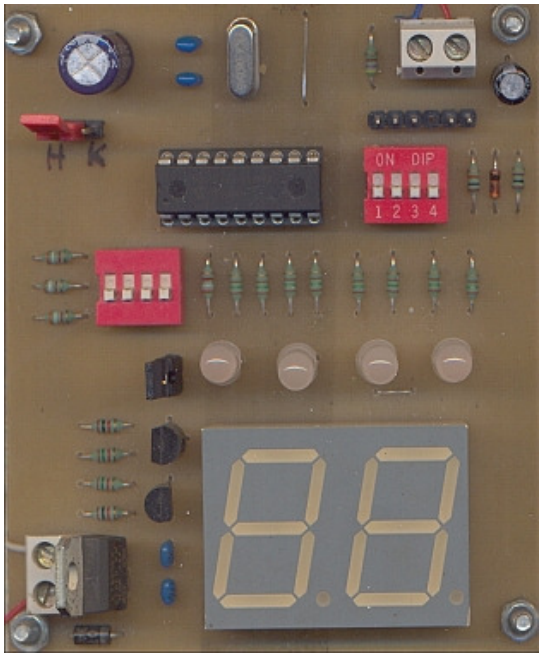
Mikrovezérlők oktatása

Tananyag

Készítette: Juhász Róbert



Tartalomjegyzék

Bevezetés

Ezen munkámmal a mikrovezérlők oktatása során szerzett tapasztalataimat szeretném megosztani az olvasóközönséggel.

A történet hosszú időre nyúlik vissza, valamikor 1999-ben vetődött fel bennem, hogy a hagyományos digitális technika mellett az egyre inkább teret nyerő programozható eszközökkel is kellene foglalkoznom. Sikerült is beszereznie az iskolánknak (Mechatronikai Szakközépiskola és Gimnázium, itt tanítok 1992 óta) egy Siemens mikroprocesszoros fejlesztőrendszert. Ezen kezdtem el tanulni az assembly programozás rejtelmeit. Nem sokkal ezután olvastam egy felhívást középiskolás diákok számára, amelyben mikrovezérlő programozó versenyt hirdettek. Ekkor még nem igazán tudtam, hogy mi is az a mikrovezérlő, miben más, mint a mikroprocesszor. Amikor azonban utánanéztem, rájöttem, hogy éppen ez az, amit kerestem. A mikrovezérlő ugyanis egy egytokos mikroszámítógép, tehát egyesíti magában a mikroprocesszort, a memóriát és a perifériákat. Ez a megoldás nagymértékben leegyszerűsíti a programfejlesztést. Néhány lelkes diákot magam köré gyűjtve elkezdtük együtt tanulni a programozást. Sikerességünket jelzi, hogy az egyik tanítványom megnyerte Miskolcon az országos programozó versenyt.

Ekkor fogalmazódott meg bennem a gondolat, hogy ezt sok mindenkinek meg lehetne tanítani, ez a mai világban nagyon hasznos tudomány, hiszen ilyen mikrovezérlők találhatók ma már a hétköznapi berendezésekben is (televízió, mikrohullámú sütő, stb.). Mivel a mikrovezérlők oktatása nem kapcsolódik szorosan az iskolánkban oktatott tárgyakhoz (habár az utóbbi időben próbálunk egyéb tantárgyakon belül időt szakítani rá), ezért egy szakkört indítottam el.

Az elsődleges célom az volt, hogy olyan segédeszközöket kutassak fel, illetve készítsek, amelyek a tanulók számára könnyen hozzáférhetőek, otthon is megépíthetőek legyenek. A mikrovezérlő programfejlesztéséhez a Microchip ingyenesen hozzáférhetővé tette az MPLAB programját. A beégetéshez (felprogramozáshoz) is sikerült két ingyenes programot találnom. A szükséges hardver elemek saját készítésű, olcsó berendezések. A két szükséges dolog tehát az égető panel és az úgynevezett „próbapanel” mindenki számára egyszerűen megépíthető. A programozáshoz szükséges ismerni az adott

mikrovezérlőt, azonban a leírások csak angol nyelven elérhetők, ezért az eredeti Microchip adatlapról készítettem egy 35 oldalas fordítást.

A könnyű (akár otthonról való) hozzáférhetőség érdekében készítettem egy honlapot, ahová ezeket, a hasznos információkat (programok, kapcsolási rajzok, nyáktervek, beültetési rajzok, alkatrészjegyzékek, fordítások, mintaprogramok) felraktam.

A következőkben bemutatom, hogyan tudjuk a mikrovezérlő programozásának, alkalmazásának alapjait tanulóinkkal elsajáttíttatni, ezzel is szélesítve, gazdagítva az iskola tevékenység repertoárját.

1. Mikrovezérlők története

A villamos szakmák az elmúlt 50 év során óriási fejlődésen mentek keresztül. A fejlődés olyan mértékű volt, amit korábban el sem tudtunk képzelni. Éppen ezért nagyon fontossá válik az oktatás előkészítő funkciója. Olyan ismeretek befogadására kell felkészítenünk a tanulókat, amikről nem tudjuk, hogy mi lesz fejlődésük következménye, erről legfeljebb csak megérzéseink lehetnek. A szakmában tanító pedagógusoknál előtérbe kerül a szinte permanens szakmai megújulás. Sohasem felejtem el azt az élményemet, amikor szakközépiskolás koromban először találkoztam a LED-del, ekkor lehetet először kapni Miskolcon ezt az alkatrészt. A mai gyerekeknek ez már teljesen természetes. Nagy lelkesedéssel készítettük az első LED-es astabil multivibrátorunkat. A kollégiumi szobában marattuk ki a nyákot. Mivel fúrógépünk nem volt a forrpontokat szöggel lyukasztottuk ki, és egy csehszlovák gyártmányú „pillanatpákával" forrasztottuk be az alkatrészeket. Az alkotás öröme nagy boldogsággal töltött el mindannyiunkat.

A fejlődés során az egyik fontos mérföldkő az elektronikus számítógép feltalálása volt. Az ENIAC (Electronic Numerical Integrator And Computer) a Manhattan-terv keretében készült. A gépet a Pennsylvania egyetemen építették, a munkát 1946-ban fejezték be. A munkában oroszlánrészt vállalt Neumann János. Ma is az általa kialakított elven működnek az asztali számítógépek. Olyan előrelátó és humánus volt, hogy elvét sosem engedte szabadalmaztatni. Azért, hogy ezt megakadályozza egy publikációban nyilvánosságra hozta a Neumann-elméletet, ami a szabadalmi védettséget lehetetlenné tette. Az ENIAC 18 ezer elektroncsövet tartalmazott, több mint 100 kW elektromos energiát fogyasztott és 450 m^2 helyet foglalt el (több mint 30 m hosszú termet építettek az elhelyezéséhez). A gép tömege 30 tonna volt, megépítése tízmillió dollárba került. Három nagyságrenddel gyorsabb volt, mint a relés számítógépek: az összeadást 0,2 ms, a szorzást 3 ms alatt végezte el. A programja azonban fixen be volt huzalozva a processzorba és csak a villamos csatlakozások átkötésével lehetett megváltoztatni. Az elektroncsövek megbízhatatlansága miatt a gép csak rövid ideig tudott folyamatosan működni.

Az ENIAC-ot ballisztikai és szélcsatorna-számításokra használták. Egy trajektória kiszámítása a gépnek 15 másodpercig tartott, ugyanez egy szakképzett embernek asztali kalkulátorral 10 órás munka volt.

A fejlődés következő állomását a W. Schockley, W. H. Brattain és J. Bardeen által Bell laboratóriumában feltalált tranzisztor szolgáltatta. A tranzisztort 1947-ben fedezték fel, és 1948-ban publikálták. A találmány jelentőségét mutatja, hogy 1976-ban Nobel-díjat kaptak a feltalálói. A tranzisztor kis méretével és fogyasztásával, valamint hosszú élettartamával hamar kiszorította az elektroncsöveket. Megjelentek az első telepes, hordozható készülékek. A tranzisztor feltalálásával megjelentek a második generációs számítógépek. A kis méret lehetővé tette a nyomtatott áramkörök kialakítását. Az első nyomtatott áramkör 1958-ban jelent meg. Ez a megoldás kiküszöböli a vezetékes összekötésekből származó hibákat, nagymértékben növelve ezzel az áramkör megbízhatóságát.

Az integrált áramkörök megjelenése továbbvitte ezt a fejlődést. Az integrált áramkörben ugyanis egy hordozó szilícium lapkán egyetlen gyártási folyamatban alakítják ki az alkatrészeket, és az ezeket összekötő vezetőpályákat. Egyes szakemberek az integrált áramkörök feltalálásának jelentőségét a könyvnyomtatáséhoz mérik. A tetszés szerinti darabszámban, több nagyságrenddel olcsóbban és nagyságrendekkel megbízhatóbb minőségben előállítható elemek az élet minden területén nagy változásokat idéztek elő. Az első integrált áramkört 1959-ben készítették, de csak 1962-ben került kereskedelmi forgalomba. Az integrált áramkörök hatalmas fejlődést produkáltak (kezdve) a néhány alkatrészt tartalmazó SSI (Single Scale Integration) áramköröktől a több millió tranzisztort tartalmazó nagymértékben integrált VLSI (Very Large Scale Integration) áramkörökig. Gábor Dénes a nagy integráltsági fokú technológiát a XX. század legfontosabb találmányai között említi.

Ezek az integrált áramkörök vezettek a harmadik generációs számítógépek kialakulásához. A tranzisztorokat kiszorítják az integrált - egyelőre az alacsony és közepes integráltsági fokú - áramkörök. Megváltozik a gépek struktúrája, átalakulnak funkcionális egységei, s kialakul a fejlett, egységes csatornarendszer, amely közvetlen kommunikációs kapcsolatot biztosít az egységek között.

Az 1971-es esztendő fordulatot hoz a számítógépek történetében, az Intel cég piacra dobja az első mikroprocesszort 4004-es néven. Ez a processzor 4 bit széles adatokkal és 8 bit széles utasításokkal dolgozott. Külön adat (1kB) és programmemóriával (4kB) rendelkezett. A 4004 46 utasítást ismert, 2300 tranzisztort tartalmazott és 16 lábú DIP tokba szerelték. 1972-ben kiadták a kibővített változatát 4040 néven. Az utasításkészletét és a memóriáját megnövelték az előzőhöz képest. Még ebben az évben megjelenik az immár 8 bites 8008-as mikroprocesszor.

A Texas Instruments az Intel 4004/4040 processzorát követve kiadja a 4 bites TMS 1000-t. Az 1974-ben megjelent TMS 1000 volt az első mikrovezérlő. Ez egyetlen tokba építve tartalmazott adatmemóriát (RAM), programmemóriát (ROM) és I/O egységet, lehetővé téve a működést külső kiegészítő áramkörök nélkül. A mikrovezérlő tartalmazott egy 4 bites akkumulátort, egy 4 bites Y és egy 2-3 bites X regisztert, amellyel a belső 64, illetve 128 félbájtos RAM-ot lehetett megcímezni, valamint egy 1 bites státuszregisztert. A 6 bites programszámláló kombinálva egy 4 bites lapozó regiszterrel és opcióként 1 bankváltó bittel 1, illetve 2kB-os memória (ROM) címzését tette lehetővé. A szubrutinok, elágazások kezelésére tartalmazott egy 6 bites vermet és egy 4 bites lapozó puffert. Érdekessége a programszámlálónak, hogy nem számláló, hanem egy visszacsatolt léptetőregiszter volt, így az utasítások a memóriában nem egymás után sorban következtek. Utasításkészlete úgy épült fel, hogy tartalmazott tizenkét 8 bites huzalozott (fix), valamit 31 darab 16 bites felhasználó által mikroprogramozott utasítást, amelyet egy PLA valósított meg. A hardveresen huzalozott utasítások végrehajtási ideje 1 gépi ciklus volt, megszakítási lehetőséggel nem rendelkezett.

Szintén 1974-ben jelenik meg a 8008 továbbfejlesztése a 8080-as Intel mikroprocesszor, amely 8 bites adatbuszt és 16 bites címbuszt tartalmazott. A belsejében hét 8 bites regisztert (A-E, H, L), tartalmazott, amelyekből a BC, DE, és HL összekapcsolhatók voltak 16 bitessé. A 16 bites veremmutatóval egy 8 mélységű verembe lehetett elmenteni a visszatérési címeket. A 8080-as processzor volt az első széles körben elterjedt személyi számítógép az Altair 8800-as „agya”. Az Intel később ezt tovább fejlesztette, és 1976-ban kiadta a 8085-ös processzort. Ebbe beépítettek két új utasítást, amellyel a három

szintén új fejlesztésként megjelent megszakítási vonalat lehetett tiltani vagy engedélyezni, valamint kapott soros vonali kivezetéseket is. A hardver is egyszerűsödött, az új processzor már csak egyetlen +5V-os tápfeszültséget igényelt.

A Zilog cég 1976-ban kiadja, a sokáig nagy népszerűségnek örvendő Z-80-as processzort, ami a 8080-as továbbfejlesztésének tekinthető (volt Intel mérnökök tervezték). A Z80-as 8 bites adatbusszal és 16 bites címbusszal rendelkezett és tudta futtatni a 8080as összes utasítását, amelyet kibővítettek 80 új utasítással (1, 4, 8 és 16 bites műveletek, páros című blokkmozgató és I/O utasítások). A regisztertömböt két bankra osztva megduplázták, amely növelte a sebességet. A Z-80-ba beépítettek két indexregisztert (IX, IY), valamint kétszintű megszakítási rendszert kapott. Az eredeti Z-80 órajelfrekvenciája 2,5MHz volt a Z80-H típusé 8MHz, a CMOS verzióé (Z80-C) pedig 10MHz.

Nem sokkal a 8080 kiadása után 1975-ben a Motorola bemutatta 6800-as processzorát. Néhány Motorola tervező kilépve a cégtől a MOS Technologies-nél kezdett el dolgozni (később a Commodore megvásárolta), és itt jött létre a 650x sorozat. Ebbe a sorozatba tartozott a 6501 (lábkompatíbilis a 6800-as processzorral) és a 6502 (a korai Commodore, Apple és Atari gépekben használták). A 6800-as változatai közül a 6510-et a Commodore 64-ben, a 6507-et pedig az Atari 2600-ban alkalmazták. A 650x sorozat az úgynevezett „little endian” technikát használta (a cím alsó bájtját hozzá lehetett adni az indexregiszterhez, miközben a felső bájtot lehívtuk), utasításkészlete teljesen eltért a 6800-tól. Az ára mintegy negyede volt a 6800-hoz képest (kevesebb, mint 1000 dollár). Ez az alacsony ár tette lehetővé, hogy a korai személyi számítógépek processzorává váljon. 1977-ben a Motorola jelentősen továbbfejlesztette a 680x-es sorozatot, és kiadta a 6809-es processzort. A 6809-es két 8 bites akkumulátort (A és B) tartalmazott, amit lehetett egyetlen 16 bites regiszterként (D) használni. A processzor két indexregisztert (X, Y) és két veremmutatót (S, U) tartalmazott, amivel nagyon sokrétű címzési módot lehetett megvalósítani. A 6809 64kB memória kezelésére volt alkalmas. További jellegzetessége, hogy ez volt az első 16 bites aritmetika, ami tartalmazott szorzó utasítást.

Az Advanced Micro Devices (AMD) is színre lép az Am2901-es processzorával. Ez egy úgynevezett „bitszelet” mikroprocesszor, ami azt jelenti,

hogy felépítése moduláris: tetszőleges szélességű (nx4bit általában) ALU építhető, tetszőleges utasításkészlet alakítható ki. Az Am2901 4-es bitszelet processzor volt 16 regiszterrel és 4 bites ALU-val. A vezérlése mikroprogramozott volt, a belső ROM-ból olvasódtak ki az egyes utasítások végrehajtásának lépései. Később az Am2903 már hardveres szorzót is tartalmazott. Az AMD alkotta meg elsőként a lebegőpontos műveletek elvégzésére alkalmas matematikai társprocesszort. Az AMD9511 1979-ben készült el. 32 bites lebegőpontos műveleteket tudott végezni. A 16 bites ALU képes volt összeadásra, kivonásra, szorozásra, osztásra, szinusz és koszinuszt számolására, műveletvégzési sebessége minden akkori processzornál nagyobb volt.

1977-ben az Intel is elkészíti a maga mikrovezérlőjét, a 8048-at, ami alacsony árával és kis méretével igen jó „ötletnek" számított. Az adatmemória a tokon belül volt kialakítva, a programmemóriát viszont kívülről kellett csatlakoztatni (Harvard-architektúra volt, bár a program és az adat ugyanazt a címvonalat használta). A 8048-at felváltotta a 8051 és a 8052, amely már beépített programmemóriát (ROM) tartalmazott. A 8051-es már rugalmas, két bájt széles utasításkészletet tartalmazott 8 bit széles regiszterekkel és akkumulátorral. Az adatmemória 128 bájtos volt, amit direkt vagy indirekt regiszteres címzéssel lehetett elérni. Ezen kívül felette tartalmazott még egy 128 bájtos részt, amit a 8052-nél csak indirekt lehetett elérni (veremként használták). Külső memóriát a 8048-hoz hasonlóan el lehet érni közvetlenül (256 bájtos lapokban az I/O portokon keresztül), vagy a 16 bites DPTR címregiszterrel. A közvetlenül elérhető adatterület feletti 32 hely bitenként címezhető. Az adat és a programmemória azonos címterületen osztozik (a címvezetékek is, ha külső programmemóriát alkalmazunk). Habár komplikált ez a memóriaszervezés, mégis rugalmas beágyazott tervezést enged meg. Népszerűségét bizonyítja, hogy 1988-ig több mint 1 billiót adtak el belőle.

A PIC mikrovezérlők gyökere a Harvard egyetemre (Harvard-architektúra) nyúlik vissza, a Védelmi Minisztérium projektjének keretében készült. A Harvard-architektúrát először a Signetics 8x300-ban használták, majd a General Instruments adaptálta és a periféria illesztő vezérlőiben (peripheral interface controller, azaz PIC) alkalmazta. A gyártás később (1985) az arizoniai Microchip Technology-hoz került, s a PIC lett a cég fő gyártmánya. A PIC

mikrovezérlők típusválasztéka mára igen széles lett: PIC10x, PIC12x, PIC14x, PIC16x, PIC18x, PIC24x. Ezekben a sorozatokban szinte minden alkalmazásra találhatunk megfelelő mikrovezérlőt. A 12-es sorozat például nagyon egyszerű mikrovezérlő, 6 I/O lábat, 25-128 regisztert, és egy beépített 8 bites számláló/időzítő modult tartalmazott. A 18-as sorozat pedig 16-70 I/O lábat, 256-3936 regisztert, több 8/16 bites számláló/időzítőt, számos perifériát: A/D átalakító, címezhető szinkron/aszinkron soros port, SPI és I^2C busz, összehasonlító/kiolvasó/PWM modul, analóg komparátor, stb. tartalmaz. A Microchip nagyon jó kézikönyveket és alkalmazási leírásokat bocsát a felhasználók számára (ezek természetesen angol nyelvűek). A fejlesztéseket mindig úgy végzi, hogy előtte kikéri a felhasználók véleményeit, s ennek alapján készül el a következő generációs PIC. Ezen mikrovezérlők ára nagyon kedvező, 500-2500Ft között mozog, a tanulók otthoni felhasználásra is megvásárolhatják nagyobb anyagi ráfordítás nélkül. A Microchip a programfejlesztéshez szükséges MPLAB programot ingyen biztosítja a felhasználóknak.

A dolgozat a PIC16F84-es mikrovezérlő oktatásának bemutatását, az ehhez szükséges programok leírását, a szükséges hardverelemek elkészítésének teljes dokumentációját tárgyalja. Mindenkinek sok sikert kívánok a mikrovezérlők programozásának és oktatásának elsajátításában.

