SZAKDOLGOZAT

Kigyósi Tibor

Debrecen

2010

Debreceni Egyetem

Informatika Kar

Mikrokontroller alapú léptetőmotor-vezérlő program fejlesztése

Témavezető:

Szabó Zsolt

Mérnök tanár

Készítette:

Kigyósi Tibor

Mérnök informatikus

Debrecen

2010

Tartalomjegyzék

Ábrajegyzék

1. Bevezetés

1.1. Témaválasztás

Mérnök informatikus szakon a Mérés és folyamatirányítás nevű szakirányt választottam, melynek során több tantárgy is a mikrokontrollerek programozását ismertette meg velünk. A téma elnyerte a tetszésemet, ezért úgy döntöttem, szeretném jobban megismerni ezeknek az eszközöknek a működését és programozási technikáit. A mikrokontrollerek felhasználása széles körben elterjedt, az egyik ilyen terület a léptetőmotorok vezérlése. Mikrokontrollerek segítségével vezérelt léptetőmotorokat az ipar számos területén használnak. Ezek mélyebb tanulmányozásával a későbbi tanulmányaim és munkáim során hasznosítható fontos ismeretekre tehetek szert.

1.2. A szakdolgozat célja

A szakdolgozat alapjául szolgáló vezérlő program fejlesztésével egy – a témavezetőnk, Szabó Zsolt által fejlesztett – nagyobb projektbe kapcsolódtunk be, mint programozók, szaktársammal, Kovács Györggyel. Az alkalmazáshoz egy modult a csoporttársam készített, egyet pedig én magam. A program folyamatos fejlesztés alatt áll, a felhasználók igényeit igyekszünk maximálisan kielégíteni. A későbbiekben több további részt is tervezünk beépíteni a szoftverbe, célunk egy univerzális, bármely mikrokontrollerre könnyedén átprogramozható léptetőmotor-vezérlő program kifejlesztése.

Az elkészült programot a Debreceni Egyetem Orvos- és Egészségtudományi Centrumában található PET-CT Diagnosztikai Kft. használja. A mikrokontrollerek által vezérelt léptetőmotort egy CT-berendezésbe építették be. A motorok segítségével egy asztalt mozgathatunk, melyre egy kísérleti állatot „fektetnek”.

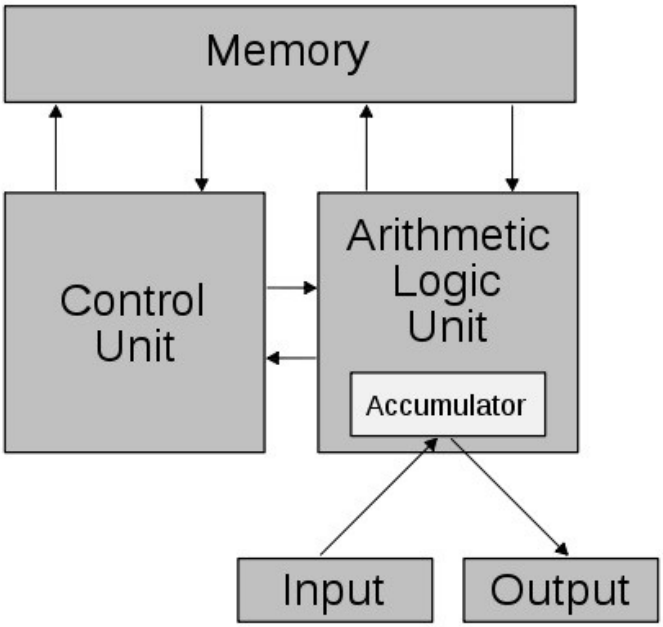
2. Alapismeretek

2.1. Mikrokontrollerek

„A mikrokontroller egyetlen lapkára integrált, általában vezérlési feladatokra optimalizált számítógép.” (http://hu.wikipedia.org/wiki/Mikrokontroller) Az áramköri lapka tartalmazza a mikrovezérlő processzorát, memóriáját és I/O eszközöket. A memória nagysága a különböző gyártók által készített eszközökben más és más. Vannak olyan típusok, amelyek nem tartalmaznak integrált memória egységet, bár a modernebb mikrokontrollerekben ez megtalálható. A mikrokontrollerek ezeken az eszközökön kívül tartalmazhatnak még más berendezéseket. Ilyenek például az I/O portok, melyek segítségével megteremthető a kommunikáció a számítógépünk és a vezérlőnk között. Tartalmazhat még ADC-t (Analog to Digital Converter), így analóg adatokat is küldhetünk a mikrokontrollernek, melyeket digitálissá konvertálhatunk. Ezeken kívül találhatunk még rajtuk órajelegységet, időzítőket és további perifériákat gyártótól és típustól függően.

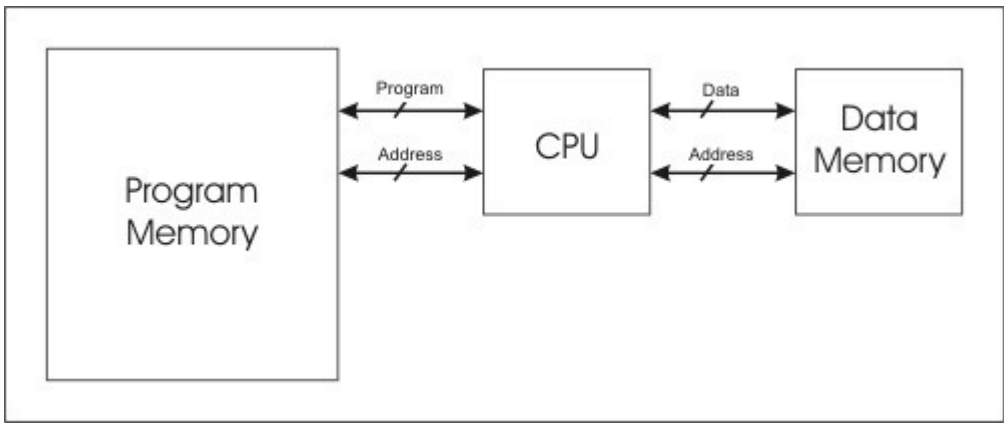
2.1.1. Harvard-architektúra

A mikrokontrollerek architektúrája eltér a hagyományos számítógépektől, ez a Harvard-architektúra. A Harvard-architektúrában az adat- és utasítás buszok elkülönülnek, így egyidejűleg mindkét sínen folyhat kommunikáció. Ezt nevezzük párhuzamos adatátvitelnek. Ezen kívül különálló a program- és adatmemória, melyeket külön alrendszer kezel. A Neumann-architektúrában egyetlen busz található, melyet adatok és utasítások átvitelére is használunk, így ezeket a transzfereket ütemezni kell, nem folyhatnak egy időben. A program és az adat egyazon memóriában található, és ugyanaz az alrendszer kezeli őket.



Ábra 1: Neumann architektúra

A Neumann-architektúrán alapul a legtöbb általános célú mikroprocesszor felépítése. Ezzel a megvalósítással a struktúra egyszerűbb, míg a Harvard-architektúra alapján felépített processzorokkal a műveletek gyorsabban elvégezhetők, nagyobb a teljesítményük.



Ábra 2: Harvard architektúra

2.1.2. Mikrokontrollerek felhasználása

Meglehetősen sokféle mikrokontroller található a piacon, alacsony áruk és kis fogyasztásuk nagyban hozzájárult az elterjedésükhöz. A legnagyobb gyártók az Atmel, Microchip (PIC) és az Intel. A felsorolt cégek eszközeivel az egyetemi órák keretében ismerkedtünk meg.

A mikrokontrollerek a modern elektronikai eszközök alapvető építőkövei. Felhasználásuk nagyon sokrétű, megtalálhatjuk őket az autók motorvezérlő rendszereiben, elektronikus mérőműszerekben, nyomtatókban, mobiltelefonokban, számítógép billentyűzetben, televíziókban, rádiókban, CD lejátszókban, valamint biztonsági berendezésekben is.

