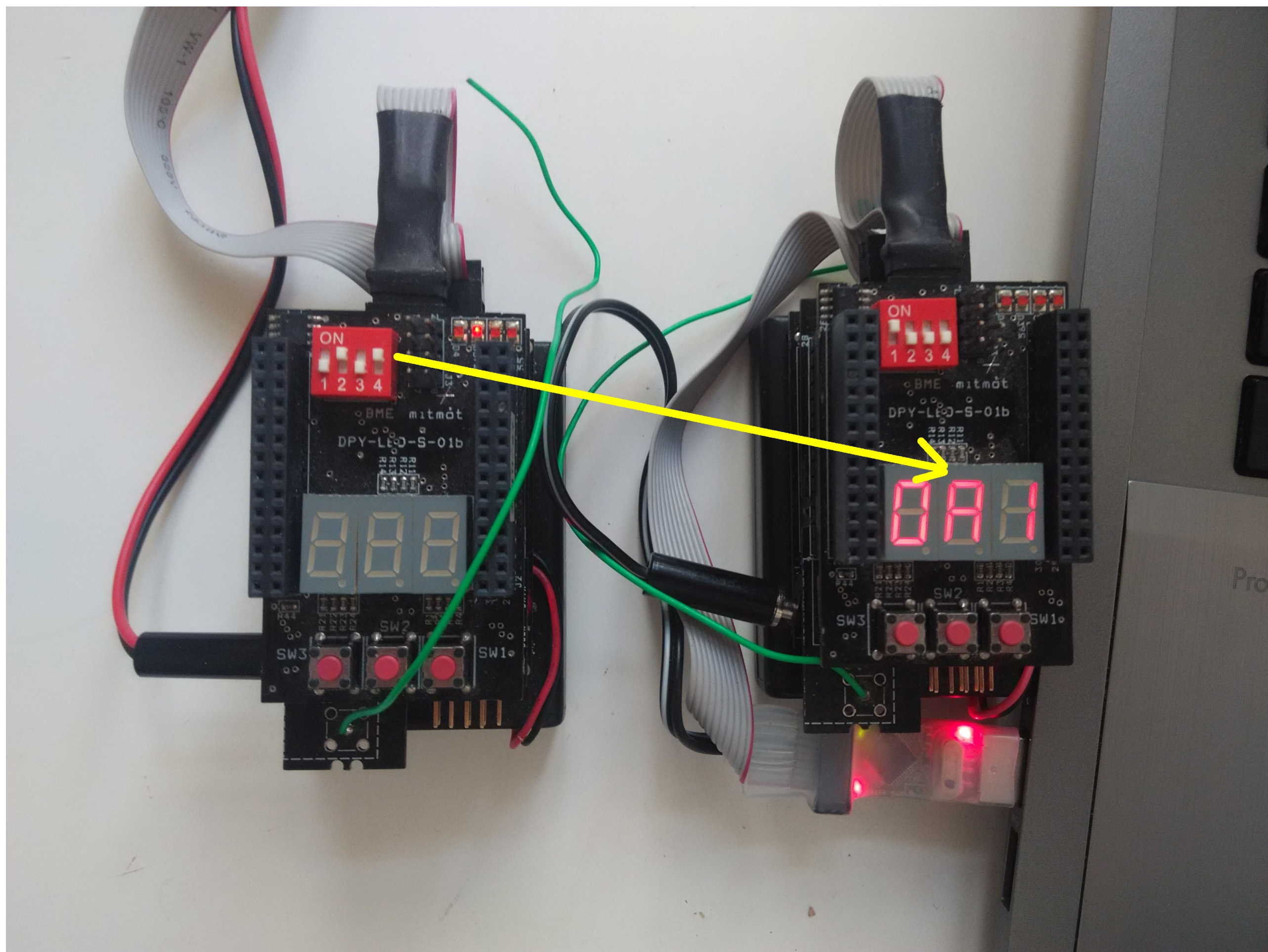
Ha a master megkapta a parancsot a PC-től, hogy elkezdheti a rádiós adatátvitelt akkor a master a 4.2.1. fejezetben tárgyaltak szerint továbbítja a programkódot amit slave a saját külső EEPROM-jába ír. Ekkor a master első két hétszegmensű kijelzőjén az éppen aktuális üzenet csomag sorszáma látható, a baloldai kijelzőn pedig a csomagküldési próbálkozások száma, csak úgy mint a 6.1. ábrán.