2. Az oktatáshoz szükséges segédeszközök

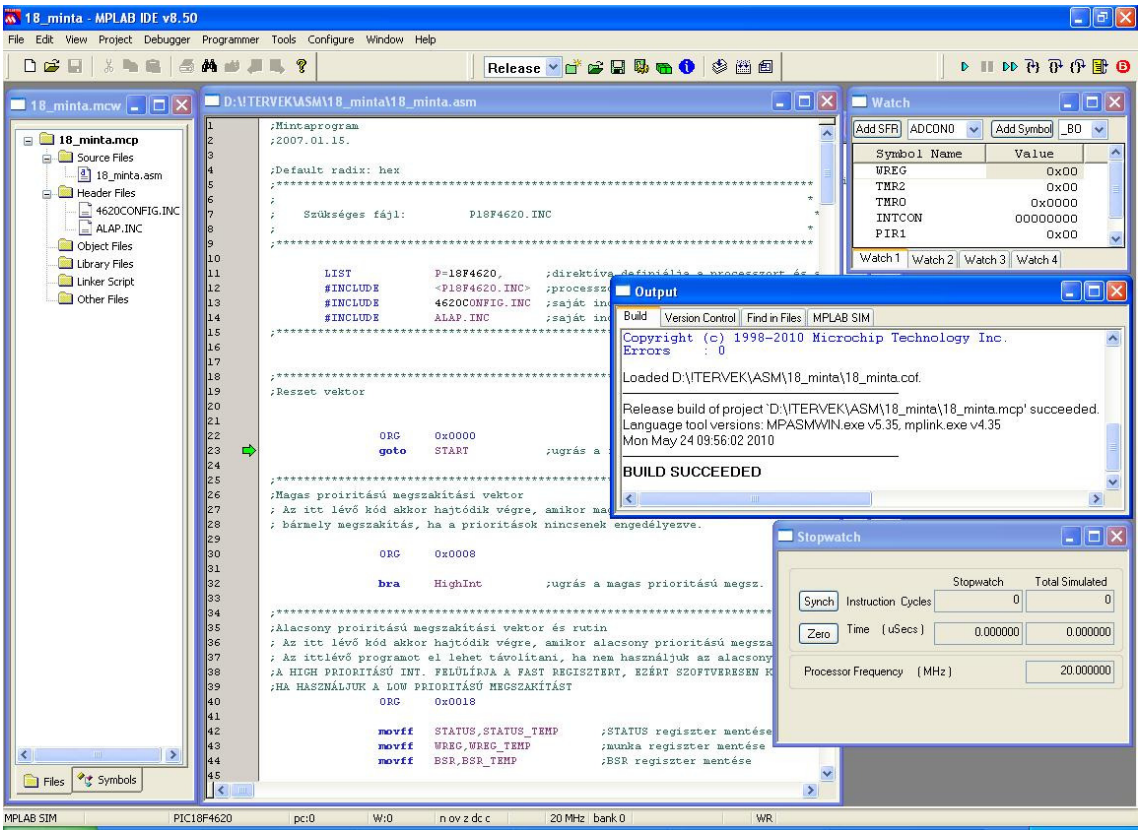
A mikrovezérlők programozásának oktatásához szükségünk van néhány segédeszközre, amelyeket ebben a fejezetben ismertetek. Az egyes hardverelemek részletes leírása (nyákterv, beültetési rajz, alkatrészjegyzék, stb.) a mellékletben található, ezek alapján bárki utána építheti.

A szükséges elemek:

- Fejlesztő program (MPLAB)
- Programozó szoftver (PICALL, IC-PROG, WinPic800)
- Programozó készülék (saját készítésű)
- Próbapanel (saját tervezésű)

2.1 Az MPLAB program

A fejlesztőkörnyezet legfontosabb szoftverelemét az MPLAB program (1. ábra) adja, amelyet a Microchip cég (www.microchip.com) ingyenesen biztosít



1. ábra

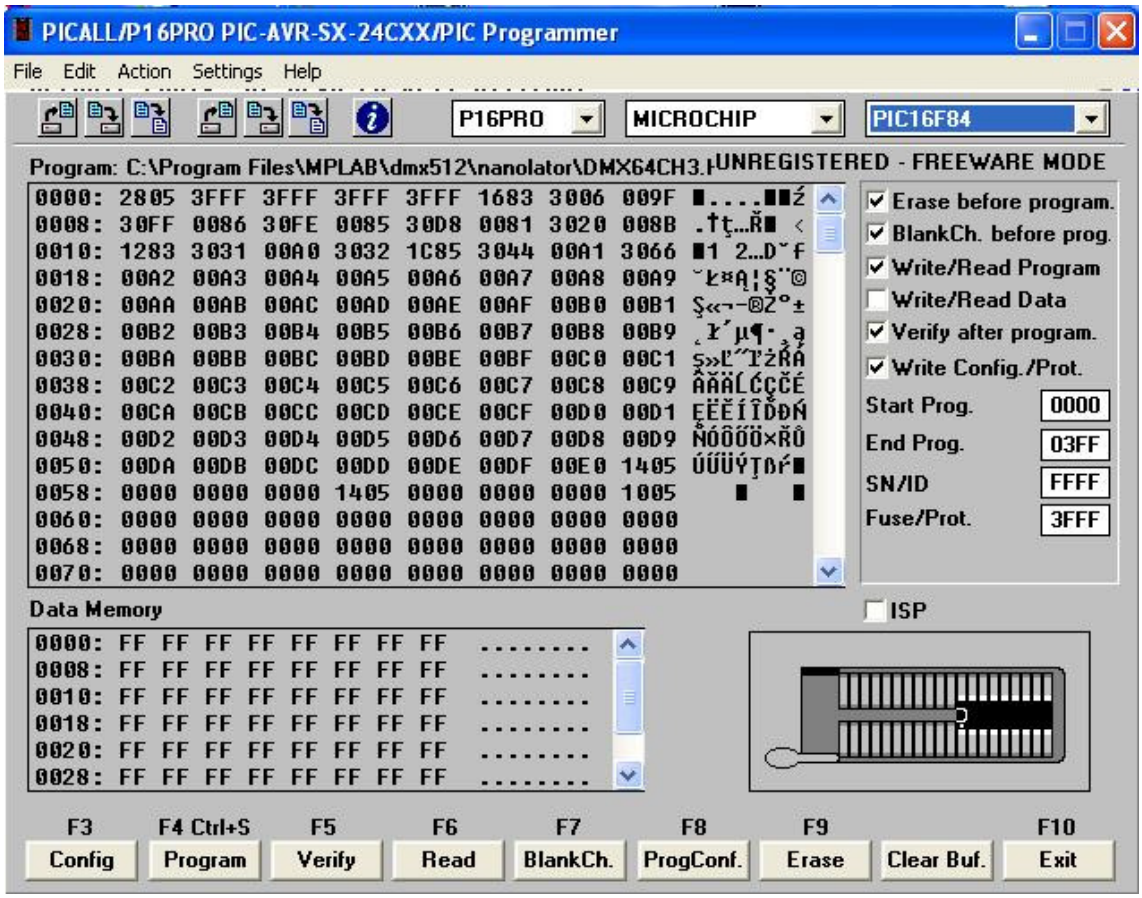
az általa gyártott eszközök szoftverfejlesztéséhez. Ez a program tartalmaz egy szövegszerkesztőt a programok megírásához, egy keresztfordítót a

forrásprogramok gépi kóddá alakításához, valamint egy beépített szimulátort a programok teszteléséhez bizonyos korlátokkal természetesen.

A program letölthető a Microchip honlapjáról, vagy a saját honlapomról (http://plc.mechatronika.hu). A program folyamatosan frissül, hiszen újabb és újabb mikrovezérlők kerülnek forgalomba. Jelenleg a szoftver a 8.50-es verziónál tart. A 8-as MPLAB használatában nincsenek érdembeli különbségek az egyes alverziók között.

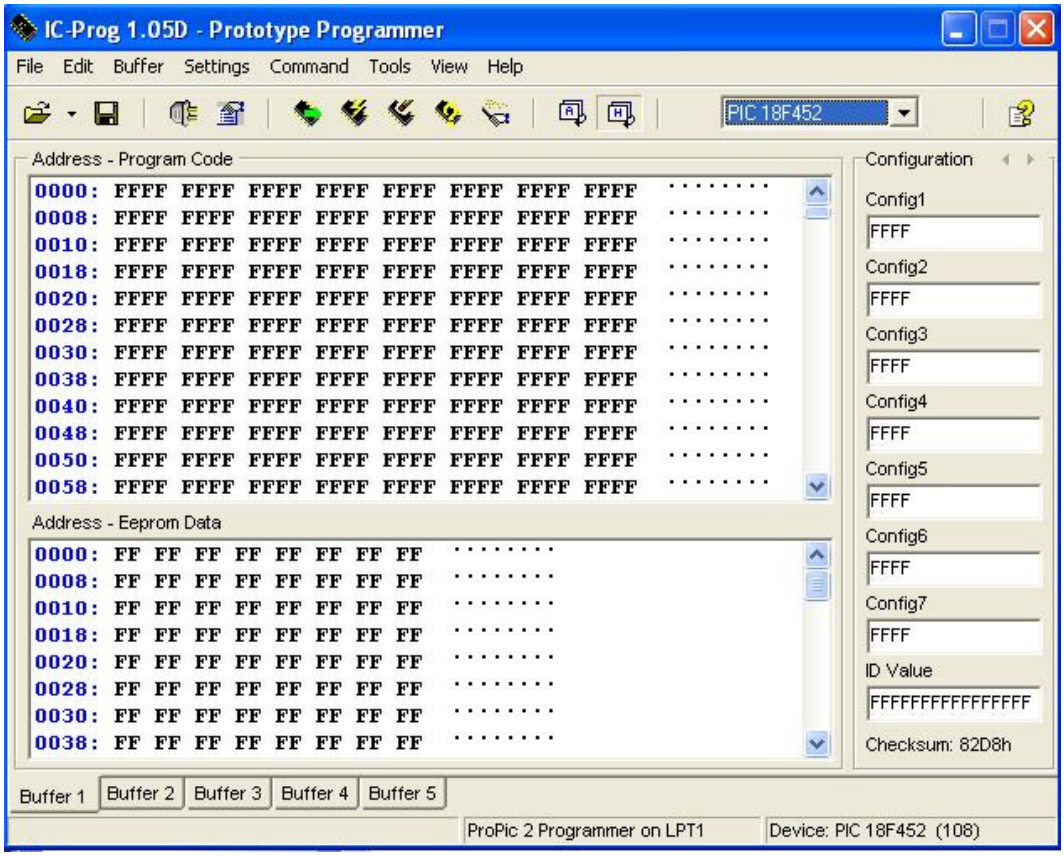
2.2 A programozó szoftver

A megírt és gépi kódra lefordított programjaink mikrovezérlőbe való

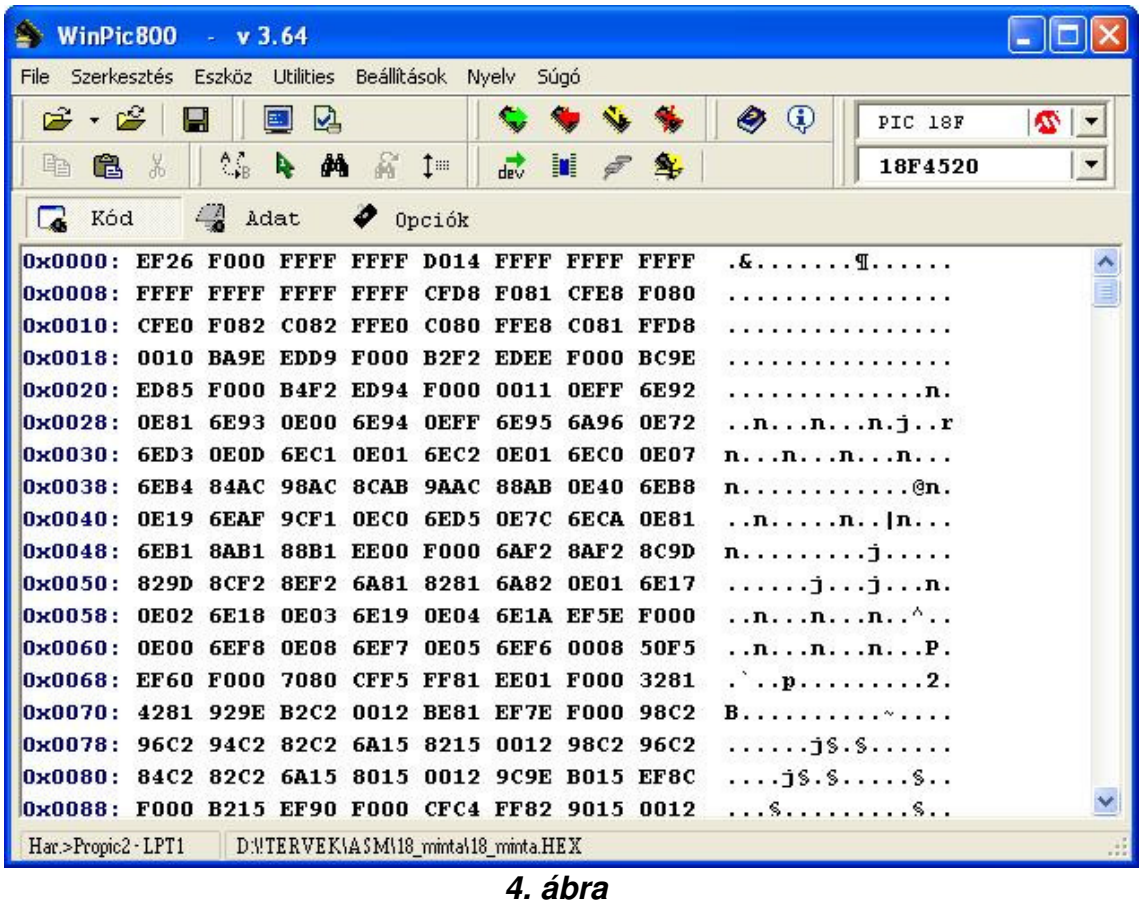


2. ábra

beégetésére szolgálnak ezek a programok. Nagyon sok ilyen ingyenesen hozzáférhető program található. Az egyik ilyen program a PICALL (2. ábra) - www.picallw.com -, a másik pedig a 3. ábrán látható IC-PROG (www.ic-prog.com).


IC-Prog 1.05D - Prototype Programmer
File Edit Buffer Settings Command Tools View Help
PIC 18F452
Address - Program Code
0000: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ....
0008: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ....
0010: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ....
0018: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ....
0020: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ....
0028: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ....
0030: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ....
0038: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ....
0040: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ....
0048: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ....
0050: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ....
0058: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ....
Address - Eeprom Data
0000: FF FF FF FF FF FF FF FF ....
0008: FF FF FF FF FF FF FF FF ....
0010: FF FF FF FF FF FF FF FF ....
0018: FF FF FF FF FF FF FF FF ....
0020: FF FF FF FF FF FF FF FF ....
0028: FF FF FF FF FF FF FF FF ....
0030: FF FF FF FF FF FF FF FF ....
0038: FF FF FF FF FF FF FF FF ....
Configuration
Config1
FFFF
Config2
FFFF
Config3
FFFF
Config4
FFFF
Config5
FFFF
Config6
FFFF
Config7
FFFF
ID Value
FFFFFFFFFFFFFFFF
Checksum: 82D8h
Buffer 1
Buffer 2
Buffer 3
Buffer 4
Buffer 5
ProPic 2 Programmer on LPT1
Device: PIC 18F452 (108)

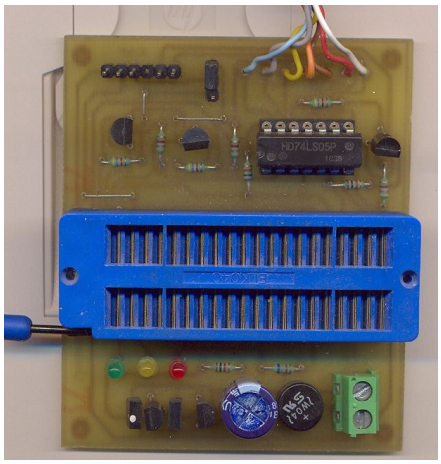
3. ábra


WinPic800 - v 3.64
File Szerkesztés Eszköz Utilities Beállítások Nyelv Súgó
PIC 18F
18F4520
Kód
Adat
Opciók
0x0000: EF26 F000 FFFF FFFF D014 FFFF FFFF FFFF .&.....¶.....
0x0008: FFFF FFFF FFFF FFFF CFD8 F081 CFE8 F080 ....
0x0010: CFE0 F082 C082 FFE0 C080 FFE8 C081 FFD8 ....
0x0018: 0010 BA9E EDD9 F000 B2F2 EDEE F000 BC9E ....
0x0020: ED85 F000 B4F2 ED94 F000 0011 0EFF 6E92 ....n.
0x0028: 0E81 6E93 0E00 6E94 0EFF 6E95 6A96 0E72 ..n...n...n.j..r
0x0030: 6ED3 0E0D 6EC1 0E01 6EC2 0E01 6EC0 0E07 n...n...n...n...
0x0038: 6EB4 84AC 98AC 8CAB 9AAC 88AB 0E40 6EB8 n.............@n.
0x0040: 0E19 6EAF 9CF1 0EC0 6ED5 0E7C 6ECA 0E81 ..n.....n..|n...
0x0048: 6EB1 8AB1 88B1 EE00 F000 6AF2 8AF2 8C9D n.........j.....
0x0050: 829D 8CF2 8EF2 6A81 8281 6A82 0E01 6E17 ......j...j...n.
0x0058: 0E02 6E18 0E03 6E19 0E04 6E1A EF5E F000 ..n...n...n..^..
0x0060: 0E00 6EF8 0E08 6EF7 0E05 6EF6 0008 50F5 ..n...n...n...P.
0x0068: EF60 F000 7080 CFF5 FF81 EE01 F000 3281 .`..p.........2.
0x0070: 4281 929E B2C2 0012 BE81 EF7E F000 98C2 B..........~....
0x0078: 96C2 94C2 82C2 6A15 8215 0012 98C2 96C2 ......j$.$......
0x0080: 84C2 82C2 6A15 8015 0012 9C9E B015 EF8C ....j$.$.....$..
0x0088: F000 B215 EF90 F000 CFC4 FF82 9015 0012 ...$.........$..
Har.>Propic2 - LPT1
D:\TERVEK\ASM\18_minta\18_minta.HEX

4. ábra

Mindkét program egyszerűen kezelhető, mindegyiknek van előnye, illetve hátránya a másikkal szemben. A PICALL nagyon gyorsan „éget”, sokféle mikrovezérlőt ismer, az IC-PROG lassabb, viszont nagyon sokféle hardverrel használható. Megemlíthetjük még a WinPic800 (www.winpic800.com) szoftvert is, amely szintén nagyon jól használható (4. ábra).