2.2. Léptetőmotorok

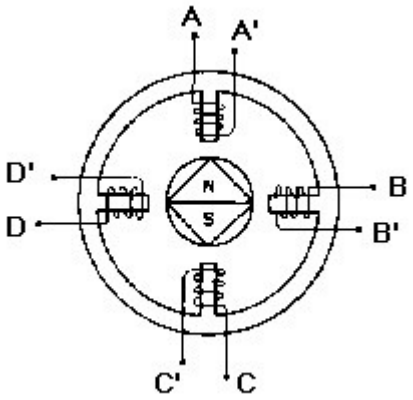
A léptetőmotorok olyan elektromechanikus eszközök, melyek az elektromos impulzusokat diszkrét mechanikai mozgássá alakítják át. Ezen impulzusok segítségével megadott lépésszámmal tesznek meg egy teljes körülfordulást. Digitális jellel vezéreljük őket, ellentétben az egyenáramú motorokkal. Ennek eredményeképpen egyszerűbben valósítható meg a számítógépekkel való kommunikáció. Minden lépésnél adott hibaszázalékkal dolgoznak (körülbelül 3-5%), de ezek egymástól függetlenek, nem adódnak össze, ez az ilyen fajta motorok nagy előnye. Nem tartalmaznak szénkefét, így élettartamuk jelentősen megnő, hiszen ezek a súrlódás hatására elkopnak, és cserélni kell őket. Hátrányai, hogy nem megfelelő vezérlés esetén rezonancia léphet fel, valamint rendkívül nagy sebességeken nem könnyű működtetni.

A léptetőmotorok felhasználása széles körű, általában olyan iparágakban terjedtek el, ahol a pozicionálás pontossága létfontosságú, például gépjárművekben vagy orvosi műszerekben. Ide tartozik még a számítástechnika – használják merevlemezekben, nyomtatókban, szkennerekben, ventilátorokban, CD és DVD meghajtókban, valamint a floppy meghajtókban. Ezeken kívül persze még rengeteg ipari berendezésben megtalálhatjuk őket.

2.2.1. Léptetőmotorok típusai

A léptetőmotoroknak két nagy csoportja létezik, az állandó mágneses és a változó reluktancia motorok. Ezeken kívül beszélhetünk hibrid típusokról, amelyek a vezérlés szempontjából nem különböznek az állandó mágneses társaiktól.

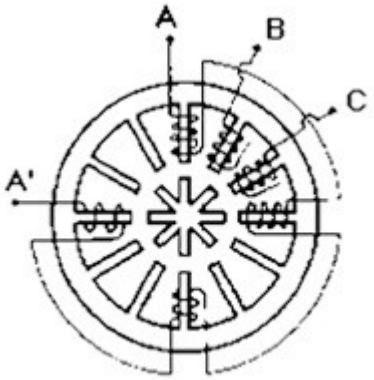
Állandó mágneses (PM, Permanent Magnet) motorok: működése az állandó mágnesből álló rotor és az elektromágneses mező között fellépő kölcsönhatáson alapul. A motoron váltakozó északi és déli mágneses pólusok találhatók, melyek a tengellyel párhuzamos vonalban helyezkednek el. Sorban mágnesezzük a tekercseket, a motor pedig a mágneses vonzás irányába fordul. A rotor mágnesezett pólusai biztosítják a megnövekedett mágneses fluxust, így a PM motor a VR-hez képest jobb nyomaték-karakterisztikával rendelkezik. Lépésszögük általában 45 vagy 90 fokos, ugyanakkor van tartónyomaték, amikor a motor tekercsei gerjesztetlen állapotban vannak.



Ábra 3: Állandó mágneses léptetőmotor felépítése

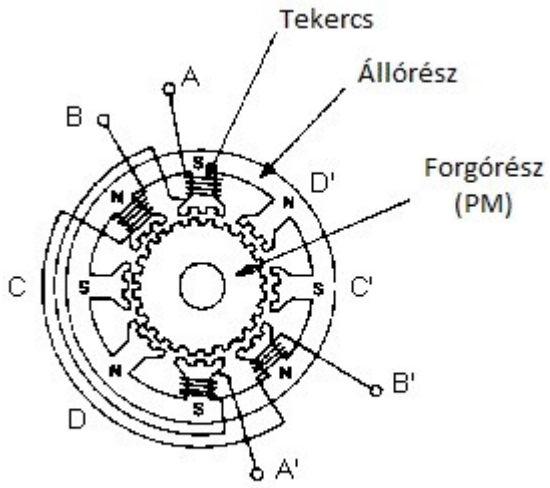
Változó reluktancia motorok (VR, Variable Reluctance): abban különböznek a PM-motoroktól, hogy állandó mágneses helyett lágyvas rotor található benne, ennek következtében nincs tartónyomaték. Amikor az állórész tekercsei feszültség alá kerülnek, a pólusok mágnesezetté válnak. A forgás úgy következik be, hogy a feszültség alatt álló tekercsek magukhoz vonzzák a rotor fogait. A lépésszöget a rotoron és az állórészen található

fogak száma, valamint a tekercsek száma határozza meg, mely általában 5-15 fok. A motor magas sebességeken képes működni, a nyomaték azonban általában alacsony.



Ábra 4: Változó reluktancia léptetőmotor felépítése

Hibrid motorok: a hibrid léptetőmotor drágább a PM-motornál, viszont sokkal jobb teljesítményt produkál felbontás, nyomaték és sebesség terén. A lépésszög általában 3,6 és 0,9 fok között mozog ezeknél a típusoknál. A hibrid motorok a PM és VR változatok előnyeit egyesíti. A forgó- és az állórész a VR-hez hasonlóan fogazott, a tengely körül pedig egy mágnes helyezkedik el.

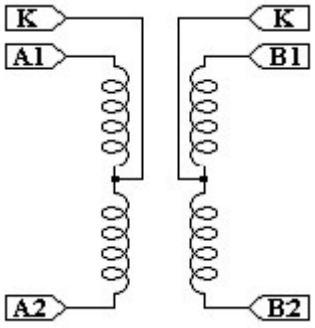


Ábra 5: Hibrid léptetőmotor felépítése

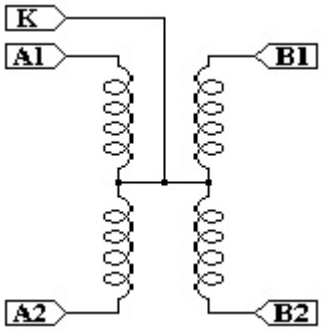
A léptetőmotorok két legelterjedtebb típusa az állandó mágneses és a hibrid típus.

2.2.2. Vezérlési módok

Unipoláris vezérlés: Az unipoláris léptetőmotorok két tekercset tartalmaznak, a mágneses mező két irányában. Így a mágneses polaritások megfordíthatók az áramirány változtatása nélkül is. A tekercsek középkivezetésekkel is rendelkeznek, így a vezetékek száma öt vagy hat, attól függően, hogy a középvezetékeket összekötik-e vagy sem. A motor folyamatos mozgatásához a két tekercsre megfelelő sorrendben kell áramot adnunk. Ehhez a tekercsvégeket sorban földpotenciálra kell kötnünk, a középkivezetéseket pedig a tápegység pozitív kivezetésére. A hat kivezetéssel rendelkező unipoláris motorok bipolárisan is vezérelhetők.

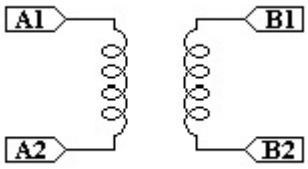


Ábra 6: Hat kivezetéses unipoláris léptetőmotor elvi rajza



Ábra 7: Öt kivezetéses unipoláris léptetőmotor elvi rajza

Bipoláris vezérlés: A bipoláris motorokban a tekercsek nem tartalmaznak középkivezetést. Egy mágneses pólus megfordításához az áramirányt is meg kell fordítani, emiatt az áramkör megvalósítása bonyolultabb az unipoláris motorhoz képest. A működés megoldásához H-híd kapcsolást alkalmaznak. Kétszer annyi vezetékkel rendelkeznek, mint az unipoláris motorok, viszont egy időben fele annyi vesz részt a működésben. A felépítés eredményeképpen a tekercset jobban kihasználja ez a típusú vezérlés, így ezek a motorok erőteljesebbek az unipoláris társaiknál.