Ha minden adat átvitele sikeres volt a master küld egy üzenetet a slave-nek amiben jelzi, hogy nincs több adat, és a slave elindíthatja a bootloader alkalmazást. Ekkor a master automatikusan törli a saját EEPROM memóriájának tartalmát.



6.4. Ábra: master-slave az újrakonfigurálás után, miközben a slave a programmemóriát frissíti.

Miközben fut a bootloader alkalmazás, ami a slave külső EEPROM-jának tartalmát a flash memóriájába írja, a master természetesen nem tudja elérni ezért próbálkozások száma folyamatosan nő, ahogy ez a kijelzőn is látszik, és a kapcsolók állására is az alapértlemezett nulla értéket mutatja.



6.5. Ábra: master-slave az újrakonfigurálás után

Az új felhsználói programunk lekérdezi a slave kapcsolóinak állását, és azok bitenkénti negált összegét adja vissza. Ahogy az a mellékelt ábrán is látszik, az újrakonfigurálás sikeres volt. A slave (bal oldali eszköz) kapcsolóinak OFF,ON,OFF,ON állásának megfelelő 1010 bináris kód jelenik meg a jobb oldali master kijelzőjén, hexadecimális formában.

7 Összefoglaló értékelés

A program tesztelése során egy rövid, egyszerű felhasználói programot a PC-től számítva átlagosan 35 másodperc alatt vitt át a hálózat, és történt meg az újrakonfigurálás. A tesztek során egyszer sem tapasztaltam rendellenes működést, hibát.

Amennyiben a feljebb tárgyalt alkalmazásunkat szeretnénk egy feladat ellátására felhasználni bizonyos megkötésekkel kell élnünk. Habár igyekeztem a fejlesztés során minél jobban elrejteni a felhasználó elől ezt funkciót, hogy az alkalmazásfejlesztést a lehető legkevésbé akadályozza, mégis a tervezés során figyelembe kell venni pár szempontot:

- A NRWW memóriába lehetővé kell tenni, a bootloader alkalmazás hívását, illetve a folyamatos kapcsolatot a slave- és a master között.
- Az általam illesztett EEPROM memória chip használa kerülendő a felhasználói programban, mivel ez a bootloader számára van fenttartva.
- A flash memóriában helyet kell hagyni a bootloader alkalmazásnak.

Mindent összevetve a fejlesztés sikerrel zárult, a példaalkalmazás több ízben is sikeresen végre hajtotta a kívánt feladatot, és közben a tőle elvárt működésnek is eleget tett. Mind használhatóságban, mind integritásban felhasználhatónak bizonyul egy valós alkalmalmáshoz.

A példaalkalmazás csupán egy master-slave-ből álló, pont-pont kapcsolatú hálózatban mutatja be a probléma megoldását, de a valóságban valószínűleg ennel bonyolultabbra kell számítanunk, ezért a jövőben érdemes a hálózati kommunikációt még modulárisabban felépíteni, hogy könnyen beilleszthető legyen más topológiájú hálózatokba is. Ezen felül érdemes valamilyen visszaállító módot kidolgozni a slave egységen belül, ami egy esetleges hibássan feltöltött program esetén visszaállítja a slave-en található programot egy olyan verzióra, aminek segitségével újra tudjuk kezdeni a távoli felkonfigurálást.

8 Irodalomjegyzék

[1]: Budapesti Műszaki és Gazdaságtudományi EgyetemMéréstechnika és Információs Rendszerek TanszékBeágyazott Információs Rendszerek csoport, AVR mikrovezérlő kártya felhaználói dokumentáció, 2005

[2]: wikipédia, Harvard architecture,

[3]: Atmel, 8-bit Atmel Microcontroller with 128KBytes In-System Programmable Flash,

[4]: Budapesti Műszaki és Gazdaságtudományi Egyetem, Méréstechnika és Információs Rendszerek Tanszék, ISM sávú rádiós kártya felhasználói dokumentáció,

[5]: BME-MIT, A DPY-LED perifériakártya Felhasználói és fejlesztői dokumentáció, 2005

[6]: BME-MIT, Soros átviteli protokolok,

[7]: Dr. Tevesz Gábor docents, Mikrokontroller alapú rendszerek,

[8]: Microchip, 24FC1025 data sheet,

[9]: lura.sk, happyJTAG weboldal,

[10]: GNU project, avr-libc Reference Manual,

[11]: GNU project, Bootloader Support Utilities documentation,