2.3 A programozó készülék

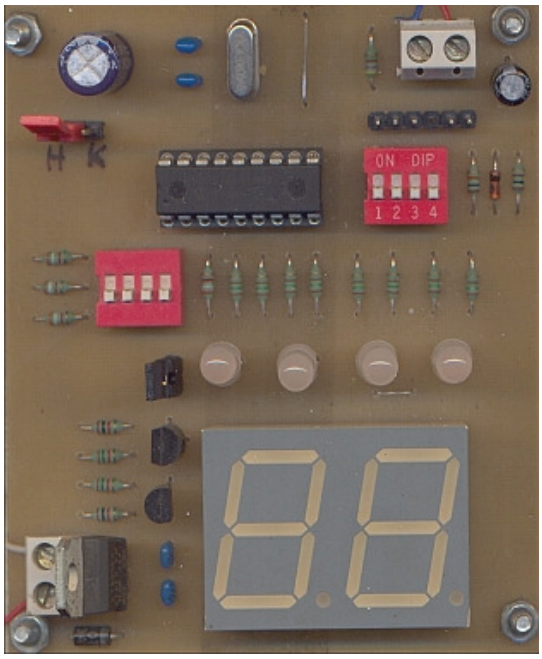


5. ábra

A „beégetésre” a szoftveren kívül szükség van egy programozó készülékre is (5. ábra). Igyekeztem olyan kapcsolást keresni, amely olcsón és egyszerűen kivitelezhető. A választásom egy jól kiforrott kapcsolásra esett, amelyet az eredetileg David Thait publikált és Bojan Dobaj fejlesztett tovább. Ez alapján készítettem el – olcsó alkatrészekből – a programozó nyomtatott áramköri tervét és magát a panelt. Akik „profibb” égetőt szeretnének, azok számára ajánlom a honlapomon megtalálható átdolgozott ICD2 és PICkit2 programozót. Ezek a készülékek nem csak programozásra, hanem hibakeresésre – debug – is alkalmasak. A másik nagy előnyük, hogy nem kell hozzá külön szoftver, az MPLAB programból is egy mozdulattal elindíthatjuk a programozást, vagy hibakeresést. A másik nagy előnye a programozóknak, hogy a számítógép USB portjára csatlakoztatható (sajnos a soros és a párhuzamos portok eltűntek a legtöbb PC-ről). Az USB-s verziónak megvan az az előnye is, hogy nem igényel külső tápegységet. Ezen kívül, ha az áramkörön belüli programozást használjuk – ICSP csatlakozó – akkor még a céláramkörnek sem kell külső tápfeszültség, mert az ICD2, illetve a PICkit2 képes táplálni azokat.

2.4 A próbapanel

Természetesen mindenki szeretné élőben is látni munkája gyümölcsét, ezért célszerű készíteni egy ún. próbapanelt. A programok futását lehet ugyan szimulálni, de a valós működéssel ezt össze sem lehet hasonlítani! A szimulátorban ráadásul nagyon sok minden nem úgy viselkedik, ahogy a való életben. Arról nem is beszélve, hogy a tanulóknak az nyújtja a legnagyobb sikerélményt, ha a valóságban is látják munkájukat működni! Az általam tervezett próbapanel fényképe a 6. ábrán látható. A próbapanel egyszerűen elkészíthető, csekély alkatrészigényű áramkör. Ennek ellenére igen sokoldalúan felhasználható. Találhatóak rajta kapcsolók, kétszínű LED-ek, és hétszegmenses kijelzők. Az első generációs próbapanelból két verzió készült el. A második verzióban cserére került a hétszegmenses kijelző, mert a régi típust már nem lehetett kapni. Emellett rákerült az ICSP csatlakozó is, így nem kell kivenni a programozáshoz a PIC-et.



6. ábra

3. Mikrovezérlők programozásának oktatása

Ez a fejezet részletesen tárgyalja a mikrovezérlők oktatásának menetét lépésről lépésre. Bemutatja az eszközök kezelését, a szoftverek használatát, a programozás fogásait, stb. Az oktatást egy konkrét mikrovezérlőn, a PIC16F84-en keresztül ismerteti a fejezet. Ez az alapok elsajátítására kitűnően megfelel. Igyekeztem kis lépésekben haladni, hogy a kezdők számára is jól érthető és világos legyen a leírás. Fontos szempont, hogy mindig olyan foglalkozást tartsunk, ami sikerélményhez juttatja a tanulókat. Minden alkalomkor legyen valamilyen kézzel fogható gyakorlati feladat is. Később, amikor már tudásuk elmélyült, bonyolultabb feladatoknál csapatmunkában is dolgoztathatjuk tanulókat, illetve versenyeztethetjük is őket egy-egy feladat megoldásával kapcsolatban. A jó munkát mindig jutalmazzuk valamilyen formában!

Fontos kihangsúlyozni, hogy mindig ellenőrizzük a berendezéseink – számítógép, programok, programozó készülék, stb. – működőképességét. A mintaprogramokat mindig próbáljuk ki működés közben is. Nincs kínosabb annál ugyanis, ha nekünk sem működnek a dolgok. Apró lépésekben haladjunk, ne akarjunk egyszerre sok dolgot megtanítani a gyerekeknek. Hagyjuk a tanulókat önállóan is tevékenykedni, ne adjunk nekik mindent készen, mert ez nem fejleszti a konstruktív gondolkodást. Egy-egy adott problémára nagyon sokféle algoritmus gyártható, ezért legyünk felkészültek mindig az adott témában, mert ha a tanulók elakadnak valamiben, akkor nem tudunk nekik segíteni (nem tudunk az ő fejével gondolkodni).

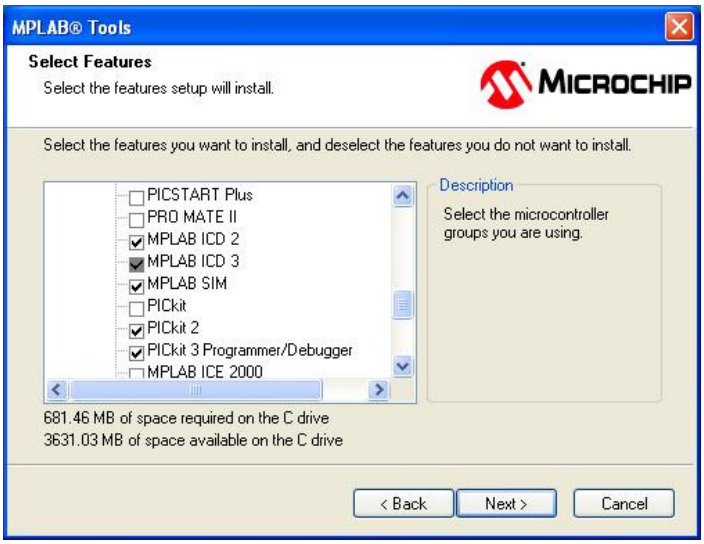
3.1 Bevezető foglalkozás

Az első foglalkozás célja a fejlesztőeszközök, valamint az MPLAB program bemutatása.

Célszerű azzal kezdeni munkánkat, hogy a használni kívánt eszközöket bemutatjuk a tanulóknak. A bevezetőben kitérhetünk a mikrovezérlők történetére, fejlődésének lépcsőire, alkalmazhatóságának sokszínűségére.

A programok megírására, lefordítására, szimulálására az MPLAB programot használjuk. Az ilyen „mindent az egyben” programot integrált fejlesztőkörnyezetnek (Integrated Developing Environment - IDE) nevezik Jelen

esetben az 5.7-es verziót ismertetem, annak ellenére, hogy nem ez a legújabb verzió. Természetesen ennek nem célja, hogy az összes funkció bemutatásra kerüljön, csak a leglényegesebbeket ismerttetem, illetve a funkciók bemutatása igény szerint bővül a feladatokhoz igazodva. Az MPLAB-ban végzett munkákkal kapcsolatban először két fontos fogalmat tisztázzunk a tanulókkal: a *projekt*et és a *munkakörnyezet*et (workspace).



7. ábra

A *projekt* koncepció azt jelenti, hogy az adott feladathoz tartozó fájlokat egy úgynevezett projektfájlban tartjuk nyilván. Ha ezt a fájlt megnyitjuk, a nyilvántartott fájlok a fejlesztéskor azonnal betöltődnek. A projektek kezelése a *Project* menüpontból érhető el.

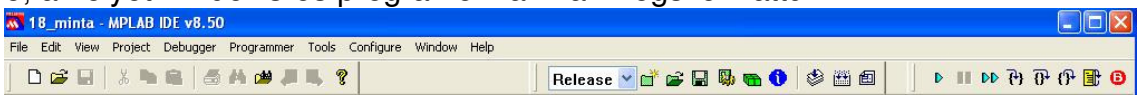
A *munkakörnyezet* a kiválasztott mikrovezérlőről, a szimulációról, a programozó eszközökről, a megnyitott ablakokról, és azok helyéről valamint egyéb IDE beállításokról hordoz információt.

A program telepítése nagyon egyszerű, kitömörítés után dupla kattintással indítsuk el a kapott fájlt, s a telepítés automatikusan végbemegy, illetve választhatjuk az egyéni, testreszabott telepítést. Ebben az összetevők kiválasztásánál a 7. ábra szerint járjunk el, hiszen csak azokat az eszközöket érdemes kiválasztani, amik rendelkezésünkre állnak. A telepített összetevőket később bármikor módosíthatjuk (8. ábra). A legtöbb helyen a tanulói gépeken központi vándorló profilból történik a bejelentkezés, illetve a tanulók csak korlátozott felhasználói jogosultságokkal rendelkeznek. Célszerű ezért a program helyes működésének érdekében az MPLAB telepítési mappájára teljes hozzáférést biztosítani a tanulóknak (ez legtöbbször a C:\Program Files\Microchip könytárat jelenti).

8. ábra

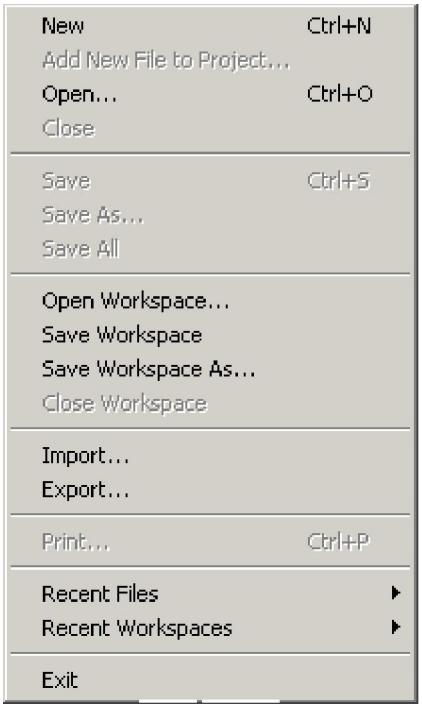
Első lépésben minden tanuló hozzon létre a profiljában, vagy a C: meghajtón egy saját mappát (pl. monogram vagy becenév), és ezen belül az „elso” nevű könyvtárat. Ügyeljünk arra, hogy a könyvtár, illetve a fájlnevekben sose használjunk ékezetes betűket, valamint speciális karaktereket, mivel a szoftver angol nyelvterületről származik, így ezek később rejtélyes hibák forrásai lehetnek! Ennek kettős a célja, egyrészt így nem keverednek a fájlok egymással, másrészt pedig tudunk dolgozni „projekt”-ben. Kerüljük az ékezetes betűket a könyvtárak használatánál, mert kellemetlen meglepetések érhetnek bennünket a program használata során.

A telepítés után egy szokványos legördülő menüs rendszer tárul a felhasználó elé, amelyet windows-os programoknál már megszokhattunk:



Az MPLAB program használatáról részletes – mintegy 100 oldalas – magyar nyelvű leírás olvasható a honlapomon: http://plc.mechatronika.hu/mplab/mplab_v810_leir.pdf. Itt csak a legfontosabb tudnivalókat ismertetetem. A menürendszer első elemét a jól megszokott **File** menü képezi. Itt találhatóak a mentéssel, megnyitással és exportálással kapcsolatos műveletek. A másik nagyon fontos menüsor, amire részletesebben kitérek, az a **Projekt** menü lesz. Az összes munkánkat projektekben fogjuk ugyanis végrehajtani. Ami azt jelenti, hogy minden új munkánk külön projektbe fog kerülni, ami egyben másik könyvtárat is jelent. Ezzel lehetőség nyílik arra, hogy munkáinkat megfelelően rendszerezzük. Ezt tudatosítsuk a tanulókban is.

Először tehát tekintsük át a fájl menüt (9. ábra):



9. ábra

New: Új fájl létrehozása
Add New File to Project...: Új file hozzáadása a projekthez
Open...: Létező fájl megnyitása.
Close: Aktív megnyitott fájl bezárása

Save: Fájl mentése
Save As...: Fájl mentése más néven
Save All: Minden megnyitott fájl mentése.

Open Workspace...: Munkaasztal megnyitása
Save Workspace: Munkaasztal mentése
Save Workspace As...: Munkaasztal mentése más néven
Close Workspace: Aktív munkaasztal bezárása.

Import...: Debug fájl, vagy futtatható hex fájl importálása
Export...: Hex fájlba történő exportálás

Print...: Nyomtatás

Recent Files: Legutóbbi fájlok megnyitása
Recent Workspaces: Legutóbbi munkaasztalok megnyitása

Exit: Kilépés az MPLAB-ból

Az MPLAB szövegszerkesztője tulajdonképpen egy átlagos Windows alapú szövegszerkesztő, amelyet olyan részekkel egészítettek ki, amelyek a programírást megkönnyítik. Az **Edit** menü felépítése (10. ábra):

10. ábra

Undo: Visszavonás
Redo: Ismétlés

Cut: Kijelölt szöveg másolása a vágólapra és eredeti helyéről való törlése (kivágás)
Copy: Kijelölt szöveg másolása a vágólapra
Paste: Vágólapon lévő szöveg beillesztése
Delete: Kijelölt szöveg törlése

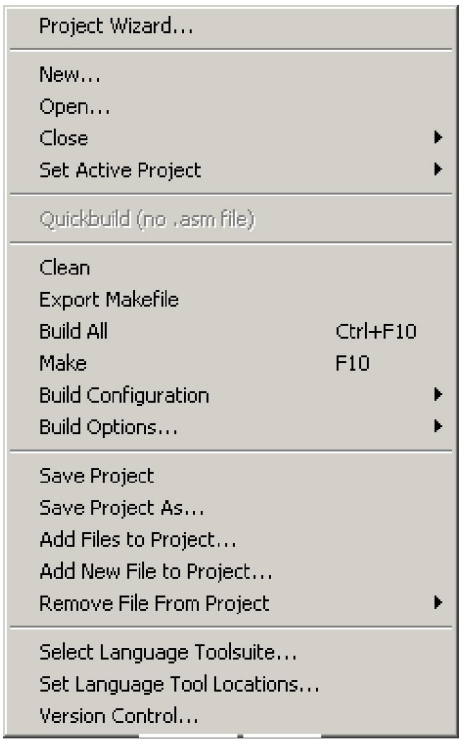
Select All: Aktív fájlban minden szöveg kijelölése

Find...: Kulcsszó keresése szövegben
Find Next: Következő keresése
Find in Files...: Fájlban történő keresés
Replace...: Sztring cserélése szövegben
Go To...: Ugrás a szövegben címkére, vagy sorra

Properties...: Tulajdonságok (Az editor beállításai)

A negyedik legördülő menübe a projekttel kapcsolatos műveletek kerültek megjelenítésre. Ez a menü tartalmazza a projektünkhöz tartozó fontos beállításokat, amelyre a későbbiekben még visszatérünk, hiszen alapvetően meghatározzák a fordítási opciókat.

A többi menüpontot átugorhatjuk egyelőre, – az alapokhoz nem szükséges az ismeretük – nézzük inkább a **Project** menüt. Felépítése a 11. ábrán látható:



11. ábra

Project Wizard: Projekt varázsló

New...: Új projekt létrehozása
Open...: Projekt megnyitása
Close: Projekt bezárása
Set Active Projekt: Aktív projekt kiválasztása

Quickbuild (*.asm file): Aktív asm forráskód gyorsfordítása (projekt nélküli munkát tesz lehetővé)

Clean: Törli az aktív projekt közbenső fájlait (hex-; object-; debug fájl)
Export Makefile: Fájl kiexportálása
Build All: A projekt összes fájljának fordítása
Make: Azon fájlok fordítása, melyek megváltoztak a legutóbbi fordítás óta
Build Configuration: Fordítás konfigurálása (Release:„éles működésre”; Debug: Hibakeresésre (csak ICD2,vagy REAL ICE))
Build Options: Fordítás tulajdonságainak beállítása (forráskódé; projekté)

Save Project: Projekt mentése
Save Project As...: Projek mentése más néven
Add Files to Project: Fájl hozzáadása a projekthez
Add New File to Project...: Új fájl létrehozása a projekten belül
Remove File From Project: Fájl eltávolítás

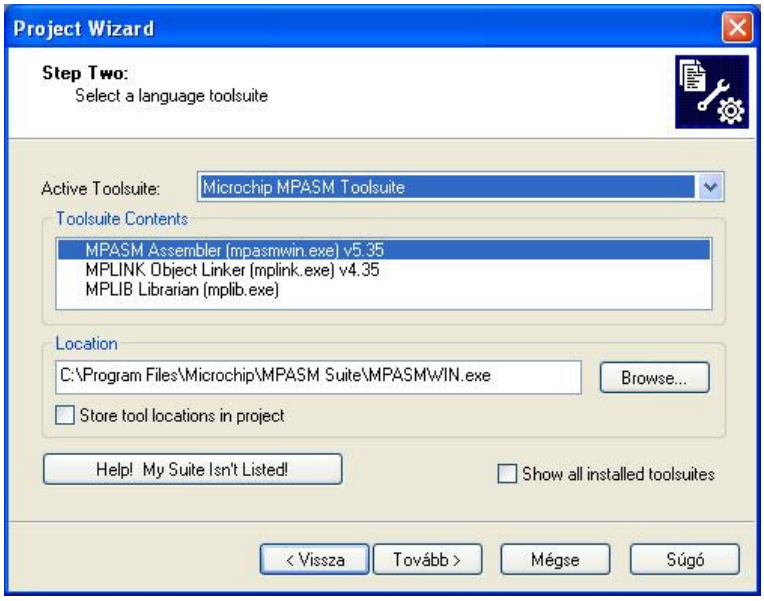
Select Language Toolsuite...: Fordító kiválasztása
Select Language Tool Locations...: Fordító elérési útvonalának kiválasztása

Az alapismeretek után az első projekt elkészítése következik. Az elején létrehozott könyvtárunkban másoljuk be a honlapomon lévő http://plc.mechatronika.hu/mintaprg/mintaprg.htm oldalról lementett első számú (alap.asm) fájlt. Indítsuk el az MPLAB programot, és a könyvtárunkban és a **Project Wizard** segítségével hozzunk létre egy új projektet (*Project> Project Wizard*) a 12. ábrán látható módon. Miután a tovább gombra kattintottunk, egy újabb ablak bukkan fel, ahol a Device feliratnál a legördülő menüből válasszuk ki a 16F84-es processzort (13. ábra). Tovább haladva ismét egy felugró ablak tűnik elénk, ahol felül a legördülő menüben a fordító eszközt tudjuk kiválasztani. Válasszuk itt ki a „Microchip MPASM Toolsuite”-ot. Ekkor az alatta lévő ablakban megjelennek a hozzátartozó eszközök, vagyis a fordításhoz

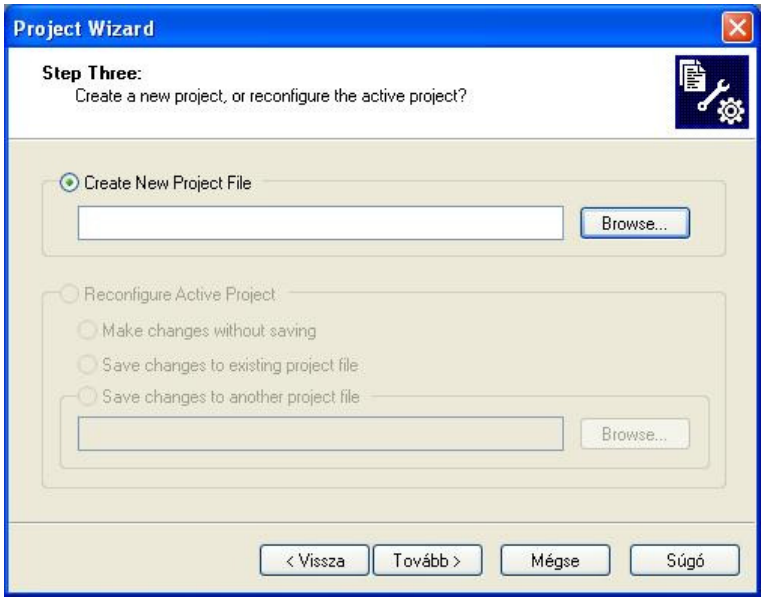
12. ábra

13. ábra

szükséges fájlok. Amennyiben valamelyik eszköznél piros X jelenne meg, akkor manuálisan kell megkeresni a szükséges fájlt. Ez a következőképpen történik: kattintsunk az adott eszközre és a „Location” feliratnál a „Browse" rádiógombra kattintva „mutassuk meg” neki a fájl helyét. A fájlok az MPLAB telepítési mappáján belül – C:\Program Files\Microchip általában – az MPASM Suite könyvtárban találhatók. A „varázsló” harmadik lépésében a projekt létrehozása történik. A „Create New Project File” beviteli mezőnél a „Browse” gombbal adjuk meg az általunk létrehozott könyvtárat. Kétszer kattintva lépjünk be a könyvtárba, majd és a fájlnévhez írjuk be, hogy elso, és kattintsunk a Mentés gombra. Rákattintva a tovább gombra megjelenik a varázsló negyedik lépésének párbeszédablaka. Itt kell hozzáadni a forrásfájt – ezt később is megtehetjük a projekt fülön – a projektünkhöz. Válasszuk ki az elején bemásolt asm fájlt, és az Add gombbal adjuk hozzá a projektünkhöz (16. ábra). Ismét a Tovább, majd a Befejezés gomb következik.