Ábra 8: Négy kivezetéses bipoláris léptetőmotor elvi rajza

2.2.3. Léptetőmotorok működtetése

Működtetésük a motorban található tekercsek feszültség alá helyezésével történik. Attól függően, hogy milyen lépésközzel fordul a motor, beszélhetünk teljes lépéses (full step), féllépéses (half step), valamint mikrolépéses (micro step) üzemmódról. Ezeken kívül természetesen más módszerek is léteznek, de ezek ritkábban használatosak. A legegyszerűbb motorok 90 fokos lépésközzel működnek, a legnagyobb felbontásúak 1,8, vagy akár 0,72 fokot is fordulhatnak egy lépésre. A lépésköz csökkenthető még különböző üzemmódok használatával, így a pozicionálás tovább finomítható.

Teljes lépéses üzemmód: sorban adunk áramot a tekercsekre külön-külön, így ahány tekercs található a motorban, annyi lépés szükséges egy körülforduláshoz. Négy tekercsnél például a lépésenkénti szögelfordulás 90 fok. A forgásirány természetesen megfordítható.

Fél lépéses üzemmód: először egy tekercs kap áramot, aztán ezzel egyetemben a következő tekercs is áram alá kerül. Ilyenkor a motor a két tekercs közé mozdul, majd az első tekercset „elengedjük”, így továbbfordulhat. A lépések száma tehát megkétszereződik, a

szögelfordulás pedig a felére csökken. Például egy négy tekercses motornál féllépéses üzemmódban a lépések száma nyolc, a lépésenkénti szögelfordulás pedig 45 fok.

Mikrolépéses üzemmód: ezzel az üzemmóddal állítható be az adott motornál elérhető legkisebb szögelfordulás. Két tekercs között osztjuk szét a motor névleges feszültségét / áramát, ezután az egyik tekercsen növeljük, a másikon ezzel együtt csökkentjük. Így adott lépésközzel a növekvő feszültségű tekercs felé fordul a motor.

Hullám meghajtás (Wave Drive): hullámmeghajtásnál egy időben egy tekercs van feszültség alatt. A lépésszám megegyezik a teljes lépéses üzemmóddal, azonban a motor a névlegesnél jóval kisebb nyomatékkal működik.

3. Használt készlet

Az elkészült teljes programot egy Silicon Labs által gyártott **C8051F120** kódjelű mikrokontrollerre fejlesztették ki. A következőkben a mikrovezérlő felépítését ismertetem.

Nagysebességű 8051-es mikroprocesszor

- 100 MIPS (Million Instructions per Second – Millió utasítás perc másodperc) sebesség
- 100 MHz-es rendszeróra

Analóg perifériák

- 12-bites ADC (Analog to Digital Converter)
- 8-bites ADC
- 2 darab 12 bites DAC (Digital to Analog Converter)

Digitális perifériák

- 64 digitális I/O port
- 5 darab 16-bites időzítő
- 16-bites programozható számláló tömb (PCA – Programmable Counter Array)
- I2C kompatibilis SMBus
- Egyidejűleg 2 soros port érhető el
- „Watchdog” időzítő

Memória

- 8448 bájt adatmemória
- 128 kB Flash-memória, mely belső programozható rendszermemória
- Külső párhuzamos adatmemória interfész

Órajel-generátor

- Belső 24,5MHz-es oszcillátor, mely 2%-os pontossággal támogatja az UART műveleteket

Ábra 9: C8051F120 kódjelű mikrokontroller

4. Vezérlő program

A diplomamunka alapjául szolgáló program megírásakor egy nagyobb projektbe kapcsolódtunk be szaktársammal, Kovács Györggyel, melynek lényege a Debreceni Orvostudományi Egyetemen található PET CT egyik műszerének kezelése volt.

4.1. A főprogram felépítése

Általában a mikrokontrollerek programozása C nyelven történik, mi is ezt választottuk a vezérlő program megírásához, ugyanis több egyetemi óra keretében is ezt a programnyelvet használtuk mikrovezérlőkhöz. A főprogram úgynevezett modulokból áll, igyekszik minden fontosabb működést külön modulban megvalósítani. Ilyenek például a motorvezérlés, a soros kommunikáció, a parancsértelmező, az eseményvezérlő, stb. Fontos része a programnak – mivel több fejlesztő dolgozik rajta egy időben –, hogy ezek a modulok teljesen elkülönüljenek egymástól, egyfajta objektumorientáltságot megvalósítva C nyelv használatával. Ezt a vezérlési egységek header fájljainak két részre bontásával érjük el. Minden modul „*modulnév.c*” elnevezésű állományában hasonlóképpen néz ki az első sor:

```
#define THIS_IS_MODULNÉV
```

Ennek segítségével tudjuk elkülöníteni a header fájlok részeit. Az

```
#if defined(THIS_IS_MODULNÉV)
```

után lévő blokkban találhatók azok a konstans-, változó- és függvénydeklarációk, amelyeket az adott modul *.c kiterjesztésű kódja fog az „include” paranccsal meghívni. Tehát egy másik modul számára ezek a változók és függvények így „láthatatlanok” lesznek. A header fájl másik része egy újabb feltétel:

```
#if !defined(THIS_IS_MODULNÉV)
```

A feltételt követő programegységben adjuk meg azokat a deklarációkat, amelyek egy másik modul számára is elérhetőek lesznek, ha „*include*” parancs segítségével a header fájlt behívják.

Ennek a felépítésnek az eredményeképpen valósíthatunk meg egyfajta objektumorientáltságot C nyelv használatával. Szemléletesen az első feltétel után található változók és függvények „*private*”, a második feltétel után találhatóak pedig „*public*” láthatóságúak lesznek.

Így érthető el az, hogy minden programozó egymástól függetlenül írja meg az általa fejlesztett modult. Csak a „közös” függvények formális paramétereit és visszatérési típusát kell tudnia, a modul elkészítésének módja a programozóra van bízva.

4.2. Parancsértelmező modul

Feladatom a főprogramon belül egy parancsértelmező modul megírása volt. Ennek lényege egy olyan állapot vezérelt program, mely képes két típusú parancs feldolgozására, valamint könnyedén felvehetők új utasítások. A programrész nem mikrokontroller-függő, tehát bármely más mikrovezérlőre fejlesztett alkalmazásba implementálható és használható. A program folyamatos fejlesztés alatt áll, bármikor egyszerűen kiegészíthető új funkciókkal. Az elkészült modul implementálása után három fő parancsot építettek be a programba, valamint több más típusú utasítást. A főbb parancsok a következők:

- *motor*: a mozgást vezérlő és a motor beállításainak módosítására alkalmas parancs
- *ver*: a verziószám lekérdezése
- *help*: segítség

4.2.1. Header fájl

A header fájl az előbbiekben leírt konvenció alapján készült el, megtalálható benne minden konstans, változó és függvény deklarációja. Az első két sorban található utasítások a következőképpen néznek ki:

```
#ifndef __CMD_H__
#define __CMD_H__
```

Ezen két utasítás feladata a header fájl újrafeldolgozásának megakadályozása a programon belül. Egy include hívás során az történik, hogy a header fájl szövegét a fordító a program szövegébe helyezi. Így ha többször hívnánk be az említett állományt, az hibákhoz vezetne a programban. Ennek megoldásaképpen a header fájlon belül deklarálunk egy *__CMD_H__* (ez természetesen tetszőleges lehet) konstanst. Tehát ha ez a konstans már létezik (már *include* paranccsal behívtuk a header fájlt), akkor az *#ifndef* feltételtől annak *#endif* zárásáig tartó részt a fordító átugorja, mivel a feltétel nem teljesül.

A header fájl a kívülről elérhető, valamint nem elérhető változók, konstansok és függvények alapján két részre osztható. A „belső” programozási eszközök az

```
#if defined(THIS_IS_CMD)
```

feltétel után vannak deklarálva.

```
#define MCMD_INIT_CHAR          '<'
#define MCMD_FINI_CHAR          '>'
#define MCMD_DEFAULT_COMMAND    '?'
```

A fenti három konstanst az M-típusú parancsok használják, melyek működését a későbbiekben mutatom be.

Ezek után a különböző STATE konstansoknak adunk értéket, amelyek segítségével kezeljük a parancsok feldolgozását. Az utasításoknak két csoportja van, az M és a W típusúak. Ezekhez különféle állapot konstansokat kell felvennünk.

```
#define MCMD_STATE_INIT             0x00
#define MCMD_STATE_GET_FUNCTION     0x01
#define MCMD_STATE_GET_ADDRESS      0x02
#define MCMD_STATE_GET_VALUE        0x03
#define MCMD_STATE_EXECUTE          0x04
#define MCMD_STATE_ERROR            0x05
```

```
#define WCMD_STATE_INIT          0x06
#define WCMD_STATE_COLLECT       0x07
#define WCMD_STATE_EXECUTE       0x08
#define CMD_STATE_DONE           0xff
```

A CMD_STATE_DONE nevezetű állapot az alapértelmezett. A parancsértelmező indulásakor ezt az értéket veszi fel az erre a célra deklarált változó. Értelemszerűen a W-vel kezdődő konstansok a W-típusú, az M-mel kezdődőek pedig az M-típusú parancsok feldolgozását segítik. Az MCMD_STATE_INIT, valamint a WCMD_STATE_INIT a parancsok feldolgozásának inicializáló állapotai, itt már elkülönülnek a típusok. Az M-típusú parancsoknak három olyan állapota van, melyekben a beírt szöveget dolgozzuk fel, ezek a GET_FUNCTION, GET_ADDRESS és GET_VALUE státuszok. Az M-típusú utasításoknál találunk egy MCMD_STATE_ERROR nevezetű állapotot is, mely az esetleges hibák kezelését végzi, a W-típusúaknál azonban ez másképp történik. Végül mindkét típusú parancshoz létezik egy EXECUTE állapot, mely a tényleges végrehajtást végzi.

A következő két konstanst a parancsokat tároló tömbhöz használjuk:

```
#define CMD_BUFFER_SIZE          16
#define CMD_NUM_OF_COMMANDS      4
```

A CMD_BUFFER_SIZE konstanssal a puffer méretét adhatjuk meg, amely egy később deklarált tömb elemszáma lesz. A CMD_NUM_OF_COMMANDS-ban pedig a különböző kiadható parancsok számát adjuk meg.

Ezek után különböző hibaüzenetekhez deklarálunk sztring konstansokat, melyekben megadjuk az üzenet szövegét. Az *strCOMMANDS[CMD_NUM_OF_COMMANDS]* egy tömb, melybe felvesszük az összes W-típusú parancsot, amelyet a felhasználó kiadhat. Minden parancs szövege előtt található egy szám. Ennek lényege, hogy a program az utasításokhoz különböző jogosultsági szinteket kezel, melyekről a későbbiekben még szót ejtünk.

A továbbiakban deklaráljuk az összes változót, melyeket a program során használunk, valamint a függvényeket, melyekből három található a modulban. A változók deklarálásánál annyit érdemes megemlíteni, hogy különböző tárolási osztályok segítségével helyezhetjük a változóinkat a memória különféle területeire. Ezek az osztályok a következők:

xdata: ha ezzel a tárolási osztállyal deklarálunk egy változót, az a külső RAM-ban kerül elhelyezésre. Címzése 3-4 bájtos mutatón keresztül történik. Ha egy változót tárolási osztály megnevezése nélkül deklarálunk, akkor ez az alapértelmezett osztály. Általános felírási módszere:

```
__xdata unsigned char test_xdata;
```

idata: az ezen osztállyal deklarált változók a belső RAM közvetlenül címezhető területén kerülnek elhelyezésre. Az „i" jelentése *„immediate"* (azonnali), mivel címzése egy bájtos, az adat gyorsan elérhető.

```
__idata unsigned char test_idata;
```

pdata: jelentése *„paged xdata",* az xdata memória elején helyezkedik el, tehát fizikailag az xdatához férünk hozzá a használatával, azonban itt lapozásos módszerrel tároljuk az adatokat.

```
__pdata unsigned char test_pdata;
```

A header fájlban deklarált függvények:

- *cmd_get_level:* A jogosultságok kezelését végző függvény.
- *cmd_parse:* A parancsértelmező modul fő függvénye, mely az utasítások feldolgozását végzi.
- *cmd_initialize:* A *main* függvény számára írt inicializáló függvény, mely elvégzi az alapértelmezett beállításokat.

A header fájl második részében mindössze két függvénydeklaráció található. Ezek a *cmd_parse* és a *cmd_initialize,* melyeket a main függvény használni fog a működés során.

4.2.2. A működést végrehajtó függvények

A cmd_get_level függvény: feladata, hogy egy adott parancshoz adja vissza a hozzá tartozó jogosultsági szintet. A függvény u8_t típust vár paraméterül, és egy ugyanilyen típusú értékkel tér vissza. Ez egy a programon belül definiált előjel nélküli karakter (unsigned char) típus. A függvény paraméterül egy parancs indexet vár, amelyet megvizsgál, és ha nem létezik ilyen indexű parancs az utasításokat tartalmazó tömbben, nullával tér vissza. Különben beállít egy mutatót a tömb megfelelő elemére. Mivel a tömbben az utasítások sztringként vannak tárolva, ezért a megfelelőt át kell alakítanunk számmá. A header fájlban a következőképpen vannak tárolva az utasítások:

```
char* code strCOMMANDS[CMD_NUM_OF_COMMANDS] =
{
        "0 parancs1",
        "0 parancs2",
        "9 ver",
        "9 help",
};
```

Látható, hogy a sztring első karaktere jelöli a parancs jogosultsági szintjét. A mutató beállítása utána megvizsgáljuk, hogy az egy hexadecimális 0 karakterérték és egy hexadecimális 9 karakterérték közé eső számra mutat-e. Ha igen, akkor ennek értékéből kivonva a hexadecimális 0 értéket, megkapjuk a jogosultsági szintet egy feldolgozásra már alkalmas számként.

A cmd_parse függvény: az általam írt programrészlet legfontosabb eleme, ez a függvény végzi a parancsok feldolgozásának lényegi részét. Visszatérési értéke nincs (void típus), formális paraméterlistája pedig kételemű, egy piofile_t* és egy u8_t típus. Az u8_t-ről az előbbiekben már szó esett, a piofile_t pedig egy ugyancsak általunk definiált típus. Ez egy olyan struktúra, amely olyan mutatókat tartalmaz, melyek különböző függvényekre mutatnak.

A struktúra deklarációja a következőképpen történik:

```
typedef struct
{
   void (*tx)(u8_t c) reentrant;
   u8_t (*rx)(void) reentrant;
   u16_t (*poll)(void) reentrant;
   u16_t (*poll_line)(void) reentrant;
   void (*tx_block)(u8_t* dp, u16_t wLength) reentrant;
   u16_t (*rx_block)(u8_t* dp, u16_t wMaxLength) reentrant;
   void (*sync)(void) reentrant;
}  piofile_t;
```

Az általam írt programban ezek közül a függvények közül az *tx(), rx(), poll_line(),* valamint *sync()* függvényeket használtam.

- tx: ezen függvény segítségével a kimenetre egy karakter küldhető
- rx: ezzel a függvénnyel tudunk egy bejövő karaktert beolvasni
- poll_line: ezzel olvashatunk be egy sort a bemenetről
- sync: a bemeneti és kimeneti pufferek kiürítését végző függvény, mely alapállapotba állítja az erre szolgáló változókat

A parancsértelmezőben a *poll_line* függvény segítségével egy változóban eltárolunk egy sort a bemenetről, amelyet később feldolgozunk. Ezt egy while-ciklus követi, melyben a feltétel az, hogy a sort tároló változó nem egyenlő nullával. A cikluson belül az *rx* függvény használatával kiveszünk egy karaktert a feldolgozás alatt álló sorból, majd a sorváltozóból ezt a karaktert ki is töröljük. Ezt követi egy hexadecimális konvertáló rész, melyet a későbbiekben ismertetek.