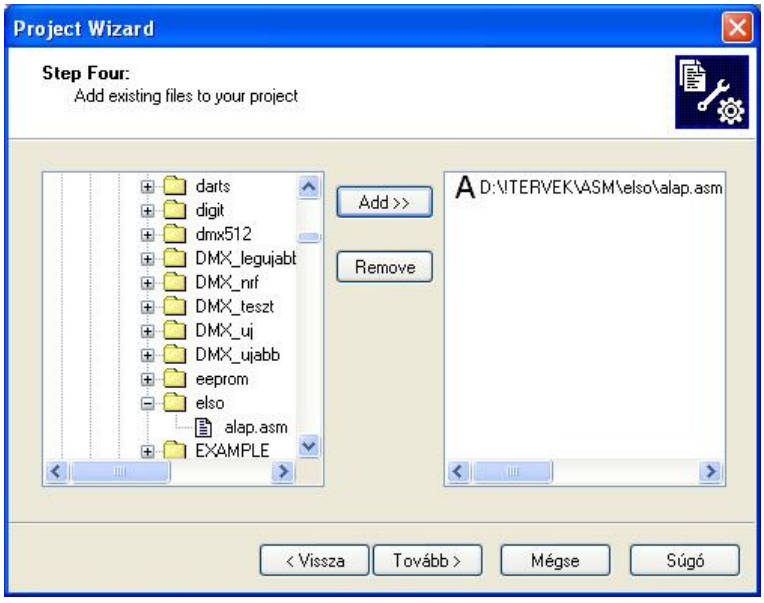
14. ábra



15. ábra

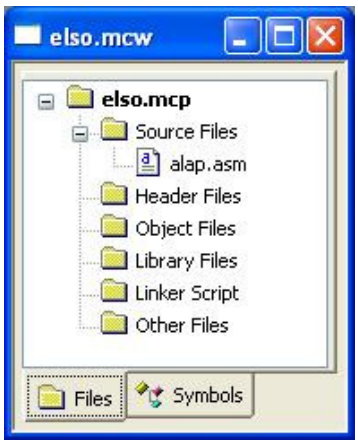
Ezután szabjuk testre kicsit a projektünket. Ehhez nyissuk mg a *Project>Build*

Options>Project párbeszédablakot. Az „MPASM Assembler” fülön a Default Radix-nál a Hexadecimal rádiógombot válasszuk ki. Az alapértelmezett számformátum beállítása nagyon fontos, hibás kiválasztása rejtélyes hibák forrása lehet. Azért célszerű a hexadecimálist választani, mert ilyenkor minden jelölés nélkül beírt számot hexadecimálisként értelmez a fordító. Külön kell jelölni viszont a decimális D’10’, vagy .10, és a bináris B’10100101’ számokat! Az alapértelmezett számformátumot tüntessük fel a forrásprogramunk elején is! A beállító menü a 17. ábrán látható. Ezután a „Categories” legördülő menünél az „Output-ot” válasszuk. Eztán a „Diagnostics level”



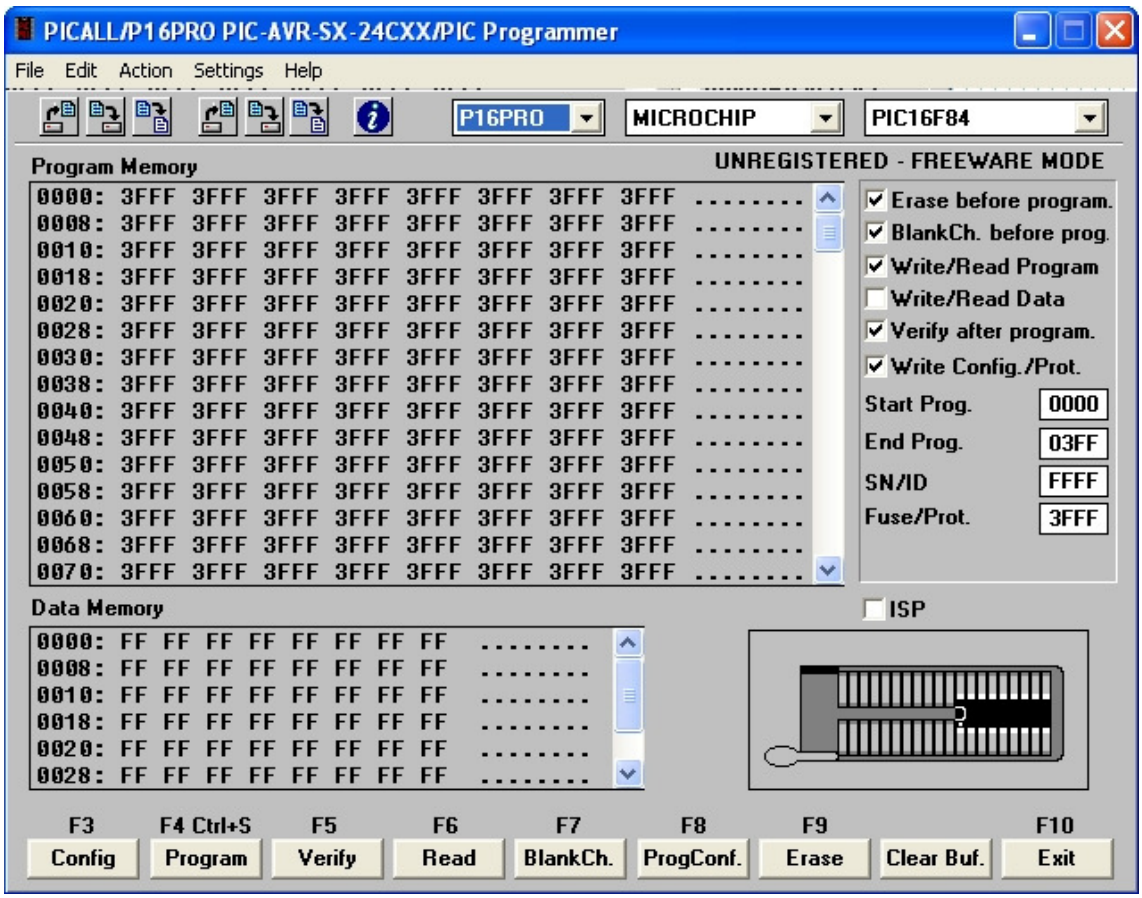
16. ábra

17. ábra



19. ábra

menüből az „Errors and warnigs” lehetőségeket válasszuk. Ez azt jelenti, hogy fordítás után a program hibaüzeneteket és figyelmeztetéseket küld, üzeneteket nem. Fontos megjegyezni: amennyiben átnevezzük az alap.asm fájlt, vagy másik fájlt másolunk a project-könyvtárunkba, akkor a régi forrásfájlt el kell távolítani, és az új fájlt kell a projekthez hozzáadni (18. ábra)! Ezt a Projekt ablakban tehetjük meg. Amennyiben nem látszana,



18. ábra

akkor a „Wiew” menüben tegyünk pipát a „Projekt” elé. A *File>Open* menüpontban nyissuk meg az alap.asm fájlt, vagy a „Projekt” ablakban kattintsunk kétszer az alap.asm fájlon. Rendezzük el nekünk tetszően az ablakokat. Ahhoz, hogy ez a program legközelebbi betöltésekor is így jelenjen meg végezzük el a következőket: a *Configure>Settings* menüben a „Workspace” fülön pipáljuk ki a „Reload last workspace at startup-ot”. Ezzel az

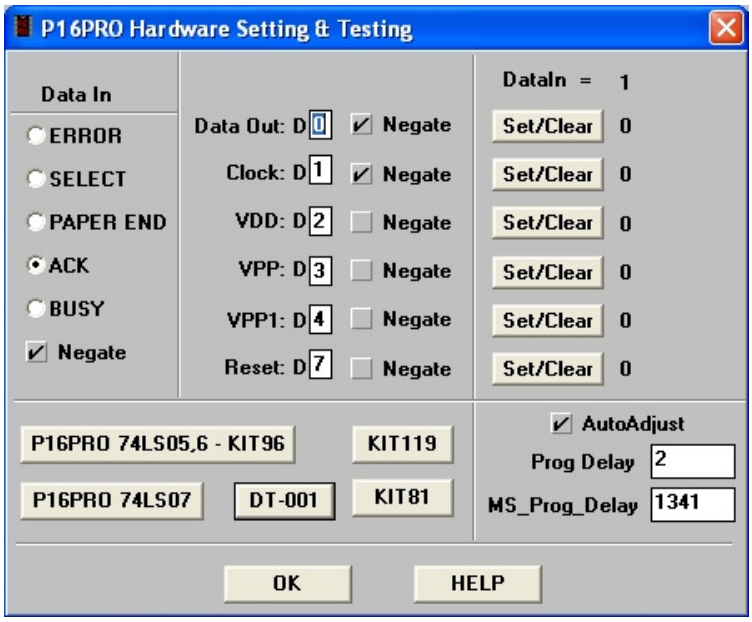
alapokkal készen is vagyunk. Azért, hogy az eddigi munkánk ne vesszen el, a *Project>Save Project* menüpontban mentsük el beállításainkat!

Ezután fordítsuk le a programunkat a *Project>Build All* pontban, vagy használjuk a <Ctrl+F10> billentyűkombinációt. A sikeres fordítás után – a Build Options ablakban a „Build completed successfully" felirat jelenik meg – következhet a program beégetése. Indítsuk el a PICALL programot, és a felső sorban lévő legördülőmenükben válasszuk ki programozó készüléket (P16PRO), a gyártót (MICROCHIP), és a típust (PIC16F84) ebben a sorrendben. A programozó készülék részletes leírása a mellékletben található meg. A jobboldali ablakban a „Write/Read Data" – erre csak akkor van szükségünk, ha a belső EEPROM-ot is használni akarjuk – kivételével pipáljuk ki az összes jelölőnégyzetet (19. ábra). Az EEPROM memóriával célszerű megfontoltan bánni, mivel nem írható korlátlanul. Az egyes opciók jelentése a következő:

- Erase before program – törli a mikrovezérlőt, mielőtt az új programot beégetnénk. Célszerű bekapcsolni, mert a tapasztalataim szerint biztosabb így az égetés
- Blank Ch. Before prog. – ellenőrzi a mikrovezérlőt, hogy üres-e a memóriája, mielőtt beégetné a programot
- Read/Write Program – a programunk beírását, illetve kiolvasását kapcsolja be, e nélkül a proramunk nem égetődik be
- Write Config./Prot. – az úgynevezett konfigurációs bitek (pl. oszcillátor típusa), és a kódvédelem égetését teszi lehetővé, nélküle nem fognak működni a progjamjaink.

Fontos megjegyezni, hogy csak olyan programokat égessünk be, amelyeknél a kódvédelem ki van kapcsolva, mert ellenkező esetben többször nem tudjuk égetni a PIC-et.

Ezután a programozó testreszabása következik. Nyissuk meg a *Settings>Hardware Settings/Test* menüt, és a megjelenő ablakban a „P16PRO 74LS05,6 – KIT96" egységet válasszuk ki. A *File>Open File* (vagy az F2 billentyű) menüben nyissuk meg a lefordult [.hex] kiterjesztésű fájlunkat. Helyezzük a programozóba megfelelő pozícióban a PIC-et. Mutassuk meg a tanulóknak, hogy mi alapján lehet azonosítani a mikrovezérlő tokján az 1-es lábat! Az egyes mikrovezérlők megfelelő pozícionálását a programozó készülék



20. ábra

beültetési rajzán találhatjuk meg (melléklet 2. ábra). A pozíciót ezen kívül a PICALL program is mutatja. A vízszintes pozíció egyértelmű, a keskeny tokokat pedig mindig a felső két tüskesorba kell helyezni. Célszerű a mikrovezérlőt eleve egy precíziós IC foglalatba tenni, ugyanis mechanikailag hamarabb elhasználódik, mintsem elérnénk a maximális programozási ciklusszámot. Következhet az égetés az *Action>Program* (vagy F4) pontban. A sikeres égetés után jöhet a próba. Vegyük ki a programozóból a PIC-et csipesz vagy IC kiszedő segítségével. Hívjuk fel a figyelmet arra, hogy kézzel semmiképp sem szabad kivenni a mikrovezérlőt a programozóból, mert balesetet okozhat! A kivett IC-t kikapcsolt tápfeszültség mellet helyezzük a próbapanelba. A JMP2 jumpert zárjuk rövidre, kapcsoljuk rá a tápfeszültséget a panelra és ellenőrizzük a működést. Jelen esetben a 4 db kétszínű LED zöld fénnyel világít. Amennyiben az ICSP csatlakozót használjuk, akkor a mikrovezérlőt nem kell kivenni a próbapanelból, csak a négy DIP kapcsolót kell a programozás idejére OFF állásba kapcsolni, majd a teszthez újra ON állásba! Természetesen a programozáshoz használhatjuk az IC-PROG, illetve a WinPic szoftvert is, illetve programozóként a honlapomon található ICD2-t, vagy PICkit2-t is.

A bemutató után a tanulókkal gyakoroltassuk az MPLAB kezelését, a programok lefordítását, az égető kezelését és a próbapanel használatát, hogy a következő foglalkozáson ez már kézség szintjén menjen. Az egyes tanulói munkahelyeket kialakíthatjuk úgy is, hogy mindegyik helyre rakunk egy-egy programozót és próbapanelt, vagy egy központi géphez rendeljük hozzá csak. Tapasztalatom szerint elegendő az utóbbi megoldás, ritkán fordul elő ugyanis, hogy mindenki egyszerre akar égetni.

3.2 Mikrovezérlők szoftverfejlesztése, az assembly alapjai

A második foglalkozás elején röviden fogaljuk össze az eddig tanult ismereteket.

3.2.1 Mikrovezérlők szoftverfejlesztése

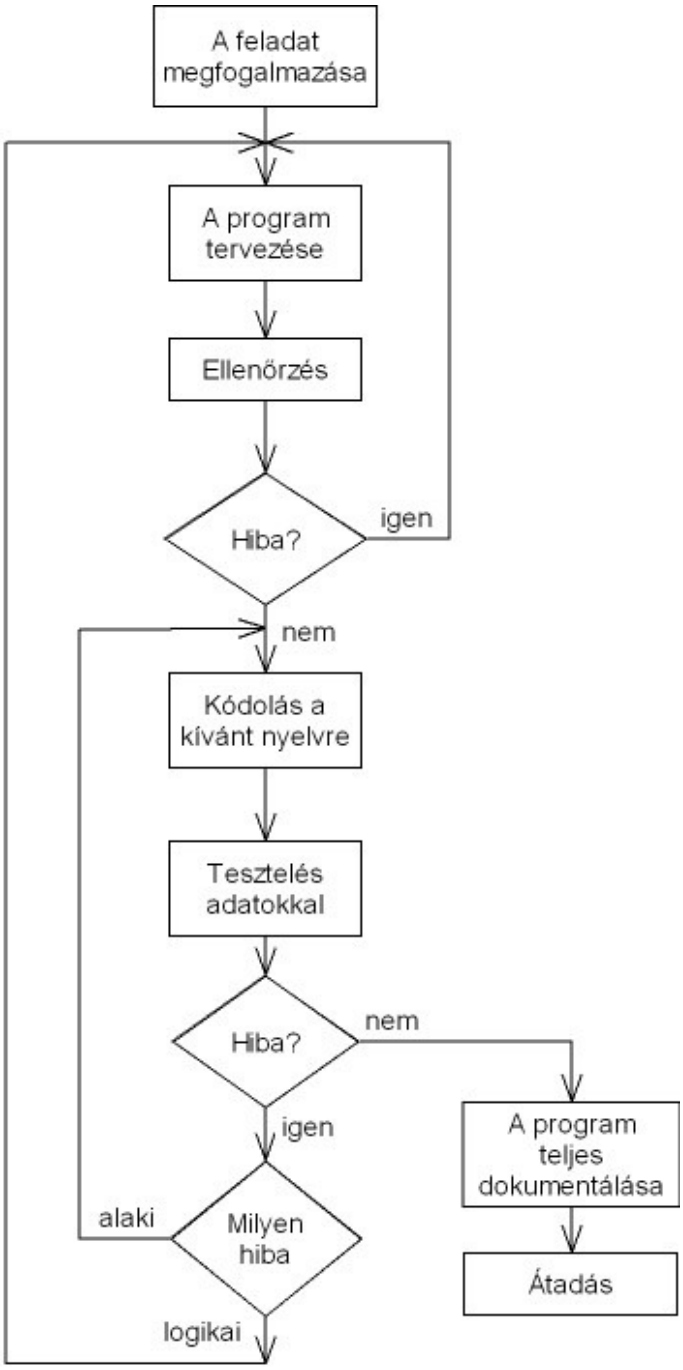
A mikrovezérlős fejlesztésekben a programozásnak a legtöbb esetben nagyobb a jelentősége van, mint a hardver áramköri kialakításának. Ezért a programozás alapjainak elsajátítása nagy fontossággal bír (*Dr. Kónya*, 2003)!

A programozás nem csak a mikrovezérlőknél kerül előtérbe, hanem ez általános fogalom az informatikában, amelynek általánosan elterjedt módszerei, szabályai vannak. A programnak nagy jelentősége van, hiszen a mikrovezérlőben futó programmal valósítjuk meg a megoldandó feladatot!