A következő rész egy do-while ciklus, amely egyszer biztosan lefut, majd a feltételek kiértékelése után vagy folytatódik a futás, vagy vége a ciklusnak. A feltételben az aktuális állapotot tároló változót vizsgáljuk, hogy MCMD_STATE_INIT, MCMD_STATE_EXECUTE, MCMD_STATE_ERROR, WCMD_STATE_INIT vagy WCMD_STATE_EXECUTE állapotban van-e. Ha ezek közül egyik sem teljesül, a ciklus nem fut le többször. A cikluson belül egy switch-case struktúra található, melynek lehetséges esetei a változó által felvehető állapotok. Ezeknek két csoportja van, az M-típusú állapotok és a W-típusúak. Ez a két csoport azért különül el, mert a program megrendelőinek kérésére két típusú parancsot kellett beleépítenünk. Az M-típusú parancsok szintaktikája többféle lehet:

- <p?>
- <p=45>
- <p3A=45>

Tehát nyitó ('<') és záró ('>') karakterek közé kell beírni a parancsot. Az első egy lekérdező utasítás, melyben a függvény nevét adjuk meg, amit egy kérdőjel követ. A második egy értékadás, ahol meg kell adnunk a függvény nevét, utána egy egyenlőségjelet, és végül a kívánt értéket. Az utolsó a legösszetettebb M-típusú parancs, melyben megadjuk a függvény nevét, egy maximum nyolc bájton tárolható hexadecimális számot, egy egyenlőségjelet, és végül ugyancsak egy maximum nyolc bájton tárolható hexadecimális számot.

Az programban találhatunk egy konvertáló részt, mely a hexadecimálisan címeket és értékeket ismeri fel. A *„0”* és *„9”* karaktereket, valamint a köztük találhatóakat úgy alakítjuk számmá, hogy értékükből kivonjuk a *„0”* karakter értékét. Az *„a”* és *„f”* közöttiekből az *„a”,* valamint *„A”* és *„F”* közöttiekből az *„A”* karakter értékét vonjuk ki, majd mindkét esetben tízet adunk a számhoz, így megkapjuk a decimális értéket. Ha egyiket sem ismertük fel, ez a változó 0xFF értéket kap.

A másik csoportot a *„W”,* azaz *Word* (Szó) típusú parancsok alkotják, melyeknél az utasítás szövege egy teljes szó, például:

help vagy *motor*

Tehát a kiadható parancsok e kétféle osztályához más-más feldolgozó utasítások szükségesek, ezért van szükség két típusú állapotcsoport felvételére is.

MCMD_STATE_INIT:

Ebbe az állapotba akkor kerülünk, ha a parancs első vizsgálata során (a whitespace karakterek átugrása után) az M-típusú parancsok nyitókarakterét ismerjük fel. Ezekhez a parancsokhoz a header fájlban deklaráltunk négy változót, melyekben a kiadott parancs paramétereit tároljuk. Ezek a Function, Address, Command és Value változók. Ebben az esetben három változó – Function, Address és Value – értékét állítjuk nullára, valamint a Command változó felveszi az alapértelmezett ('?') értéket.

Ezután módosítjuk az állapottároló változót, amely MCMD_STATE_GET_FUNCTION állapotba kerül.

MCMD_STATE_GET_FUNCTION:

Az előző ágban erre az értékre állítottuk az állapotváltozót, úgyhogy ha újra idekerül a vezérlés, akkor ez a feltétel fog teljesülni. Megvizsgáljuk az előzőekben értéket kapott karakter változót, hogy parancskezdő (MCMD_INIT_CHAR, '<') vagy parancs lezáró (MCMD_FINI_CHAR, '>') karakter-e. Ha kezdő karakter, akkor az állapotváltozó értéket MCMD_STATE_INIT-re állítjuk, ha pedig vége karakter, akkor MCMD_STATE_EXECUTE-ra. Ha ezek közül egyik sem, akkor a kezelendő parancshoz jutottunk, ezért a Function változó felveszi a karakter értékét, az állapotváltozó pedig MCMD_STATE_GET_ADDRESS állapotba kerül.

MCMD_STATE_GET_ADDRESS:

Ahogy az előző esetben is, itt is megvizsgáljuk a feldolgozás alatt álló karaktert, hogy parancskezdő vagy záró-e, és az állapotváltozót ugyanúgy állítjuk MCMD_STATE_INIT-re vagy MCMD_STATE_EXECUTE-ra, ahogy azt tettük az ezt megelőző esetben. Ha a feltételek közül egyik sem teljesül, az előbbiekben bemutatott konvertálás során keletkezett számértéket (*hexval*) vizsgáljuk. Amennyiben ez nem 0xFF, az Address változó értékadását a következőképpen végezzük el:

```
Address = (Address << 4) + hexval;
```

Vegyük például, hogy a kapott hexadecimális szám: A7. Első lépésben a karakterváltozónk az „A” értéket tárolja, a hexval tartalma pedig 10. Ekkor a memóriában az Address változó a következőképpen néz ki:

0	0	0	0	0	0	0	0

Ezt néggyel balra eltolva a tartalma egyelőre nem változik, viszont hozzáadjuk a 10 értéket, így a következőt kapjuk:

0	0	0	0	1	0	1	0

A következő lépésben a karakterváltozónk felveszi a „7” karakter értékét, a hexval értéke ekkor 7 lesz. Az Address változót néggyel eltolva, majd hozzáadva a hexval tartalmát, megkapjuk a kívánt értéket:

1	0	1	0	0	1	1	1

Ha nem INIT vagy FINI_CHAR-ral volt dolgunk, és a hexval értéke is 0xFF volt, akkor a Command változó felveszi a karakter értékét, az állapotváltozó pedig MCMD_STATE_GET_VALUE állapotba kerül.

MCMD_STATE_GET_VALUE:

Ebben az állapotban is az előzőekhez hasonlóan megvizsgáljuk, hogy a karakterváltozó MCMD_INIT_CHAR vagy MCMD_FINI_CHAR értékű-e. Ha igen, ugyanazokat az utasításokat végezzük el, mint az előző állapotoknál, azaz MCMD_STATE_INIT vagy MCMD_STATE_EXECUTE értékre állítjuk az állapotváltozót. Ha egyik feltétel sem teljesül, az átkonvertált értéket (*hexval*) vizsgáljuk, és az előzőekben leírt módon tároljuk el a Value változóban a kapott hexadecimális értéket.

Ha az előzőekben tárgyalt feltételek közül egy sem teljesül, akkor az állapotváltozó értéket MCMD_STATE_ERROR állapotúra módosítjuk.

MCMD_STATE_EXECUTE:

Ebben az állapotban történik az M-típusú parancsok feldolgozása.

```
if(level != 0) {
        bios_print_string(pf, strPRIVILEGED);
        wLen = 0;
        pf->sync();
        m_bCmdState = CMD_STATE_DONE;
}
```

Ha a parancs jogosultsági szintjét tároló változó nem egyenlő nullával, akkor nincs engedélyünk a parancs elvégzésére. Így kiírunk egy hibaüzenetet, a sorváltozót nullára állítjuk. Ezen kívül meghívjuk a sync() függvényt, mellyel alaphelyzetbe állítjuk a puffereket, majd az állapotváltozót CMD_STATE_DONE állapotra módosítjuk, így a parancs végrehajtása nem folytatódik.

```
else if(m_bFunction == 'v') {
        rc = 0;
        pf->tx(CHAR_CODE_CR);
        pf->tx(CHAR_CODE_LF);
        pf->tx('v');
        bios_print_string(pf, strVERNUMBER);
        pf->tx(CHAR_CODE_CR);
        pf->tx(CHAR_CODE_LF);
}
```

Ha van jogosultságunk a végrehajtáshoz, akkor a Function változó értékét vizsgáljuk. Egy feltétellel teszteljük, hogy ez az érték egy „v” karakter-e, ugyanis ez az egyik feldolgozandó parancs. A képernyőre a következőket írjuk ki egy Carriege Return (CR, „kocsi vissza”), valamint egy Line Feed (LF, soremelés) után:

v1.2.3.646

A verziószám természetesen más is lehet, az aktuálisan letárolt számtól függően, majd ezt megint egy kocsivissza és egy soremelés követi.