A számítógépes problémamegoldás során a következőkből indulunk ki:

- Ismert bemenő adatok
- az ismeretlen kimeneti adatok
- összefüggések az ismert és az ismeretlen között

A problémamegoldás során megfelelően egymás után kapcsolt műveletek sorozatával eljutunk az ismerttől az ismeretlenig. A programfejlesztés főbb lépéseit a 21. ábra foglalja össze:

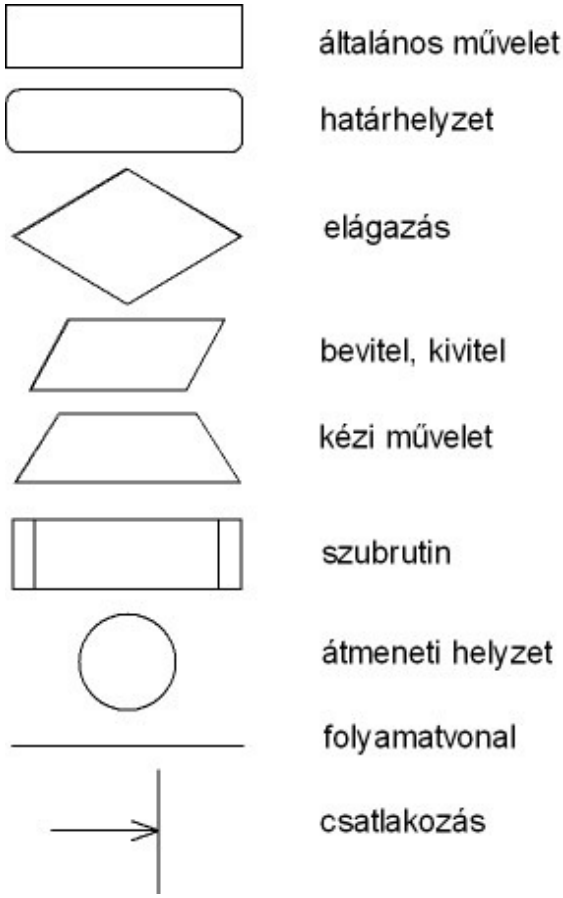

A feladat megfogalmazása
A program tervezése
Ellenőrzés
Hiba?
igen
nem
Kódolás a kívánt nyelvre
Tesztelés adatokkal
Hiba?
nem
igen
A program teljes dokumentálása
Átadás
alaki
Milyen hiba
logikai

21. ábra

- A feladat matematikai és logikai alakban való megfogalmazása.
- Tervezés, melynek során megfelelő **algoritmust** választunk. Algoritmus: aritmetikai és logikai műveletek sorozata, amely lehetővé teszi a feladat megoldását. Az algoritmikus lépések mindig műveleteket valósítanak meg. A programozás során alapvetően négy algoritmuslépést használunk:
  - Számítás: numerikus, logikai, karakterműveletek
  - Döntés: összehasonlítás alapján alternatív lépések kiválasztása
  - Bemenet: adatok bevitele a műveletvégzéshez
  - Kimenet: a kiszámított eredmények kiírása

A programok tervezésénél a következő módszerek használatosak (*Dr. Kónya*, 2003):

- Folyamatábra készítés
- Strukturált programozás
- Felülről lefelé tervezés
- Objektumorientált programozás

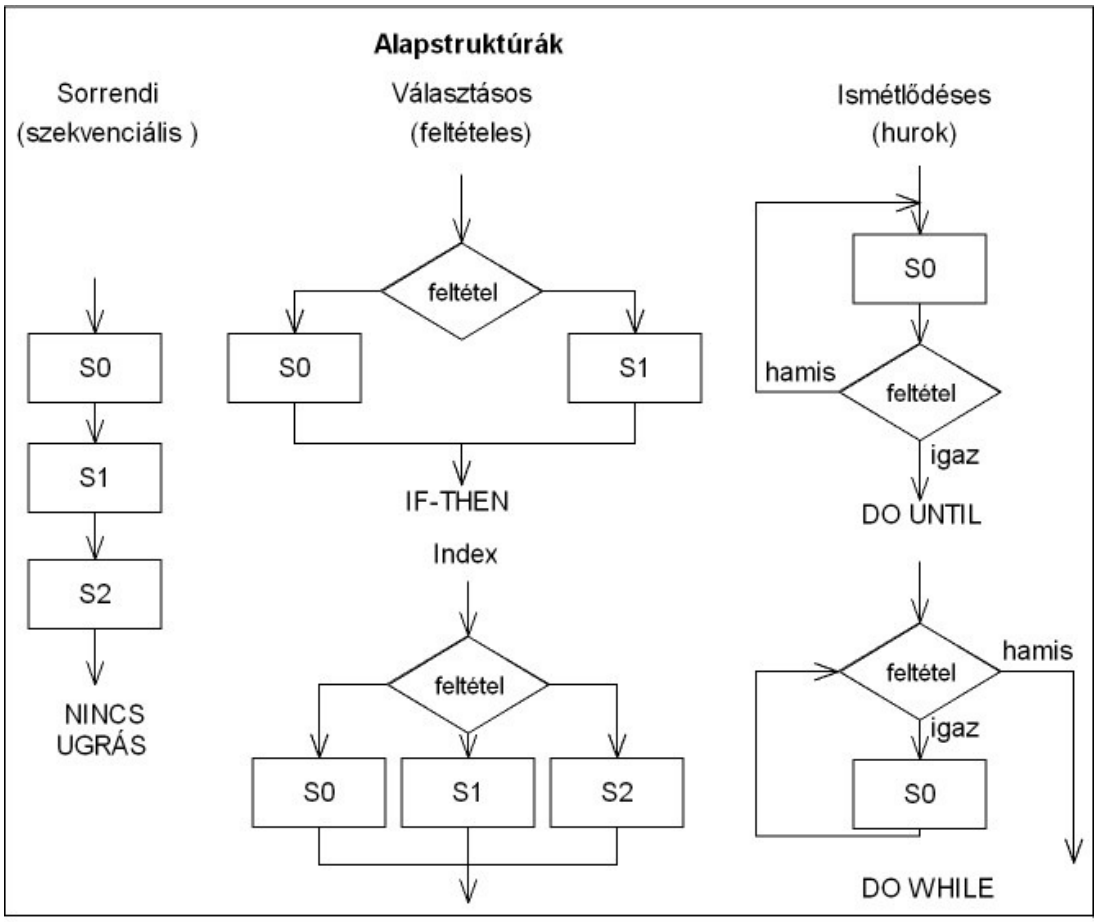


22. ábra

Folyamatábra módszer: A program tervezése jól látható, grafikus módon történik. Szabványos szimbólumokkal (22. ábra) dolgozik. A folyamatábrák megérthetőek programozási ismeretek nélkül is, de bonyolult megtervezni, megrajzolni, változtatni. Nincs egyszerű módszer a folyamatábrán történő hibakeresésre, tesztelésre. Annak eldöntése, hogy milyen részletes, vagy tömör legyen a folyamatábra, nem egyszerű. Abban az estben, ha túl tömör nem adja vissza a program minden fontos részletét, ha túl részletes, akkor pedig nehezen áttekinthető. A folyamatábrában könnyű a nyilakat ide-oda húzni, a kódolás során azonban ezt már nehéz megvalósítani a

sok ugrás miatt.

Moduláris programozás: A teljes programot részekre, más néven modulokra osztjuk. Az alapvető probléma az, hogy hogyan osszuk fel modulokra a programot, és hogyan egyesítsük ezeket komplett működő programmá? A



23. ábra

modulokat könnyű megírni, tesztelni, a bennük lévő hibát megkeresni. Egy-egy modul valószínűleg sok más feladatban is felhasználható. Több programozó is könnyen együtt tud működni. A hibák általában csak egy modulhoz kapcsolódnak, így könnyebb kijavítani. A modulok nagyon pontos, részletes dokumentációt igényelnek, mivel a program más részeire is kihatással vannak. A végső, a programba beépülő modulok tesztelése nehézkes lehet, mert egyes modulok olyan bemenő adatot kaphatnak, amelyet másik modul állít elő.

Strukturált programozás: ennél a módszernél – nevéből következően – a programokat a programozó alapstruktúrákból építi fel. Nevezzük el az egy utasítást, vagy utasítássorozatot S-nek. A jelölés felhasználásával a következő

alapstruktúrákat kapjuk (23. ábra):

Szekvenciális (sorrendi) struktúra: S0, S1, S2 … struktúrák a programban sorban, egymás után kerülnek végrehajtásra.

Feltételes struktúra: abban az esetben, ha C igaz, akkor S0, különben S1. A C itt egy feltételt jelent (IF-THEN-ELSE).

Hurok struktúra: amíg C igaz, csináld S-t. A C itt is egy feltételt jelent (DO-WHILE, DO-UNTIL).

Index struktúra: I esetén S0, S1, S2 …, Sn, ahol I az index, és 0, 1, 2, …, n értékű lehet. Abban az esetben, ha I=0, akkor S0; ha I=1, akkor S1; … ha I=n, akkor Sn hajtódik végre (DO-CASE). Matematikailag bizonyítható, hogy ezen négy struktúra felhasználásával bármilyen program elkészíthető. Előnye, hogy a műveletek sorrendjét könnyű követni, ezért könnyű hibát keresni és tesztelni. Az így felépített program már önmagát dokumentálja, könnyen átlátható. A gyakorlat azt mutatja, hogy ezt a módszert használva nő a programozói teljesítmény. Hátránya, hogy a strukturált program általában hosszabb futásidejű, és több memóriát foglal el, mint a nem strukturált változat. A korlátozott számú struktúrával néhány esetben nagyon nehéz bonyolult feladatot megvalósítani. Ugyan bármely program megírható ezeknek, a struktúráknak a felhasználásával, de nem biztos, hogy az így kapott program hatékony, és kis tárigényű lesz. Az egymásba ágyazott „ha…akkor…különben” struktúrákat nehéz követni. A programozó gondolkodásmódját leszűkíti.

Felülről lefelé programozás: a módszernek az a lényege, hogy először a teljes átfogó programot írjuk meg, a benne szereplő egyes feladatokat, eljárásokat csak kijelöljük – nevet adunk neki – és csupán definiáljuk a név mögött álló feladatot. A következő lépésben ezeket a feladatokat ismét részfeladatokra bontjuk – ha szükséges – és ismét definiáljuk azokat, és így tovább. Amikor a lebontásban eljutottunk az elemi szinthez, akkor kezdjük a program megírását. Innen kezdődik a felfelé történő programírás, és a minden szinten elvégezhető tesztelés. Hátrány egyrészt az, hogy a meglévő programokat, rutinokat nem feltétlenül tudjuk könnyen felhasználni, nem eredményez általánosan felhasználható modulokat. Másrészt nem biztos, hogy hatékony programot nyerünk a módszer alkalmazásával.

- Ellenőrzés (papíron)
- Programozás:

- Gépi nyelven
- Gépre orientált nyelven
- Feladatra orientált nyelven

Gépi nyelv: a programozó az utasításokat közvetlenül gépi kódban adja meg. Az utasítások kódját általában hexadecimális formátumban viszik be a gépbe. Az utasítások, valamint az adatok címeit előre meg kell határozni. A programmódosítások az összes cím megváltozását idézik elő. Nehézkes és időigényes.

Gépre orientált nyelv: ez egy szimbolikus nyelv, melynek utasításai a hozzá tartozó berendezés gépi nyelvű utasításaival megegyezők, vagy ahhoz hasonlóak. Az utasítások neveit könnyen megjegyezhető úgynevezett *mnemonik*okkal rövidítik. Az utasítások és adatok címei szimbolikus formában is megadhatók (címkék). Ilyen nyelv például az **assembly**. Ez a programnyelv különösen alkalmas a mikrovezérlők programozására. Ezzel a nyelvvel tudjuk a legjobban kiaknázni a mikrovezérlőben rejlő adottságokat. Az assembly-vel tudjuk a legrövidebb és leggyorsabb programkódot megírni. Ezen előnyök miatt mindenképpen az assembly nyelv használatát javaslom a mikrovezérlők programozásához!

Feladatra orientált nyelv: lehetővé teszik a feladatnak a problémákhoz igazodó megoldását, nevezik magasszintű programnyelvnek is. Az idők folyamán nagyon sokféle ilyen nyelv keletkezett (ALGOL, COBOL, BASIC, PASCAL, C, C++, stb.). A PIC mikrovezérlőkhöz is létezik C fordító, így a programokat akár C nyelven is megírhatjuk. A magasszintű programnyelvek használata a fejlesztési időt lerövidíti. Tisztában kell lennünk azonban azzal, hogy az így generált tárgyprogram sokkal **redundánsabb** lesz, sok helyet fog elfoglalni a memóriában, a futási idő megnövekszik, gyors alkalmazások megvalósítására nincs lehetőség!

- A program tesztelése. Célja a hibák felderítése és kiküszöbölése. A hibák két csoportba sorolhatók:
  - Alaki (szintaktikai) hiba: olyan utasítás írtunk, amelyet a fordítóprogram nem tud értelmezni
  - Logikai hiba: elgondolásunk hibás volt, ez nem vezet el a megoldáshoz, újra kell gondolnunk a folyamatot
- A programokat a programkönytárban összefoglalva rendszerezzük és

elkészítjük a teljes dokumentációját.

Látható, hogy a program írása nem egyszerű, hanem nagyon összetett feladat. A programfejlesztéshez többféle struktúrából válogathatunk. Fontos megjegyezni, hogy ezek nem felváltják, hanem kiegészítik egymást. A folyamatábra használata mindenképp ajánlott a tanulók számára, illetve a strukturált programozásnál célszerű az alapstruktúrákat megismertetni velük. A moduláris programozást projektmunka keretében lehet gyakoroltatni.

A programfejlesztésnél a következő alapelveket célszerű betartani:

- Kis lépésekben történő haladás és fejlesztés. A nagyobb részfeladatokat egymástól logikailag elkülönülő kis modulokból kell felépíteni, mivel ezek önállóan tesztelhetők, és esetleges megváltoztatásuk nem igényli a teljes rendszer újratervezését.
- A feladatnak megfelelő programvezérlés lehetőleg egymás utáni, egyenként végrehajtható részekből álljon, ne ide-oda ugrásokból, áttekinthetetlen programhurkokból, ciklusokból. Ez a hibakeresést is segíti.
- Minél több grafikus, vizuális leírás alkalmazása (folyamatábra módszer).
- Egyszerű és világos megfogalmazások, fogalmak használata.
- Olyan algoritmusok felhasználása, amelyek ismertek, és kipróbáltak.
- A programtervezéskor figyelembe kell venni azt, hogy melyek azok a tényezők, paraméterek, amelyek megváltozhatnak.
- A kódolást csak a programtervezés teljes befejezése után szabad elkezdeni!

Az utolsó pontra hívjuk fel külön a tanulók figyelmét! Tapasztalatom szerint ugyanis a tanulók hajlamosak mindjárt a program írásával kezdeni a feladatot, anélkül, hogy megterveznék azt. Ez az első hiba jelentkezésekor nagy problémát fog okozni, ugyanis azt sem tudjuk ilyenkor, hogy hol keressük a hibát! A tanár munkáját is nagymértékben megnehezíti, mert mivel nem tudja a gondolatmenetet, segíteni sem tud hiba esetén.

3.2.2 Az assembly programozás alapjai

A mikrovezérlőket működtető programok megírásánál a programozó számos lehetőség közül választhat, mint ahogy az előző fejezetben olvasható.

Minden digitális számítógép, bármilyen bonyolult feladatot is hajtson végre, végül is a bináris számokat utasításként értelmezve végzi el a műveleteket a szintén bináris formátumú adatokon. Ez teljesen nyilvánvaló, mivel a gépben csak bináris 0 és 1 alakú információ feldolgozása lehetséges. Az ilyen formában előálló programot nevezzük gépi kódú programnak.

A számítógép programozásában rejlő összes lehetőség legjobb kihasználását a gépi kód teszi lehetővé. Ezen a szinten írhatók a gép működése szempontjából a leghatékonyabb programok, itt használhatók ki legjobban az egyes utasítások hatásai és mellékhatásai, itt alkalmazhatja a programozó a legszellemesebb megoldásokat és trükköket (*Dr. Kónya*, 2003).

A gépi kódban történő programozás azonban az ember számára nagyon nehézkes, kézség szintű elsajátításához, a programozói rutin megszerzéséhez igen hosszú idő szükséges. Ezt a fáradtságos munkát még az is nehezíti, hogy meg kell tanulni az utasításoknak megfelelő bináris, hexadecimális, vagy oktális kódot, és ki kell számítani a programban szereplő címek abszolút, vagy relatív értékeit.

A gépi kódú programozás ezen hátrányait – az előnyök megtartása mellett – küszöböli ki az assembly nyelven történő programozás!

Az assembly egy egyszerű programozási nyelv, amely lehetővé teszi, hogy a gépi kód helyett az utasításokat könnyen megjegyezhető nevekkel írjuk le. Ezeket, a kódneveket – melyek általában az utasítás funkciójának a rövidítései – *mnemonikoknak* nevezzük. Az utasításhoz háromféle operandus kapcsolódhat:

- Konstans (literál), vagyis egy állandó
- Adatregiszter címe, aminek tartalmával a műveletet végezzük
- Programmemória cím, amely a következő végrehajtandó utasítást tartalmazza

További egyszerűsítést jelent a gépi kódhoz képest, hogy a program egyes belépési pontjaira nevekkel hivatkozhatunk, ún. *címkéket* rendelhetünk hozzá.

Ezen kívül szintén nevekkel hivatkozhatunk különböző regiszterekre, bájtokra, bitekre, azaz *szimbólumokat* rendelhetünk hozzájuk. Az elnevezésekre ügyeljünk, mert ezek megállapodás szerint csak betűvel kezdődhetnek (ugyanis csak a számok kezdődhetnek számmal). A címkék általában kettőspontra végződnek (az MPLAB elfogadja kettőspont nélkül is). A utasítások elnevezései természetesen kötöttek, ezt a gyártó definiálja. A PIC mikrovezérlőknél az adatmemória két részre oszlik:

- **SFR** (Special Function Registers) – speciális funkciójú regiszterek
- **GPR** (General Purpose Registers) – általános célú regiszterek

A speciális funkciójú regiszterek elnevezéseit a Microchip definiálta, ezeket nem nekünk kell kitalálni (pl. a portok elnevezései: PORTA, PORTB, stb.). Az elnevezések a <típus>.inc fájlban, az MPLAB program könyvtárában találhatók meg az egyes típusokhoz. Jelen esetben mi a *„pic16f84.inc”* fájlt fogjuk használni. Az általános célú regisztereknek tetszés szerinti betűvel kezdő, akár bitenkénti elnevezést is adhatunk. A címkék, szimbólumok elnevezése tetszőleges, azonban hívjuk fel rá a tanulók figyelmét, hogy lehetőség szerint rövid, lényegretörő (beszédes) elnevezéseket válasszunk! A találó elnevezések ugyanis megkönnyítik a forrásprogram „olvasását”, ami a későbbiekben lényegesen egyszerűbbé teszi a program elemzését és a hibakeresést! A fent említett könnyítésekkel a programozó megszabadul a fáradtságos címszámítástól, ami a gépi kódú programozáshoz szükséges volt, ugyanis a címeket assembly-nél a fordítóprogram számolja ki.

A fordítóprogramnak a feladata, hogy értelmezze, és gépi kódra fordítsa az általunk szimbolikus formában megírt programot. Az általunk szövegszerkesztővel (ez tetszés szerinti szövegszerkesztő) megírt programot *forrásprogramnak* nevezzük, assembly-ben a kiterjesztése asm. Ezt a forrásprogramot alakítja át a fordítóprogram *tárgyprogrammá.* Az assembly programnyelvben a fordítóprogramot *assembler*-nek nevezik! Elnevezése az angol assemble (összeállítás) szóból származik (*Dr. Kónya*, 2003).

Mint minden nyelvnek, az assembly nyelvnek is van nyelvtani formai szabályrendszere, ún. szintaktikája. Az assembly program sorokból áll, melynek felépítése a következő:

címke	művelet	operandus	megjegyzés
(label)	(operation)	(operand)	(comment)
START:	MOVLW	.15	; a W-be 15-öt töltünk

Egy programsor tartalmazhat *utasítást,* amelyet a gép végrehajt, vagy *direktívát*, amely a fordítóprogramnak szól. Az egyes mezőket minimum egy szóközzel kell elválasztani, de sokkal áttekinthetőbbé válik a forrásprogramunk, ha tabulátorokkal igazítjuk a mezőket. Az egyes mezők értelmezése a következő:

- Címkemező: ide írjuk azokat a neveket (címeket), melyekkel a program lényeges pontjait (pl. szubrutin belépési pont, ugrási cím, stb.) meg akarjuk jelölni. A fordítóprogram a fordítás során a címkéhez a memóriabeli elhelyezkedési címét rendeli. Abban az esetben tehát, ha a program során bárhol erre a címkére hivatkozunk, akkor a memóriabeli címének értéke helyettesítődik be. A címke csak betűvel kezdődhet, és általában kettőspontra végződik (a Microchip assemblerénél a kettőspont elhagyható).
- Művelet (utasítás) mező: ide írjuk a megfelelő utasításokat, valamint a direktívákat.
- Operandusmező: itt kell megadni az előző mezőben szereplő utasításhoz, vagy direktívához tartozó operandusokat. Az operandusmezőben egy, vagy két operandus állhat, illetve lehet olyan eset is, hogy nincs operandus. Két operandus esetén azok egymástól való elválasztására vesszőt kell használni. Az operandusmezőben a következő elemek – melyek kifejezéseket is alkothatnak – szerepelhetnek:

    - Konstans (literál, állandó), amely lehet decimális, hexadecimális, oktális, vagy bináris
    - Szövegkonstans: idézőjelek, vagy aposztrófok közé írt karaktersorozat, amely ASCII kódját adja vissza a kifejezés
    - Szimbólum: betűvel kezdődő elnevezés
    - Regiszternév: a processzor belső regisztereinek, bitjeinek szimbolikus elnevezései (a gyártó definiálja)

- Feltétel (státusz) bit (Flag Bit): ezek valamilyen műveletvégzés eredményeképpen állnak be, az adott művelet után beállt állapotról adnak jelzést (pl. kinullázódott egy regiszter, átvitel történt összeadáskor, stb.), ezeket szintén a gyártó adja meg
- Műveleti jelek:
  + összeadás
  - kivonás
  * szorzás
  / osztás
  & logikai ÉS
  ^ logikai VAGY
- Speciális jelek
  $ az utasításszámláló (programszámláló) aktuális értéke

- Megjegyzés mező: ide saját megjegyzésünket, magyarázó szövegünket helyezhetjük el. A megjegyzést mindig pontosvesszővel kel kezdeni. Minden pontosvessző után írt szöveg megjegyzésnek tekinthető, ilyen módon akár egy teljes sor is lehet megjegyzés.