Az utasítás feladata, hogy írjuk ki a program verziószámát, mely egy külön header fájlban van letárolva strVERNUMBER konstans formájában. A kiíratást a bios_print_string függvény használatával végezzük el.

Ezek után bármilyen új paranccsal bővíthető a kód, ez a következőképpen tehető meg:

```
/*
else if(m_bFunction == '') {
        rc = ??Function(pf, m_bFunction, m_bCommand, m_wAddress, m_wValue);
}
*/
```

Ezt a kódrészletet egy megjegyzésben helyeztem el, így ha később más programozók bővíteni szeretnék, ennek alapján ez egyszerűen elvégezhető. A feltételvizsgálatba az idézőjelek közé fel kell venni a kód betűjelét, majd a parancs hatására végrehajtandó függvényt kell behelyettesíteni a ??Function részbe. Így máris egy új utasítással bővítettük a parancsértelmezőnket.

Ha az előzőekben felsorolt feltételek egyike sem teljesül, az állapotváltozó értékét MCMD_STATE_ERROR állapotúra váltjuk, mert a parancs ebben az esetben nem feldolgozható.

Ezen a case-ágon belül még egy feltétel található, mely megvizsgálja az állapotváltozó értékét.

```
if(m_bCmdState != MCMD_STATE_ERROR) {
    if(rc != 0 ) {
                pf->tx('[');
                pf->tx('E');
                pf->tx('R');
                pf->tx(' ');
                pf->tx(m_bFunction);
                pf->tx(' ');
                bios_print_hex8(pf, rc);
                pf->tx(']');
    }
    wLen = 0;
    pf->sync();
    m_bCmdState = CMD_STATE_DONE;
}
```

Ha ez nem MCMD_STATE_ERROR állapotú, tehát az előző feltételeknél nem az „else” ág futott le, akkor végrehajtjuk a blokkban található utasításokat. Ezután megvizsgáljuk azt a változót, amelynek értékét a lefutott függvény állítja be. Ha ez nem nulla, az azt jelenti, hogy hiba történt a végrehajtás során, a következő szöveget írjuk a képernyőre:

[ER függvénynév hibakód]

A következőkben a sorváltozó értékét nullázzuk, lefut a sync() függvény, valamint az állapot CMD_STATE_DONE-ra módosul.

MCMD_STATE_ERROR:

Ha ebben az állapotban vagyunk, az azt jelenti, hogy valamilyen hiba lépett fel a feldolgozás során. Ezért tx() függvények segítségével a képernyőre az alábbi üzenetek valamelyikét írjuk:

- [ER függvény_neve ??]
- [ER ? ??]

A hibaüzenet attól függ, hogy a Function változó kapott-e már előzőleg értéket vagy sem. Ezek után a sorváltozót nullázzuk, valamint lefut a sync() függvény. Az állapotváltozónk értékül pedig a CMD_STATE_DONE állapotot kapja.

A következőekben a „W” (Word) típusú parancsok feldolgozását végző állapotokat vesszük sorba. Ezek az utasítások, melyek egész szavakból állnak, például a *help*, vagy a *motor* parancs.

WCMD_STATE_INIT:

Ez az inicializáló állapot, ahol különböző beállítások történnek. A puffer indexét jelző változót nullára állítjuk, utána pedig egy karakterkódolás történik. Ha nagybetűt kaptunk, akkor kisbetűsre állítjuk a könnyebb feldolgozhatóság érdekében, majd az index alapján az első karaktert eltároljuk egy pufferben, az indexet pedig növeljük. Az állapotváltozó értékét WCMD_STATE_COLLECT-re állítjuk.

WCMD_STATE_COLLECT:

Ebben az állapotban megvizsgáljuk, hogy a karakterváltozó értéke egyenlő e valamelyik konstanssal a következők közül:

- CHAR_CODE_CR: Carriege Return („kocsivissza”)
- CHAR_CODE_LF: Line Feed (soremelés)
- CHAR_CODE_NULL: nullkarakter

- CHAR_CODE_SPACE: szóköz
- CHAR_CODE_TAB: tabulátor

Ha a feltétel igaz, akkor a parancs végére értünk. A vizsgált karaktert CHAR_CODE_NULL-ra cseréljük, mivel ezeket a parancsértelmezőnek figyelmen kívül kell hagynia. Ezek után az állapotváltozót WCMD_STATE_EXECUTE-ra állítjuk.

Ha a feltétel nem teljesült, akkor a karaktert – amennyiben nagybetűs – kisbetűssé kell konvertálnunk, majd beletenni a pufferbe.

A következőkben történik egy újabb feltételvizsgálat. Megnézzük, hogy a puffer indexének mérete túllépte-e a maximálisan megengedett CMD_BUFFER_SIZE-ban letárolt nagyságot. Ha igen, akkor hibaüzenettel térünk vissza, amit természetesen előzetesen letároltunk a header fájlban, itt csak hivatkoznunk kell rá. A sorváltozót nullázzuk, meghívjuk a sync() függvényt, ezzel nullázzuk a puffereket, valamint az állapotváltozót CMD_STATE_DONE állapotúra módosítjuk.

WCMD_STATE_EXECUTE:

Ez az állapot a W-típusú parancsok feldolgozásának fő része. Első lépésben kiteszünk a képernyőre egy CHAR_CODE_CR és egy CHAR_CODE_LF karaktert. Ez után megvizsgáljuk, hogy a pufferben lévő parancs megegyezik-e a parancsokat tartalmazó tömbben tároltak valamelyikével.

Ha egyik parancsot sem ismertük fel a megadott szövegben, akkor egy előre letárolt strBADCMD (Command not found. Type 'help' to see command list!) szöveget írunk ki a képernyőre, majd nullázzuk a sorváltozót, és futtatjuk a sync() függvényt.

Ha felismertük a *„help”* parancsot, akkor végigmegyünk a tömbön, amely tárolja az utasításokat, és – egy jogosultsági vizsgálat után – kiírjuk a végrehajtható parancsokat a képernyőre.

Ha az eddigi feltételek nem teljesültek, megvizsgáljuk, hogy a beírt parancs jogosultsági szintje alacsonyabb-e a megengedettnél. Ha igen, akkor kiírjuk az ehhez tartozó hibaüzenetet, majd nullázuk a sorváltozót és meghívjuk a sync() függvényt.

```
if(idx == 0) {
        // utasítások;
}
else if(idx == 1) {
        // utasítások;
}
/*
else if(idx == ) {
        utasítások;
}
*/
```

A fenti kódrészlet szemlélteti a parancsok végrehajtását végző rész felépítését. Az indexváltozóban van eltárolva, hogy a parancsot tartalmazó tömbből hányadik egyezett a bemenetről kapottal. Ennek alapján a megfelelő utasításokat végezzük el minden lehetséges parancsra. Ez a feldolgozó rész is bármikor bővíthető, csak a tömbbe kell felvennünk a parancsszót, majd az indexe alapján egy új else if-ágat nyitni neki.

CMD_STATE_DONE:

A cmd_initialize meghívásakor van az állapotváltozó ebben az állapotban, tehát ez az alapértelmezett. Ha a karakterváltozó MCMD_INIT_CHAR értékű, azaz parancs kezdő karakter, akkor M-típusú parancs van a bemeneten, így az állapotváltozót MCMD_STATE_INIT állapotúra tesszük, így elkezdődhet a parancs feldolgozása.

Különben, ha a karakterváltozó nem CHAR_CODE_SPACE, CHAR_CODE_TAB, CHAR_CODE_NULL, CHAR_CODE_CR vagy CHAR_CODE_LF értékű – tehát nem whitespace karakter –, akkor a WCMD_STATE_INIT állapotba kerül az állapotváltozó. Ha whitespace karaktert találtunk, azt ignoráljuk, és a case-ág lefutása után, valamint a fő do-while ciklus befejeztével a következő karakterre lépünk.

A cmd_initialize függvény: Ez egy külső függvény, melyet a main függvényben hívunk meg az inicializálás során. Feladata annyi, hogy az állapotváltozó értékét CMD_STATE_DONE-ra állítja.