A programok megjegyzésekkel való ellátása, „kommentezése” nagyon fontos dolog. Ugyanis az assembly-ben írt programokban „első látásra” nagyon nehéz kitalálni az egyes programrészek funkcióit. Sokszor maga a programozó is elfelejti néhány nap múlva, hogy mit is akart itt csinálni! A jó kommentezés azonban nagyban megkönnyíti a hibakeresést, illetve a programjaink továbbfejlesztését. Tapasztalatom szerint a tanulók különösen hajlamosak a megjegyzések elhagyására, ezért tudatosítsuk bennük ennek fontosságát!

A assemblyben megírt forrásprogramunkból az assembler állítja elő a tárgyprogramot. Ezt a műveletet fordításnak nevezzük. Az assembler a fordítást több menetben hajtja végre. Az első menetben az assembler végigolvassa a forrásprogramot és felépíti a szimbólumtáblázatot. Az assembler két táblázatból dolgozik, az egyikben az állandó szimbólumok találhatók, a másikban pedig a felhasználói szimbólumok. Itt rendelődnek hozzá a szimbólumokhoz a megfelelő értékek a szimbólumok memóriabeli elhelyezkedése alapján. Az olvasás befejezésekor minden szimbólumnak értéke kell, hogy legyen. Azokat a szimbólumokat, amelyek nem kaptak értéket nem definiált szimbólumnak (undefined symbol) nevezzük. A második menetben történik a tulajdonképpeni

tárgyprogram létrehozása. Ilyenkor az assembler újra végigolvassa a forrásprogramot és átalakítja a programsorokban szereplő utasításokat gépi kódra. A programsorok értelmezése során az assembler felismeri a különböző *szintaktikai* (alaki), illetve *szemantikai* (értelmezésbeli) hibákat. Ezeket hibaüzenet formájában meg is jeleníti. A hibalista és a tárgyprogram mellett a második, vagy a harmadik menetben elkészül a fordítási lista is. Ez tulajdonképpen a forrásprogram másolata, amely azonban sorról sorra tartalmazza a helyszámláló aktuális értékét, valamint az adott sor fordítása révén nyert gépi kódot, illetve hibás sor esetén a hibaüzenetet. A fordítási listában megtaláljuk még a szimbólumok névsorba rendezett táblázatát is a hozzárendelt értékekkel együtt. A lista végén pedig egy statisztikát olvashatunk a programról. A fordítási lista tehát lényegében a fordítás dokumentuma (*Dr. Kónya*, 2003).

A mikrovezérlők alkalmazásakor általában nem áll rendelkezésre az adott mikrovezérlőn futó assembler program, amivel a fordítást el tudnánk végezni. Ilyenkor az a megoldás, hogy egy másik gépen (pl. PC) végezzük el a fordítást. Ehhez megfelelő fejlesztőkörnyezet szükséges, amely rendelkezik egy ún. kereszt-fordítóval (cross-assembler). A PIC mikrovezérlőkhöz a Microchip által kifejlesztett PC-n futó MPASM kereszt-fordító a leghatékonyabb megoldás. Az MPASM az integrált MPLAB fejlesztőkörnyezet része (lásd 3.1 fejezet). A kereszt-fordítóval történő programfejlesztés során valamilyen szövegszerkesztővel kell megírni a forráskódot (jelen estben .ASM kiterjesztésű fájl). A forrásprogram megírható az MPLAB beépített szövegszerkesztőjével, vagy bármilyen más szövegszerkesztővel, a lényeg az, hogy *text* formátumú legyen. A lefordításhoz ennek a fájlnak meg kell felelnie mind a formai (szintaktikai), mind az utasítások, operátorok, direktívák helyes használatát megkívánó szemantikai követelményeknek. A fordítás során többféle fájl keletkezik (*Dr. Kónya*, 2003):

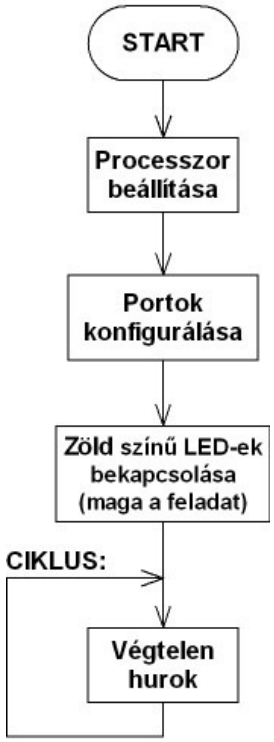
- A fordítási listát tartalmazó nyomtatható szöveges .LST kiterjesztésű fájl
- A hibaüzeneteket tartalmazó .ERR kiterjesztésű fájl
- A szimbólumokat és debug információkat tartalmazó .COD kiterjesztésű fájl
- A tárgyprogram .O kiterjesztéssel, amit az összefűző (linker) program

használ fel a továbbiakban

- A gépi kódot tartalmazó, jelen esetben .HEX kiterjesztésű fájl, amit a linker hoz létre az előzőleg külön-külön lefordított modulokból

3.2.3 Az első assembly programunk

Ezek után nézzük első assembly programunkat! A program folyamatábrája a 24. ábrán látható. A 3.1 fejezetben leírtak alapján hozzunk létre egy új



24. ábra

projektet, és a tanulókkal gyakorlásképpen gépeltessük be az alábbi programot:

```
; Egyszerű példa a próbapanel LED-jeinek működtetésére
;A panelon a JMP2-őt zárjuk rövidre, a JMP1-et pedig 2-3 állásba rakjuk fel.
;Ilyenkor a kétszínű LED-ek aktívak, a hétszegmenses kijelzők ki vannak
;kapcsolva.
;Az MPLAB-ban a "default radix"-ot hexa-ra kell állítani
;Ebben az esetben a külön nem jelölt számok hexában értendők, a B'szám'
;binárisként,
;a .szám (jó a D'szám' is, de az előbbi gyorsabb), decimálisként értelmezett
;A program annyit csinál csupán, hogy a 4db LED-et zöld színre kapcsolja.
;V1.1
;2005.10.15.
;Juhász Róbert
```

```
;-------------------------------------------------------------------------------------------------------
        LIST    P=16F84                 ;processzor típusa
#INCLUDE    "P16F84.INC"                ;Használd az ebben lévő szimbólumokat
 __CONFIG _XT_OSC&_CP_OFF&_WDT_OFF    ;Kvarc oszcillátor, ;kódvédelem
                                        ;ki, wdt ki
        ORG         0                   ;Kezdődjön 0h címen a program
GOTO        START               ;Ugorj a START címkére ORG 4
                    ;Megszakításoknak lefoglalt cím
;-------------------------------------------------------------------------------------------------------
START
        BANKSEL     TRISA               ;Váltsunk a TRISA-t tartalmazó bankba
        MOVLW       B'11111111'
        MOVWF       TRISA               ;PORTA INPUT LESZ
        MOVLW       B'00000000'
        MOVWF       TRISB               ;PORTB OUTPUT LESZ
        BANKSEL     PORTA               ;Váltsunk a PORTB-t tartalmazó bankba
;-------------------------------------------------------------------------------------------------------
MAIN
        MOVLW       B'01010101'         ;A zöld színnek megfelelő kombináció
        MOVWF       PORTB               ;Kiírás a PORTRA, LED-ek bekapcsolása
CIKLUS
        GOTO        CIKLUS              ;Önmagára ugrik, végtelen hurok
        END                 ;Forrásprogram vége (kötelező)
```

A program eleje egy rövid leírást tartalmaz annak funkciójáról, használatáról. Hasznos dolog a változások nyomon követésének érdekében, ha a programot verziószámmal, illetve dátummal látjuk el.

A program első sora a *LIST* direktívával kezdődik, ahol a mikrovezérlő típusát adhatjuk meg. A következő sorban a *#INCLUDE* direktívával a szükséges *include* fájlt adhatjuk meg a mikrovezérlőnkhöz. Ebben a fájlban definiálja a Microchip az egyes mikrovezérlőkhöz tartozó szimbólumokat, és a hozzárendelt értékeket. Pl. a 16F84-es tokhoz tartozó .INC fájl egy részlete:

```
;----- Register Files----------------------------------------------------
INDF            EQU     H'0000'
TMR0            EQU     H'0001'
PCL             EQU     H'0002'
STATUS          EQU     H'0003'
FSR             EQU     H'0004'
PORTA           EQU     H'0005'
PORTB           EQU     H'0006'
```

Ez által válik lehetővé, hogy az egyes regiszterek, bitek memóriabeli címét nem kell megjegyeznünk, szimbolikus nevekkel hivatkozhatunk rájuk. Az *__CONFIG* direktívával az ún. konfigurációs biteket állíthatjuk be. Ezek az égetés során kerülnek a mikrovezérlőbe, és a processzor konfigurálását végzik. A programunkban a következő konfigurációs biteket használjuk:

- *_XT_OSC:* a mikrovezérlő órajelforrása egy kvarckristály
- *_CP_OFF:* kikapcsoljuk a kódvédelmet
- *_WDT_OFF:* kikapcsoljuk az ún. „watchdog timer”-t

A kódvédelem kikapcsolása nagyon fontos, mert ellenkező esetben többször nem tudjuk égetni a mikrovezérlőt, ezért erre külön is hívjuk fel a tanulók figyelmét. Egyébként az arra szolgál, hogy megvédjük szellemi termékünket attól, hogy illetéktelenek is hozzáférjenek. A watchdog timer feladata pedig az, hogy az esetlegesen hibás memóriaterületre „tévedő” program esetén reszetelje a mikrovezérlőt. Ennek beillesztése csak már jól működő, kipróbált program esetén javasolt! A következő direktívával (*ORG-origin*) a fordítóprogramnak az mondjuk meg, hogy a memória melyik címétől kezdve fordítsa be a programot a memóriába. Az *„ORG 0”* megadása, tehát azt jelenti, hogy a programunk a memória 0-s címétől kezdve fordul be a memóriába. A következő *ORG* direktívával *(ORG 4)* az ún. megszakítási vektornak foglalunk le helyet a memóriában, ugyanis megszakítás esetén erre a címre fut a program. Ez a kis egyszerű program nem használ ugyan megszakítást, de a legtöbb esetben szükségünk lesz rá, ezért a tanulókkal ezt így gyakoroltassuk be. Az első utasításunk a *GOTO START*, ami azt jelenti, hogy a program a *START* nevű címkére ugorjon. A *START* címkénél ismét egy direktíva, a *BANKSEL* található. A PIC16F84 ugyanis bankszervezésű memóriát használ, így az egyes regiszterek (**SFR**) más-más bankban találhatók. A 16F84 ún.

memóriatérképe a 25. ábrán látható. Az adatmemória, mint már említettük alapvetően 2 részre tagolódik:

- Speciális funkciójú regiszterek (**SFR**)
- Általános célú regiszterek (**GPR**)

Ez a két rész 2 bankra oszlik, a 0-s és 1-es bankra. A **TRISA** és **TRISB** regiszterek az 1-es bankban, a **PORTA** és **PORTB** regiszterek pedig mellettük a 0-s bankban találhatók. Az előbbiek a portok irányát állítják be, míg az utóbbiak magukat a portokat jelölik. A 16F84-es tehát a külvilág felé két porttal rendelkezik.

Cím	Bank 0	Bank 1	Cím
00_H	INDF	INDF	80_H
01_H	TMR0	OPTION_REG	81_H
02_H			82_H
03_H	STATUS	STATUS	83_H
04_H	FSR	FSR	84_H
05_H	PORTA	TRISA	85_H
06_H	PORTB	TRISB	86_H
07_H			87_H
08_H	EEDATA	EECON1	88_H
09_H	EEADR	EEADR2	89_H
$0A_H$	PCLATH	PCLATH	$8A_H$
$0B_H$	INTCON	INTCON	$8B_H$
$0C_H$	68 darab általános célú regiszter (GPR)	A Bank 0 tükrözése	$8C_H$
$4F_H$			CF_H

25. ábra

A portokon keresztül tudunk különböző eszközöket működtetni (LED-ek, kijelzők, motorok, stb.), illetve ezeken keresztül tudunk információkat beolvasni a külvilág felől (pl. kapcsolók állapota). Ez attól függ, hogy a portot bemenetnek,

vagy kimenetnek konfiguráltuk. Az irány beállítása bitenként lehetséges, azaz egy adott porton belül lehetnek bemeneti és kimeneti lábak is. Az irány beállítására a *„TRIS”* regiszterek szolgálnak, mégpedig olyan módon, hogy amelyik portbit helyébe 1-t írtunk, az bemenet lesz (1→I, azaz INPUT), amelyik helyébe 0-t, az pedig kimenet lesz (0→O, azaz OUTPUT). A portok irányának helyes beállítása nagyon fontos, ugyanis, ha kimenetnek konfigurálunk egy olyan lábat, melyre pl. kapcsolót kötöttünk, – helyesen ez bemenet lenne – akkor, ha a kapcsoló nullába „húz”, a kimenet pedig 1-be, vagy fordítva a port lába tönkre megy! Ezt minden tanulóban tudatosítsuk. Mivel az iránybeállító és a port regiszterek különböző bankban vannak, ezért nagyon fontos, hogy mindig a megfelelő bankba váltsunk. A bankváltások elmaradása számos rejtélyes hiba forrása lehet, ezért erre is hívjuk fel a tanulók figyelmét! Miután a megfelelő bankba váltottunk a következő négy sorban beállítjuk a **PORTA**-t bemenetnek, a **PORTB**-t pedig kimenetnek. Ehhez a **TRISA**-ba csupa 1-est, a **TRISB**-be pedig csupa nullát kell betölteni. Itt egy újabb fontos dolgot kell megtanulnunk, nevezetesen azt, hogy a mikrovezérlőnk hogyan értelmezi a bevitt számokat. Kedvenc példám az „10” számsorozat. Ezt mi emberek tíznek értelmezzük, hiszen tízes számrendszerben gondolkodunk. A gép azonban nem tudja ezt, ő csak egy 1-est és egy 0-t lát. Amit értelmezhet binárisan, ekkor kettőt ér; hexadecimálisan, ekkor 16-ot ér; decimálisan, ekkor 10-et ér, stb. Emiatt külön jelölnünk kell az egyes számformátumokat. Az MPASM számformátumai az alábbi táblázatban láthatók:

Radix	**Megadás**	**Példa**
Decimális	D’<decimális szám>’ .<decimális szám>’.	D’125’ .125
Hexadecimális	H’<hexadecimális szám>’ 0x<hexadecimális szám>	H’1A’ 0x1A
Oktális	O’<oktális szám>’	O’777’
Bináris	B’<bináris szám>’	B’10001010’
Karakteres (ASCII)	A’<karakter>’ ’<karakter>’	A’J’ ‘J’

Amennyiben a projektünkben az alapértelmezett számformátumot (default radix) hexadecimálisra állítjuk, akkor csak a többi számformátumot kell külön jelölnünk, azaz pl. a 17 karaktersorozat 17_H-t jelöl! A hexadecimális számok írásánál arra ügyeljünk, hogy szám csak számjeggyel kezdődhet, ezért pl. az

F2 kifejezést 0F2-ként írhatjuk csupán! A portirányokat azért célszerű binárisan megadni, mert így azonnal látszik, hogy melyik lábat állítottuk bemenetnek vagy kimenetnek, illetve, ha módosítjuk a hardver kapcsolását, akkor a programban ezt nagyon egyszerűen javíthatjuk. Az irányregiszterek feltöltése az akkumulátor regiszteren keresztül történhet. A PIC mikrovezérlőknél az akkumulátort munkaregiszternek (**W**-work) nevezik. A **MOVLW** utasítás (Move Literal to W) egy konstanst – literál – tölt be a munkaregiszterbe, a **MOVWF** (Move W to File) utasítás pedig bemozgatja a fájlregiszterbe, ami most a **TRISA** és a **TRISB**. Fájregiszternek nevezzük mindazokat a regisztereket, amelyek az adatmemóriában találhatóak (akár általános célú, akár speciális funkciójú). Az irányregiszterek feltöltése után a *BANKSEL* direktívával visszaváltunk a portokat tartalmazó nullás bankba. A tulajdonképpeni programunk a *MAIN* (főprogram) címkénél kezdődik. A feladat az, hogy a próbapanelon található 4 darab kétszínű LED zöld fénnyel világítson. Ehhez tanulmányozzuk a tanulókkal a mellékletben (4. ábra) található próbapanel kapcsolási rajzát. A rajzon látható, hogy a LED-ek anódjai a *PORTB* kivezetéseire csatlakoznak egy-egy korlátozóellenálláson keresztül. A sorrend RB_7-től RB_0-ig: piros-zöld, piros-zöld, stb. Tehát ahhoz, hogy mind a négy LED zöld színnel világítson az RB_6, RB_4, RB_2, RB_0 kimenetekre logikai „1” (+5V), a többi lábakra pedig logikai „0” (0V) értéket kell kiküldeni. A portok jelszintje ún. TTL kompatíbilisek, a +5V körüli feszültség a logikai „1”, a 0V körüli pedig a logikai „0”. A zöld színek bekapcsolásához tehát a *01010101* bináris kombinációt kell kiküldeni a *PORTB*-re, ezt tartalmazza a főprogramban lévő két sor. Ezzel el is végeztük a feladatot, már csak a program leállításáról kell gondoskodnunk, mert ellenkező esetben a program végigfutna azokon a memóriahelyeken is, ahová nem írtunk programot. Ennek előre láthatatlan következményei lennének, amik adott esetben súlyos problémákat okoznának! A mikrovezérlőnk nem ismer külön leállító utasítást, ezért ezt nekünk külön kell leprogramoznunk. A megoldást az ún. *dinamikus stop* szolgáltatja. Létrehozunk egy végtelen hurkot úgy, hogy a program *önmagára* ugrik. A forrásprogram vége az *END* direktívával van lezárva. Ezt kötelező kitenni, hiszen ez mondja meg a fordítóprogramnak, hogy meddig kell olvasnia a forrásprogramot.

A kész forrásprogramokat fordítsák le a tanulók, és a gépi kódot égessék be a mikrovezérlőbe. Ezután következhet a tesztelés a próbapanelon. Ügyeljünk

arra, hogy minden tanulónak sikerüljön a fordítás, égetés, és tesztelés. A kezdeti sikerélmények ugyanis nagymértékben befolyásolják a tanulók további érdeklődését a téma iránt, ösztönzi őket arra, hogy újabb dolgokat tanuljanak meg, még szellemesebb programokat írjanak. Miután mindenkinek sikerült a tesztelés, kiadhatjuk az önállóan elvégzendő feladatokat. A feladat pedig úgy szól, hogy készítsünk további két programot, amelyekben a LED-ek színét pirosra, illetve sárgára lehet állítani. A két másik programot más-más néven mentsék el (javasolt a *<monogram>_piros*, illetve a *<monogram>_sarga* elnevezés). A projektben ne felejtsük el átállítani a forrásfájl nevét. Ezzel gyakoroljuk az MPLAB használatát. A tanulókat hagyjuk önállóan dolgozni felhívva rá a figyelmüket, hogy probléma esetén bármikor segítünk! A feladat megoldása a kapcsolási rajz, és a mintaprogramunk tanulmányozásával tehető meg. A piros színekhez most az RB_7, RB_5, RB_3, RB_1 kimenetekre kell logikai „1” értéket kapcsolni. A sárga színt additív színkeveréssel hozzuk létre, azaz bekapcsoljuk a LED-ek piros és zöld anódjait is. A programban ehhez csak azt kell módosítani, hogy milyen kombinációt kell kiküldeni a *PORTB*-re. Most látjuk csak igazán, hogy miért sokkal jobb a portra kiírt információt binárisan megadni!

3.3 Utasítások, direktívák, elágazások, bemenetek

A harmadik foglalkozásunk elején tekintsük át az eddigi ismereteinket! Az előző alkalommal megismerkedtünk néhány *utasítással* és *direktívával.* Az MPASM igen sokfajta és jól használható direktívát kínál fel a hatékony programozáshoz. Néhány gyakran használt direktíva az 1. táblázatban olvasható.

MPASM direktívák		
Név	**Leírás**	**Megadás**
BANKSEL	Bankválasztó utasítás generálása	BANKSEL <címke>
CBLOCK	Konstans definíciós blokk kezdete	CBLOCK [<kifejezés>]
__CONFIG	Konfigurációs bitek beállítása	__CONFIG <kifejezés> vagy __CONFIG <cím>, <kifejezés>
DB	Bájt definiálása	DB <kifejezés> [,<kifejezés>,…,<kifejezés>]
DE	EEPROM-ba írandó adatok	DE <kifejezés> [,<kifejezés>,…,<kifejezés>]
DT	Táblázat megadása	DT <kifejezés> [,<kifejezés>,…,<kifejezés>]
#DEFINE	Szöveghelyettesítő címke definiálása	#DEFINE <név>[[(<arg>,..,<arg>)]<érték>]
END	Forrásprogram vége	END
ENDC	Konstansblokk vége	ENDC
EQU	Assembly konstans definiálása	<címke> EQU <kifejezés>
#INCLUDE	Forrásfájl beillesztése	#INCLUDE <<fájlnév>> #INCLUDE “<fájlnév>”
LIST	Lista opciók	LIST [<opció>[,…,<opció>]]
ORG	Programkezdet	<címke> ORG <kifejezés>

1. táblázat

A direktívák teljes listája az MPLAB súgójából elérhető: *Help>MPASM Assembler Help>Reference>Directive Summary.*

Mivel a PIC16F84 ún. **RISC** mikroprocesszor (Reduced Instruction Set – csökkentett utasításkészletű), ezért összesen 35 darab egyszerű, könnyen megtanulható utasítást ismer (2. táblázat). Az utasítások felépítése sajátos logikára utal. Ennek gyökereit a mikrovezérlő felépítésében (architektúrájában) kell keresni. A mai gépek az ún. egycímes struktúrát használják. Első elgondolás alapján, mivel a műveletek legtöbbjének két operandusa és egy eredménye van a négycímes utasítás felépítése a következőképpen néz ki:

Műveleti kód	Operandus címe	Operandus címe	Eredmény címe	Folytatás címe

Azonban így az utasításaink szélessége nagyon nagy lenne, amihez ráadásul igen széles memória is kellene. Próbáljuk valahogyan redukálni az utasításaink szélességét!

Háromcímes utasítás:

Műveleti kód	Operandus címe	Operandus és eredmény címe	Folytatás címe

Az eredményt itt az egyik operandus helyére írtuk. Amennyiben a művelet elvégzése után is kíváncsiak vagyunk mindkét operandusra, akkor szükséges a mozgató (MOVE) utasítás bevezetése.

Kétcímes utasítás:

Műveleti kód	Operandus címe	Operandus és eredmény címe

Ebben az esetben elhagytuk a folytatás címét. Felmerül a kérdés, hogy ebben az esetben honnan tudjuk, hogy hol kell folytatni a programot. Ennek megoldására bevezettek egy különleges regisztert, amelyet utasításszámlálónak (IP – Instruction Pointer), vagy programszámlálónak (PC – Program Counter). Az előbbi elnevezés sokkal inkább kifejezi a regiszter feladatát, mégis – mint számtalan más esetben – a kevésbé találó kifejezés terjedt el a szakirodalomban. A PIC mikrovezérlőkben is a PC elnevezés terjedt el, ezért mi is ezt a terminológiát fogjuk használni. A programszámláló jelöli mindig a soron következő utasítást. A PC tulajdonképpen egy szinkron soros számlánc, melynek értéke minden utasítás lehívása után eggyel növekszik.

Mnemonik Operandus	Leírás	Cik-lus	14 bites kód MSB LSB	Állított jelző-bitek	Meg-jegy-zés
Bájt orientált fájlregiszter műveletek					
ADDWF f,d	W és f összeadása	1	00 0111 dfff ffff	C,DC,Z	1,2
ANDWF f,d	W és f ÉS kapcsolata	1	00 0101 dfff ffff	Z	1,2
CLRF f	f törlése	1	00 0001 1fff ffff	Z	2
CLRW -	W törlése	1	00 0001 0xxx xxxx	Z	
COMF f,d	f komplementálása	1	00 1001 dfff ffff	Z	1,2
DECF f,d	f csökkentése	1	00 0011 dfff ffff	Z	1,2
DECFSZ f,d	f csökkentése és ugrás, ha 0	1(2)	00 1011 dfff ffff		1,2,3
INCF f,d	f növelése	1	00 1010 dfff ffff	Z	1,2
INCFSZ f,d	f növelése és ugrás, ha 0	1(2)	00 1111 dfff ffff		1,2,3
IORWF	f és W VAGY kapcsolata	1	00 0100 dfff ffff	Z	1,2
MOVF f,d	f mozgatása	1	00 1000 dfff ffff	Z	1,2
MOVWF f	W mozgatása f-be	1	00 0000 1fff ffff		
NOP -	nincs művelet	1	00 0000 0xx0 0000		
RLF f,d	forgatás balra az átvitelbiten keresztül	1	00 1101 dfff ffff	C	1,2
RRF f,d	forgatás jobbra az átvitelbiten keresztül	1	00 1100 dfff ffff	C	1,2
SUBWF f,d	W kivonása az f-ből	1	00 0010 dfff ffff	C,DC,Z	1,2
SWAPF f,d	az f alsó és felső 4 bitjének cseréje	1	00 1110 dfff ffff		1,2
XORWF f,d	W és f kizáró-vagy kapcsolata	1	00 0110 dfff ffff	Z	1,2
Bit orientált fájlregiszter műveletek					
BCF f,b	az f adott bitjének törlése	1	01 00bb bfff ffff		1,2
BSF f,b	az f adott bitjének 1-be billentése	1	01 01bb bfff ffff		1,2
BTFSC f,b	a bit tesztelése és ugrás, ha 0	1(2)	01 10bb bfff ffff		3
BTFSS f,b	a bit tesztelése és ugrás, ha 1	1(2)	01 11bb bfff ffff		3
Konstans és vezérlésátadó műveletek					
ADDLW k	konstans hozzáadása a W-hez	1	11 11x kkkk kkkk	C,DC,Z	
ANDLW k	W és egy konstans ÉS kapcsolata	1	11 1001 kkkk kkkk	Z	
CALL k	szubrutin hívás	2	10 0kkk kkkk kkkk		
CLRWDT -	Watchdog Timer törlése	1	00 0000 0110 0100	$\overline{TO}$, $\overline{PD}$	
GOTO k	ugrás címkére	2	10 1kkk kkkk kkkk		
IORLW k	W és egy konstans VAGY kapcsolata	1	11 1000 kkkk kkkk	Z	
MOVLW k	konstans mozgatása a W-be	1	11 00xx kkkk kkkk		
RETFIE -	visszatérés a megszakításból	2	00 0000 0000 1001		
RETLW k	visszatérés szubrutinból egy konstanssal	2	11 01xx kkkk kkkk		
RETURN -	visszatérés szubrutinból	2	00 0000 0000 1000		
SLEEP -	váltás alvó módba	1	00 0000 0110 0011	$\overline{TO}$, $\overline{PD}$	
SUBLW k	W kivonása egy konstansból	1	11 110x kkkk kkkk	C,DC,Z	
XORLW k	W és egy konstans kizáró-vagy kapcsolata	1	11 1010 kkkk kkkk	Z	

2. táblázat

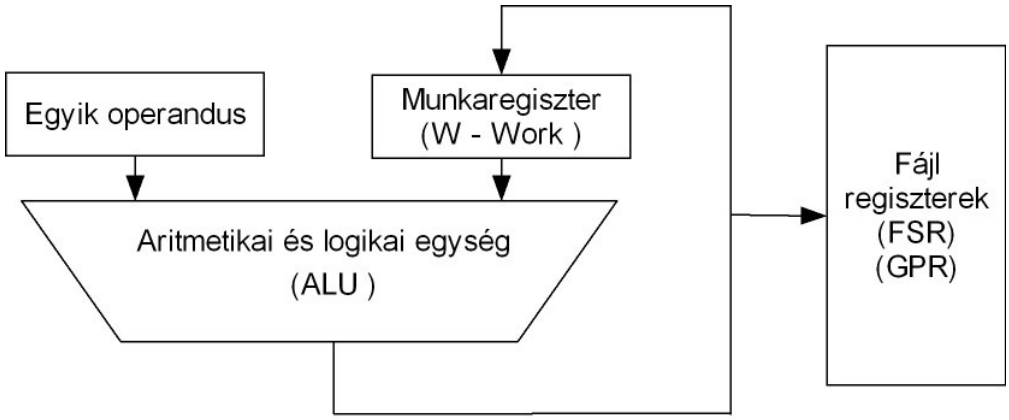
Abban az esetben, ha a program egy ún. elágaztató, vagy ugró utasításhoz – ilyen volt a *GOTO* – ér, akkor a programszámláló értéke egy adott címre cserélődik. Az ugrás után a PC az új címtől kezdve lépésről-lépésre tovább növeli tartalmát. Az ugró utasítás lehet:

- feltételes (programelágazás jöhet létre)
- feltétel nélküli

Egycímes utasítás:

Műveleti kód	Operandus címe

Ahhoz, hogy ezt meg tudjuk valósítani, a másik operandust valahol átmenetileg tárolni kell. Erre a célra szolgál az ún. akkumulátor regiszter (ACC), amit a PIC mikrovezérlőkben munkaregiszternek (**W** - Work) neveznek. A működés a 26. ábra alapján érthető meg.



26. ábra

A két operandusos művelet ezen a struktúrán csak úgy végezhető el, ha előbb az egyik operandust a **W** regiszterbe (akkumulátor) töltjük. Erre szolgálnak a **MOVLW** és **MOVF** utasítások. Miután lehoztuk az egyik operandust a második lépésben lehozzuk a másik operandust, és elvégezzük a tulajdonképpeni műveletet. A művelet eredménye kerülhet a W regiszterbe, vagy a másodikként lehozott operandus helyébe. Ez eltérés a hagyományos architektúrákhoz képest, mert ott csak az akkumulátorban keletkezhet az eredmény. Mivel az eredmény helye választható, ezért ezt az utasításokban is jelölni kell! Ezt az utasítás „d” (direction – irány) mezőjében tehetjük meg:

- d=*0* esetén a W-be kerül az eredmény
- d=*1* esetén a **fájlregiszterbe**

Az MPLAB program megengedi a következő jelölést is:

- d=*W* esetén a **W**-be kerül az eredmény
- d=*F* esetén a **fájlregiszterbe**

Ez utóbbi jelölés szemléletesebbé teszi a programjainkat. Az alábbi két példa egyértelművé teszi a cél kijelölését az utasításokban:

```
MOVLW   .12
ADDWF   PORTB,W
```

Az egyik operandusunk a 12, amit hozzáadunk a **PORTB**-hez, és az eredményt az akkumulátorban (**W**) tároljuk, illetve

```
MOVLW   .12
ADDWF   PORTB,F
```

esetén az eredmény a **PORTB** regiszterbe íródik.

Az eredmény helyének rossz megadása szintén számtalan rejtélyes hiba forrása lehet, amit nem lehet elégszer hangsúlyozni a tanulók felé!

Az akkumulátoron kívül az Aritmetikai és Logikai Egységhez szorosan kapcsolódik egy másik regiszter is, amelyik az éppen elvégzett műveletek eredményeképpen beállt állapotokról ad tájékoztatást. Ezt a regisztert feltétel, vagy státuszregiszternek (FLAG) nevezzük. Benne találhatók az egyes feltétel bitek. A PIC mikrovezérlőkben ezt **STATUS** regiszternek nevezik, felépítése a következő:

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C

Ezek közül számunkra a két legfontosabb jelzőbit a Z és a C betűvel jelzett:

- **Z**: Zero Bit, amely 1-be billen, ha valamely aritmetikai vagy logikai művelet eredménye nulla, egyébként nulla
- **C**: Carry/$\overline{Borrow}$ - átvitel/áthozat bit, amely összeadásnál (**ADDLW**, **ADDWF**) 1-be billen, ha a legnagyobb helyiértéken történt átvitel, egyébként nulla. Kivonásnál (**SUBWF**, **SUBLW**) negált áthozatként viselkedik, azaz értéke 0, ha történt áthozat, egyébként logikai 1. A forgatások is a carry biten keresztül történnek.

Az RP1 és az RP0 bitek a bankváltást végzik, a 16F84-be csak az RP0 működik, mivel csak két bank van. **A BANKSEL** direktíva ezeket a biteket állítja! Az R/W opció azt jelenti, hogy az adott bit írható és olvasható, az R pedig azt,

hogy csak olvasható. A mögötte lévő kifejezés a reszet utáni értéket adja meg. A carry és a zero bit értéke reszet után nem definiált, lehet akár 1 és 0 is.

Ezek után nézzük az elkészítendő feladatunkat. Az előző alkalommal megtanultuk hogyan lehet a kétszínű LED-eket különböző színre kapcsolni. A feladat most az, hogy a világítódiódák színét kapcsolók segítségével lehessen változtatni. A bekapcsolt szín lehet piros, zöld, vagy sárga. Ezt két kapcsolóval tudjuk kiválasztani az alábbi táblázat alapján:

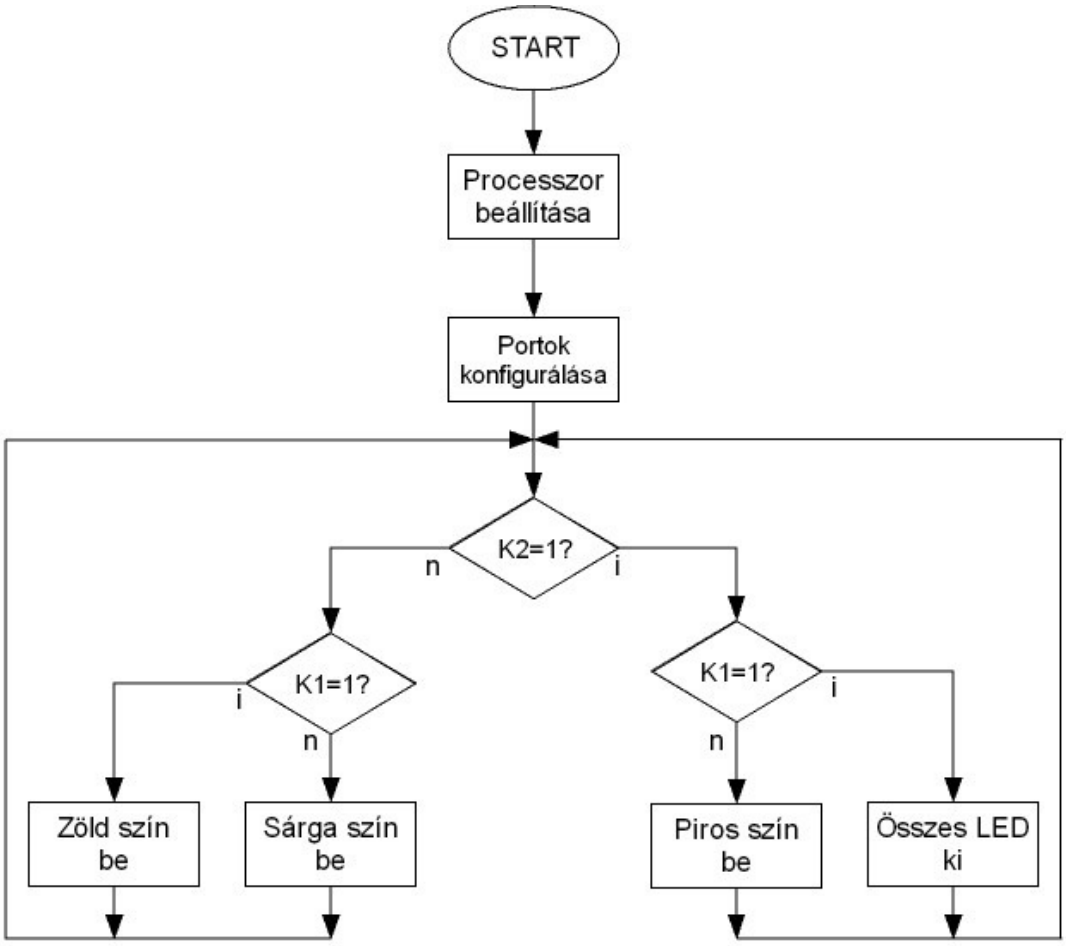
K_2	K_1	Szín
0	0	Összes LED kikapcsolva
0	1	piros
1	0	zöld
1	1	sárga

3. táblázat

A kapcsolási rajzot tanulmányozva láthatjuk, a 4 darab kapcsoló a **PORTA** kivezetéseire van kötve. Ezek közül tetszés szerint választhatunk kettőt, legyen most a K_2 az "**A"** port 3-as, a K_1 pedig a 4-es kivezetésére kötött kapcsoló. Bitekre a következőképpen lehet hivatkozni: <fájlregiszter>,<bit> - azaz jelen esetben **PORTA,3** és **PORTA,4**. Azért hogy ezt ne kelljen fejben tartani, használjunk szimbolikus elnevezést! Ezt a **#DEFINE** direktívával tehetjük meg a legkényelmesebben:

#DEFINE K2 PORTA,3

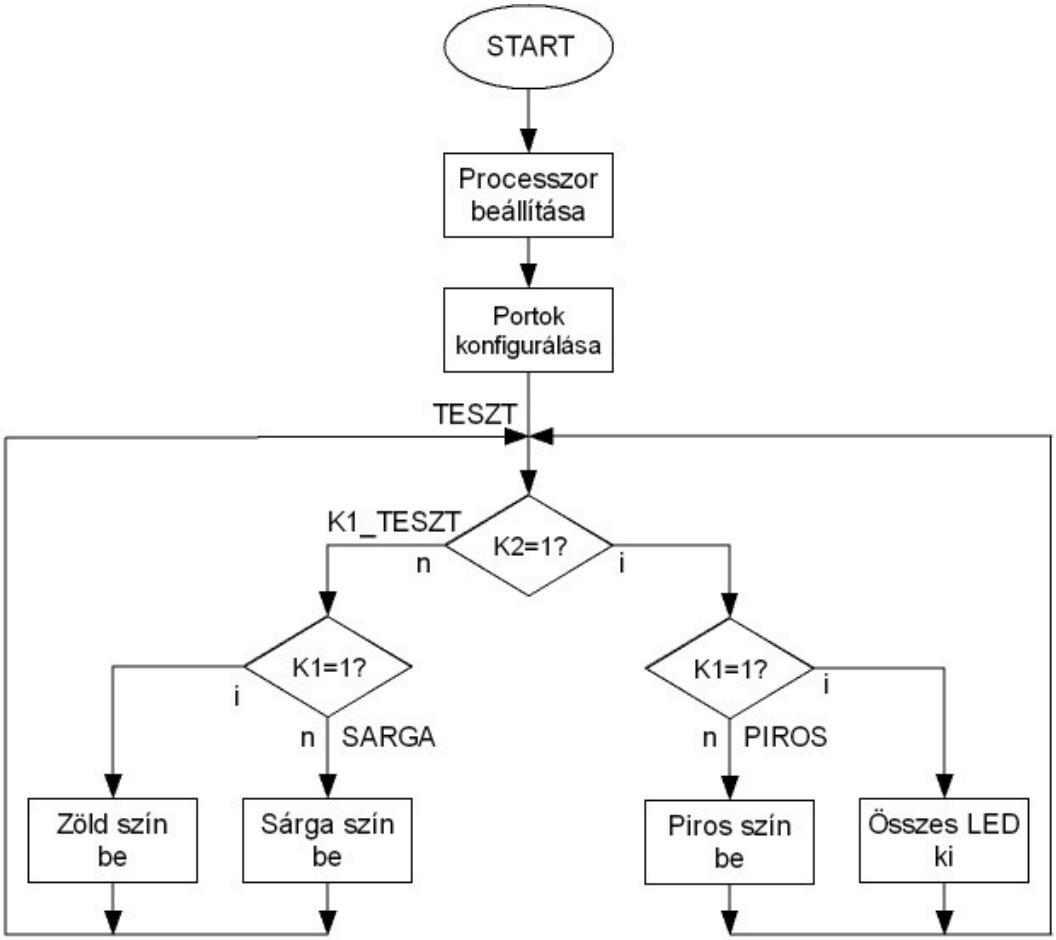
Először a folyamatábra módszernél tanultak alapján a tanulók készítsék el önállóan a program folyamatábráját! Természetesen szükség esetén segítsünk nekik, de inkább most az önálló kreatív gondolkodásra fektessük a hangsúlyt. A program folyamatábrájának egyik lehetséges megoldását a 27. ábra mutatja. Miután elkészült a logikailag helyes folyamatábra, a program találkozási pontjainak, valamint az elágazások egyik ágának szimbolikus nevet kell adni! Az elnevezések mindig „beszédesek” legyenek, törekedjünk az értelmezhetőségre és a tömörségre. A programban egy csomópont található, ahol a kapcsolók állásának tesztelése kezdődik. Adjuk ennek a csomópontnak a **TESZT** címkét. Ezután az elágazásokat nevezzük el. Egyik gyors módszer, ha a folyamatábrában konzekvensen valamelyik ágat (igen, vagy nem ág) címkézzük. Válasszuk most a *nem* ágat. Ennek megfelelően az első nem ágnak legyen **K_1TESZT** a neve (K1 kapcsolót teszteljük ugyanis). Baloldalon a második nem ágnál a sárga szín világít, legyen ennek a neve **SARGA**. A jobboldalon egy igen ágat találunk, mégpedig azt, ahol a piros fény világít, így ennek a neve **PIROS**.