A parancsok feldolgozása a következőképpen történik:

A cmd_initialize() függvény meghívása után az állapotváltozó CMD_STATE_DONE állapotban van. Ha a karakterváltozó egy M-típusú parancs nyitókaraktere, akkor az állapotot MCMD_STATE_INIT-re módosítjuk. Különben, ha nem whitespace karakter, akkor W-típusú parancsot kaptunk, úgyhogy WCMD_STATE_INIT állapotra váltunk.

M-típusú parancsnál:

MCMD_STATE_INIT a kezdőállapotunk, ahol különböző utasítások elvégzése után MCMD_STATE_GET_FUNCTION állapotra váltunk. Ennél a státusznál, ha kezdőkarakterrel van dolgunk INIT-re, ha záró karakterrel, akkor EXECUTE-ra, különben MCMD_STATE_GET_ADDRESS-re váltjuk az állapotváltozót. Mivel többféle formában adhatunk ki egy M-típusú parancsot, így minden változó bekérésénél vizsgálnunk kell, hogy zárókarakterhez értünk-e. GET_ADDRESS állapotban az Adress változó kap értéket, ha adott meg a felhasználó. Itt nyitó- és záró karakter vizsgálat után értelemszerűen INIT-re vagy EXECUTE-ra váltunk. Az ezt követő állapotunk a MCMD_STATE_GET_VALUE, ahol hasonló vizsgálatokat végzünk el, a Value változó értékadásával együtt, viszont ha nem jutunk el végkarakterig, MCMD_STATE_ERROR az ezt követő állapot. ERROR esetén egy hibaüzenetet írunk ki a parancs nevével, valamint készenléti (CMD_STATE_DONE) állapotra váltunk.

Ha az előzőekben záró karakterhez jutottunk, akkor végre kell hajtanunk az utasítást. Ez az EXECUTE állapotban történik, ahol, ha jogosulatlanok vagyunk a végrehajtásra, hibaüzenetet kapunk és alapállapotra állunk át. Ha felismerjük valamelyik parancsot, és az végrehajtható, akkor megtörtént a feldolgozás, ellenben itt is ERROR állapotba kerülünk.

W-típusú parancsnál:

A WCMD_STATE_INIT állapotban vagyunk, ahol az első karakter pufferben való elhelyezése után WCMD_STATE_COLLECT értékűre módosítjuk State változónkat. Ebben az állapotban az egész külső ciklus többszöri lefutása alatt a pufferbe gyűjtjük a parancs szövegét. Ha elértünk egy whitespace karaktert, akkor az azt jelenti, hogy vége a bemenetről érkező parancsnak, tehát WCMD_STATE_EXECUTE-ra váltjuk az állapotváltozónkat. Ha a pufferbe való tárolás közben túlléptük volna a megengedett méretet, hibaüzenettel térünk vissza és CMD_STATE_DONE állapotra módosítjuk a State-et. A parancsvégrehajtásnál, tehát WCMD_STATE_EXECUTE állapotban megkeressük, hogy a parancsokat tartalmazó tömbben van-e egyezés az általunk vizsgálttal. Ha nincs, ugyancsak hibaüzenetet írunk ki. Ha felismerjük az utasítást, de annak futtatásához nincs jogosultságunk, az erre írt hibaüzenetet írjuk a képernyőre. Ha jogosultak vagyunk az elvégzésére, akkor a megfelelő utasításokat végezzük el, a parancsnak megfelelően. Bárhogy ért véget a parancsértelmezés, a végén természetesen CMD_STATE_DONE készenléti állapotra állunk át.

4.2.3. Parancsértelmező összefoglalás

Ez a fajta parancsértelmező megoldás tehát a header fájl „objektumorientált” tulajdonságai segítségével és a függvény felépítésével bármely mikrokontrollerhez könnyedén alkalmazható. Igény szerint bármikor új parancsokkal egészíthetjük ki a programunkat. Egy ehhez hasonló értelmezőt mutatok be a következőkben, melyet a motor kezelését végző fájlban használunk.

4.3. Motor parancsértelmező

Az elkészült parancsértelmező program vázát – ahogy az előbbiekben már leírtam – egyszerűen beilleszthetjük más programba. A teljes motorvezérlő alkalmazásba más programozók az általam írt kód felhasználásával egy másik modulba is beépítették a parancsértelmezőt. E programegység működésének alapjait mutatom be a következőkben.

Ha a parancsértelmező modul a *„motor”* parancsot ismeri fel, akkor meghívja a motor modulban lévő parancsértelmező függvényt. Ez a motor_parse függvény, melynek formális paraméterlistája a cmd_parse függvényéhez hasonlóan egy piofile_t mutatóból, valamint egy u8_t típusú jogosultsági szintből áll, visszatérési típusa pedig void.

A parancsértelmező modultól való különválasztás oka, hogy a *„motor”* parancsszó után alparancsokat adhatunk meg, amelyek a léptetőmotorok mozgását fogják vezérelni. A kiadható alparancsok a következők:

- f (forward): előrefelé történő mozgás
- b (backwards): hátrafelé történő mozgás
- a (acceleration): gyorsulás változtatása
- c (clear position): pozíciót tároló változók ürítése
- s (display status): állapot kiíratása
- p (stop): megállítás
- i (information): kért paraméterekből előálló, a program által végrehajtható értékek lekérdezése
- o (move to origo): a motor kezdőpontra mozgatása

4.3.1. Beállítások a függvény elején

A cmd fájlban található cmd_parse parancsértelmező függvényhez hasonlóan itt is a *poll_line()* függvény segítségével adunk értéket egy változónak, amelyben a feldolgozás alatt álló sort tároljuk. Ezt a változót megvizsgáljuk egy feltételben, és ha nem üres, akkor egy cikluson belül átugorjuk az összes TAB vagy SPACE karaktert, míg nem jutunk el az első nem whitespace karakterhez, vagy nem érünk a sor végére.

Ha a sor üres, akkor egy hibaüzenetet írunk a képernyőre, mert nem találtunk alparancsot, a függvény futása pedig a *return* utasításra befejeződik.

Ezek elvégzése után a letárolt karaktert kisbetűssé alakítjuk, amennyiben nagybetűs volt.

4.3.2. Parancsok értelmezése

A kezdő utasítások végrehajtása után if-else if szerkezettel megvizsgáljuk, hogy az előzőekben kapott karakterváltozó melyik alparanccsal egyezik meg.

Forward és Backward, F és B:

Ha a letárolt karakterváltozónk *„f”* vagy *„b”* értékű, akkor motormozgató utasítással van dolgunk. Az állapotváltozó STATE_DONE állapotától függően végezzük el a következő utasításokat. Ha nincs alapállapotban a változó, akkor visszautasítjuk a kiadott parancsot, különben az *„f”* karakterrel előre, a *„b”* karakterrel hátrafelé mozgathatjuk a motort. A karakter után a felhasználónak egy számot kell megadnia, ami jelzi a program felé, hogy a motor hány lépést tegyen meg. A megtett lépésszámot egy változóban tárolja a program. A felhasználó felé mozgás közben a képernyőre a *„Moving…”* üzenetet írjuk ki.

Acceleration, A:

Az előzőhöz hasonlóan ez a parancs is feltételvizsgálattal kezdődik, tehát nem hajtjuk végre, ha az állapotváltozónk nem STATE_DONE értékű. Az *„a”* karakter után a felhasználónak három számot kell megadnia, melyek a következők:

- Indulási frekvencia

- Maximális frekvencia
- Gyorsítási idő (és lassítási idő)

Ha a három közül egy paraméter hiányzik, vagy a paraméterek nem megfelelőek, hibaüzenet küldünk a felhasználó felé. Amennyiben az előző hiba nem lép fel, tehát nem áll meg a futás, a képernyőre a „*Done*” üzenetet írjuk ki.

Clear Position, C:

Amennyiben STATE_DONE állapotban van az állapotváltozónk, ezt a parancsot is végrehajthatjuk. Ha nem STATE_DONE állapotban voltunk, akkor a *„Busy”* üzenetet írjuk a képernyőre. Az előre- és hátrafelé történő mozgások eredményeképpen a motor pozícióját tároló változó értéke folyamatosan változik. Így ha ezt a változót szeretnénk lenullázni, azt ezzel a paranccsal tehetjük meg. A beállítások sikeres végrehajtását a *„Done”* üzenettel jelezzük ezt a felhasználó felé.

Display Status, S:

Ezt a parancsot is csak STATE_DONE állapotban futtathatjuk, ellenben a *„Busy”* szöveget írjuk a képernyőre. Az előző parancsnál leírt motorpozíciót tároló változó értéke a felhasználó számára nem ismert, a parancs feladata ennek az értéknek a kiíratása.

Stop, P:

A Stop parancs kiadásának csak akkor van értelme, ha nem STATE_DONE állapotban vagyunk, hiszen a megállításhoz a motornak mozognia kell. Itt a feltételvizsgálat ellentétes az előzőekkel, csak akkor végezhető el, ha az állapotváltozó nem STATE_DONE állapotban van. Ha a feltétel igaz volt, akkor a kiadott utasítások elvégzik a motor megállítását.

Information, I:

Amennyiben elvégezzük a gyorsulás beállítását, és kiadjuk a motornak a mozgás parancsot, a program kiszámolja, hogy milyen módon kell mozgatni a motort. Ha azonban az

adott paraméterek mellett a mozgás nem valósítható meg, visszautasítja a program annak elvégzését, viszont kiszámítja és eltárolja azokat az értékeket, melyekkel végrehajtható a mozgás, és nem sokban térnek el a felhasználó által megadottól. Ennek a parancsnak a segítségével a felhasználó által megadott nem végrehajtható, valamint a program által számított végrehajtható értékeket adjuk vissza táblázatos formában.

Move to Origo, O:

Ez a parancs egy körpályán mozgó motor vezérléséhez szükséges. Lényege, hogy egy optocsatolót helyeznek el a motor sínje mellett, melynek helyét origónak nevezik ki. A parancs hatására a motornak addig kell mozognia valamely irányba, ameddig az érzékelő nem jelez, hogy a motor elérte a nullpontot. Amennyiben ez megtörténik, meg kell azt állítanunk. Ezt a feladatot végzi el ez az utasítás.

5. Összefoglalás

A szakdolgozat alapjául szolgáló mikrokontroller alapú léptetőmotor-vezérlő program fejlesztésénél egy nagyobb fejlesztési projektbe kapcsolódtunk be. Az elkészült programot a Debreceni Egyetem Orvos- és Egészségtudományi Centrumában található PET-CT Diagnosztikai Kft. használja. Egy CT-berendezés „asztalát” – amelyre egy kísérleti állatot helyeznek – mozgatják a vezérlő program segítségével. Az alkalmazás folyamatos fejlesztés alatt áll, így a szakdolgozat elkészültével is folytatjuk a program további módosítását, és új modulok beépítését.

Mikrokontrollernek nevezzük az egyetlen áramköri lapkára integrált számítógépet. Egy mikrovezérlő tartalmazza a mikroprocesszort, a memóriát (ROM és RAM) és az I/O vezérlőket. Az asztali számítógépek Neumann elveivel szemben a mikrokontrollerek a Harvard-architektúrára épülnek, melynek lényege, hogy a program- és az adatmemória elkülönül.

A léptetőmotorok olyan elektromechanikus eszközök, melyek az elektromos impulzusokat diszkrét mechanikai mozgássá alakítják át. Ezen impulzusok segítségével megadott lépésszámmal tesznek meg egy teljes körülfordulást. Több csoportjuk létezik, ezek az állandó mágneses, a változó reluktancia és ezek ötvözetei, azaz a hibrid léptetőmotorok. Vezérelhetők unipoláris vagy bipoláris módon, attól függően, hogyan vannak bekötve a tekercsek. A lépésszög nagysága, valamint az egy időben áram alatt lévő tekercsek számának alapján beszélhetünk egészlépéses, féllépéses, mikrolépéses, valamint hullámmeghajtásról.

Az elkészült programmal egy Silicon Labs által gyártott C8051F120 kódjelű mikrokontrollert vezérelnek, mely igen „erős” eszköz, sebessége 100 MIPS (Millió utasítás per másodperc).

A teljes program felépítése a modulokra épül, minden vezérlő funkció külön egységben található. A header fájlok két részre bontásával tesszük a modulok egymás közti kommunikációját átláthatóbbá, az objektumorientált programozásból ismert láthatóságok C programnyelvben való értelmezésével.

Az általam fejlesztett modul a vezérlő program parancsértelmező egysége volt, melyet állapotvezérelten kellett megoldanom. Egy header fájlban állapot konstansokat deklaráltam, valamint a szükséges változókat és három függvényt – egy parancsértelmező, egy jogosultságkezelő és egy inicializáló függvényt. Az állapotokat reprezentáló konstansok segítségével a parancsok feldolgozását egy switch-case struktúra használatával végeztem el. Az elkészült parancsértelmező egy olyan univerzális keretprogram, mely bármilyen mikrokontrollerhez felhasználható beépített modulként.

Egy másik programrészlet, ahol az általam fejlesztett parancsértelmezőt használták fel, a motor kezelőjében található parancsértelmező rész, melynél más típusú állapotok, és a motorkezeléshez írt függvények használatával történik a parancsok feldolgozása, majd a hozzájuk tartozó utasítások elvégzése.

A mikrokontrollerek programozása tehát számomra nem ér véget a szakdolgozat befejeztével, hiszen a programot új modulokkal kell kiegészíteni, és a későbbi munkáim során is szeretném folytatni a különböző feladatok mikrovezérlőkkel való megoldását.

6. Irodalomjegyzék

PDF-ek:

- SDCC Compiler User Guide
- Silicon Labs AN155 Stepper Motor Reference Design
- Industrial Circuits Application Note – Stepper Motor Basics

Internetes források:

Magyar:

- http://qtp.hu/elektro/leptetomotor_mukodese.php
- http://www.hun.cncdrive.com/tudasbazis/stp.htm
- http://www.hobbielektronika.hu
- http://hu.wikipedia.org/wiki/Mikrokontroller
- http://itl7.elte.hu/hlabdb/pic/pic.html
- http://eki.sze.hu/ejegyzet/ejegyzet/gyimesi/10.htm

Angol:

- http://en.wikipedia.org/wiki/Microcontroller
- http://www.howstuffworks.com/microcontroller.htm
- http://www.microchip.com/
- http://en.wikipedia.org/wiki/Harvard_architecture
- http://wiki.answers.com
- http://infocenter.arm.com/help/index.jsp
- http://en.wikipedia.org/wiki/Stepper_motor
- http://www.solarbotics.net/library/pdflib/pdf/motorbas.pdf
- http://www.cs.uiowa.edu/~jones/step/
- http://www.silabs.com/Pages/default.aspx

Képek forrása:

- http://en.wikipedia.org/wiki/File:Von_Neumann_architecture.svg
- http://www.elec.canterbury.ac.nz/PublicArea/Staff/hof/p10-embed/p10-tutorial/harvard.gif
- http://www.anaheimautomation.com/images/old-site/stepmtr6.gif
- http://www.anaheimautomation.com/images/old-site/stepmtr5.gif
- http://www.steppermotors.us/images/stepmtr7.gif
- http://qtp.hu/elektro/leptetomotor_mukodese.php
- http://qtp.hu/elektro/leptetomotor_mukodese.php
- http://qtp.hu/elektro/leptetomotor_mukodese.php
- http://media.digikey.com/photos/Silicon%20Labs%20Photos/C8051F120-GQ.JPG

7. Köszönetnyilvánítás

Ezúton szeretném köszönetemet és tiszteletemet kifejezni Szabó Zsolt tanár úrnak, aki megismertette velem a mikrokontrollerek világát, és témavezetőként szakmai felkészültségével és segítőkészségével mindenben támogatott a szakdolgozat készítése során.

Valamint köszönetemet fejezem ki Kovács György szaktársamnak, akivel közös szakdolgozat témánkat kidolgoztuk, és folyamatosan egymás segítségére voltunk a felmerülő kérdésekben.