27. ábra

Ezek alapján készíthetjük el a 28. ábrán látható végleges folyamatábrát (figyelem: a kapcsolók aktív esetben 0-t adnak). A kérdés már csak az, hogy ezeket az elágazásokat hogyan írhatjuk le assembly nyelven! Ehhez az utasításkészletet tanulmányozva két utasítást találhatunk:

- **BTFSC** (Bit Test File Skip if Clear) – azaz egy adott fájlregiszter bitjének tesztelése, és a soron következő utasítás átlépése, ha a bit logikai értéke 0 volt
- **BTFSS** (Bit Test File Skip if Set) – azaz egy adott fájlregiszter bitjének tesztelése, és a soron következő utasítás átlépése, ha a bit logikai értéke 1 volt



28. ábra

Ezekkel az utasításokkal a kapcsolók állását meg tudjuk vizsgálni, hiszen a kapcsolók a **PORTA** fájlregiszter egyes bitjeit jelentik, amit mi szimbolikusan K1-nek és K2-nek neveztünk el. Mindezek figyelembe vételével készíthetjük el programunk forráskódját assembly nyelven. A programunk eleje tulajdonképpen ugyanaz, mint az előbb, hiszen a processzorbeállítások és a portok konfigurálása nem változik. Ami újdonság, az a kapcsolók szimbolikus elnevezése. Ez a programban bárhol elhelyezhető. A programozók általában kétféle gyakorlatot követnek, vagy a program elején, vagy a legvégén definiálják ezeket. Véleményem szerint célszerűbb az elejére tenni, ezért a programjaink során ezt a gyakorlatot fogjuk követni. A program egyik lehetséges forráskódja a következőképpen néz ki:

;Egyszerű példa a próbapanel kapcsolóinak és LED-jeinek működtetésére
;A panelon a JMP2-őt zárjuk rövidre, a JMP1-et pedig 2-3 állásba rakjuk fel.

```
;Ilyenkor a kétszínű LED-ek aktívak, a hétszegmenses kijelzők ki vannak kapcsolva.
;Az MPLAB-ban a "default radix"-ot hexa-ra kell állítani
;Ebben az esetben a külön nem jelölt számok hexában értendők, a B'szám' binárisként,
;a .szám (jó a D'szám' is, de az előbbi gyorsabb), decimálisként értelmezett
;A program a kapcsolók állásától függően, a 4db LED-et zöld színre, piros színre,
;sárga színre, vagy kikapcsolja.
;V1.0
;2006.02.13.
;Juhász Róbert

        LIST    P=16F84
#INCLUDE    "P16F84.INC"            ;Használd az ebben lévő szimbólumokat
        __CONFIG    _XT_OSC&_CP_OFF&_WDT_OFF    ;Kvarc oszcillátor,
;kódvédelem ki, wdt ki

#DEFINE     K1      PORTA,4     ;K1 a PORTA 4-es lába lesz
#DEFINE     K2      PORTA,3     ;K2 a PORTA 3-as lába lesz

        ORG   0                 ;Kezdődjön 0h címen a program
        GOTO START              ;Ugorj a START címkére

        ORG   4                 ;Megszakításoknak lefoglalt cím
START
        BANKSEL     TRISA       ;Váltsunk a TRISA-t tartalmazó bankba
        MOVLW       B'11111111'
        MOVWF       TRISA       ;PORTA INPUT LESZ
        MOVLW       B'00000000'
        MOVWF       TRISB       ;PORTB OUTPUT LESZ
        BANKSEL     PORTA       ;Váltsunk a PORTA-t tartalmazó bankba
TESZT
        BTFSS       K2          ;K2=1?
        GOTO        K1_TESZT    ;Nem

        BTFSS       K1          ;K1=1?
        GOTO        PIROS       ;Nem
```

```
        MOVLW     B'00000000'  ;Összes LED ki
        MOVWF     PORTB        ;Kiírás a PORTRA, összes LED ki
        GOTO TESZT
K1_TESZT
        BTFSS     K1           ;K1=1?
        GOTO      SARGA        ;Nem
        MOVLW     B'01010101'  ;A ZÖLD színnek megfelelő kombináció
        MOVWF     PORTB        ;Kiírás a PORTRA, LED-ek bekapcsolása
        GOTO      TESZT        ;Vissza a teszthez
PIROS
        MOVLW     B'10101010'  ;A PIROS színnek megfelelő kombináció
        MOVWF     PORTB        ;Kiírás a PORTRA, LED-ek bekapcsolása
        GOTO      TESZT        ;Vissza a teszthez
SARGA
        MOVLW     B'11111111'  ;A SÁRGA színnek megfelelő kombináció
        MOVWF     PORTB        ;Kiírás a PORTRA, LED-ek bekapcsolása
        GOTO      TESZT        ;Vissza a teszthez

        END                    ;Forrásprogram vége (kötelező)
```

Az újdonság a programban a **TESZT** nevű címkénél kezdődik. A **BTFSS K2** utasítás teszteli, hogy a K2 kapcsoló értéke logikai 1, és a soron következő utasítást átlépi, ha igen, azaz az alatta lévő sor lesz a *nem* ág (**GOTO K1_TESZT**)! Az *igen* ágon a K1 kapcsoló vizsgálata következik (**BTFSS K2**). A *nem* ágon a **PIROS.** Mint látható a programban leírtunk két címkére ugrást, de még nem hoztuk létre! Ezt a két címkét a programban bárhol elhelyezhetjük, célszerű azonban sorrendben beírni. A címkék mögötti tartalmat majd később kitöltjük! Így már a programot hibaüzenet nélkül le tudjuk fordítani. Programírás közben gyakran fordítsunk, mert ezzel egyrészt ellenőrizzük a helyes gépelést, másrészt lementjük az eddig megírtakat. Térjünk vissza a programunkhoz. A K1 tesztelése után az igen ágon (a **GOTO PIROS** után) az összes LED kikapcsolása következik, hiszen egyik kapcsoló sem aktív. A **B'00000000'** helyett egyszerűen **0**-t is írhattunk volna, sőt a két sort egyetlen **CLRF PORTB** (Clear File – fájl nullázása) utasítással is helyettesíthettük volna. Ez azonban itt

nem célszerű, mert ha más színkombinációt akarunk kiírni, akkor az előbbi módszerrel sokkal egyszerűbb! Miután kikapcsoltuk az összes LED-et, a folyamatábrát követve vissza kell ugranunk a **TESZT** címkére (**GOTO TESZT**). Ezután folytassuk a programot a kimaradt címkékkel! Először a **K1_TESZT** nevű következik. Itt ismét a K1 kapcsolót teszteljük, és a *nem* ágon **SARGA** következik (**GOTO SARGA**), egyébként bekapcsoljuk a zöld színt. Ismét keletkezett egy címkénk, a **SARGA** nevű, ezt is hozzuk létre a **PIROS** után! Ezek után már csak annyi a dolgunk, hogy az egyes színek bekapcsolását a megfelelő címkékhez beírjuk. Ne felejtsük el, hogy a végén a **TESZT** címkére kell ugrani! A feladatot az *igen* ágon végighaladva is megoldhatjuk, azonban most látszik igazán, hogy a *nem* ág a kedvezőbb, ha a **BTFSS** és **BTFSC** utasításokat használjuk. Ezzel készen is van a programunk, jöhet a tesztelés.

Természetesen ennél a feladatnál számos más eltérő helyes megoldás is lehet. Hagyjuk a tanulókat szabadon, önállóan gondolkodni, ne erőltessük rájuk a mi gondolatmeneteinket, csak elvi hibák esetén korrigáljuk. A későbbiek során látni fogjuk, hogy mennyi önálló, nagyszerű gondolatokkal állnak elő a tanulók, amelyeket a tanár is beépíthet saját struktúrái közé! Ez a feladat általában már izgalmas a kezdők számára, mindenki látni szeretné a gyakorlatban is munkája gyümölcsét. Itt már versenyhelyzetek is ki szoktak alakulni, hogy kinek sikerül elsőre hibátlanul megoldania a feladatot. Arra ügyeljünk, hogy a foglalkozás végére mindenkinek legyen kipróbált működőképes programja!

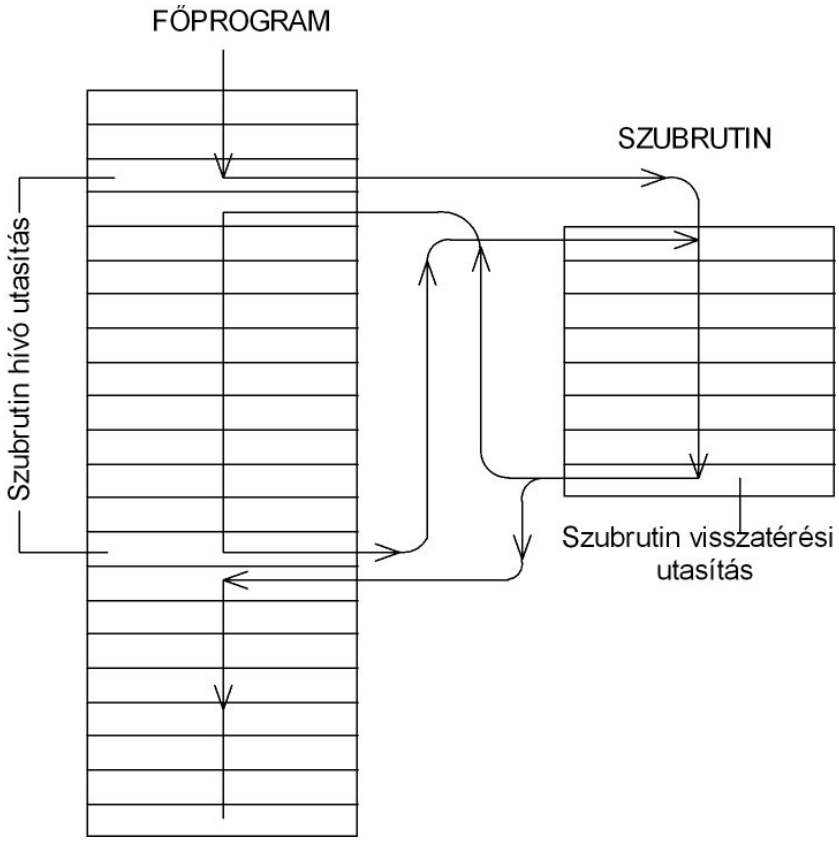
3.4 Szubrutinok, időzítés

A harmadik foglalkozásunk elején tegyünk egy rövid áttekintést az eddig tanultakról. A most következő problémakör szinte minden tanuló számára izgalmas, hiszen valamilyen izgő-mozgó programot fogunk készíteni. Amellett, hogy látványos lesz a programunk, a tanulók megismerkednek a szubrutin fogalmával és a ciklusszervezéssel! A feladat úgy szól, hogy írjunk futófény programot a 4 darab LED-re úgy, hogy a zöld szín fusson balról jobbra. A váltások közötti időtartam 500ms legyen. Könnyen belátható, hogy a programban nem kell mást tenni, mint az adott kombinációt kiküldeni a portra, majd várni, kiküldeni a következő kombinációt, megint várni, és így tovább. A program során a várakozást tehát többször is felhasználjuk. Ilyenkor nem

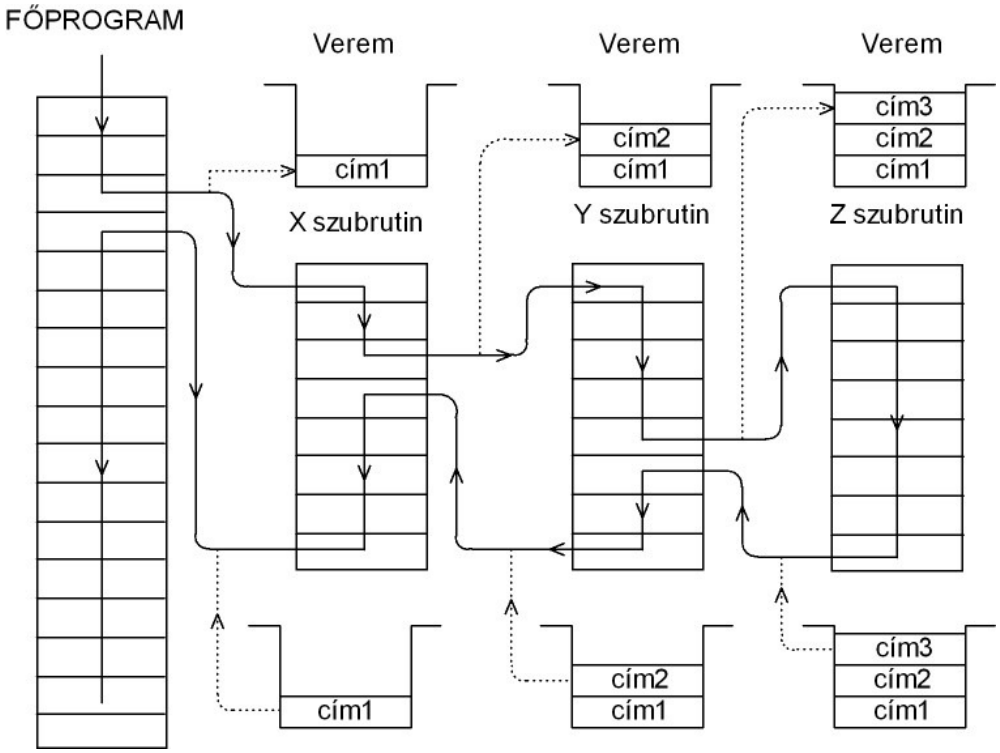
célszerű ugyanazt a programrészletet mindannyiszor megírni, hanem elég hivatkozni rá! Erre a célra szolgálnak a *makró*k és a *szubrutin*ok. A szubrutin és a makró között az az alapvető különbség, hogy a makró minden hívás helyére befordul (annyiszor foglalja a helyet a memóriában), míg a szubrutin csak egyszer foglalja a helyet. A szubrutinnak viszont az a hátránya, hogy sok „adminisztrációs" feladattal jár, mint azt a későbbiekben látni fogjuk.

Szubrutin: a programokban gyakran előforduló programrészek, amelyek a program különböző helyein felhasználhatók – meghívhatók –, de csak egyszer írták meg. A szubrutint a főprogramba két utasítás segítségével ékelik be (29. ábra):

- szubrutinhívó utasítás
- szubrutin visszatérési utasítás



29. ábra

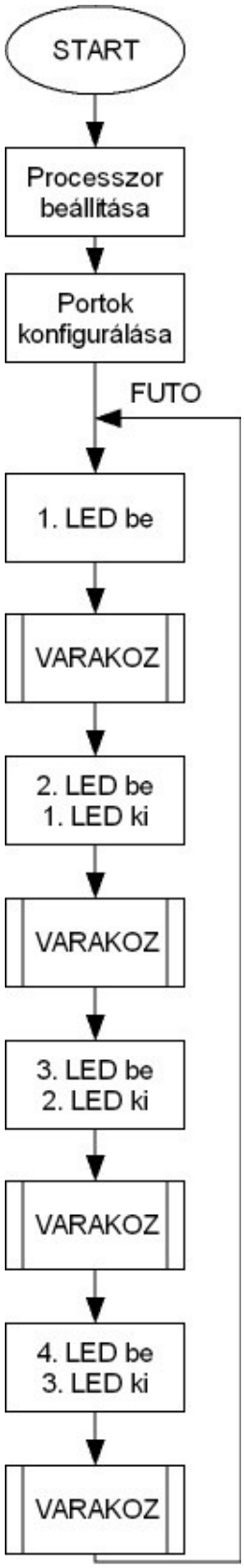


30. ábra

A szubrutinhívó utasítás a programszámlálóba (PC) a szubrutin kezdőcímét írja be, azaz a program lépésenként innen folytatódik. A szubrutin visszatérési utasítás pedig a visszatérési címet írja be a PC-be, amit még a szubrutinra ugrás előtt elmentenek. A visszatérési cím a szubrutint hívó utasítás utáni cím, ugyanis minden utasítás lehívása után a PC értéke eggyel növekszik, így a hívó utasítás utáni cím mentődik le. A visszatérési címek elmentésére szolgál a *veremtár*. A veremtár egy ún. LIFO (Last-In, First-Out), azaz az utoljára beleírt információt lehet először kiolvasni belőle. Ez az elrendezés alkalmassá teszi a visszatérési címek elmentésére. A PIC16F84-ben a veremtár is elkülönül (a Neumann-elvben egy közös memória van), nem csak az adatmemória. A verem itt 8 mélységű, és hardveresen működik, azaz a felhasználó nem tud közvetlenül hozzáférni. A 18-as sorozatban ez a felhasználók kérésére megváltozott, a verem szabadon írható. Nyolc egymásba ágyazott szubrutin után (ebbe a megszakítás is bele tartozik – lásd később) nem hívhatunk újabbat, mert a verem túlcsordul. Ez számos rejtélyes hiba okozója lehet. Ugyanúgy baj, ha alulcsordul a verem, azaz kiadunk egy szubrutin visszatérési utasítást, anélkül, hogy előtte hívtunk volna szubrutint. A mikrovezérlőnkben a **CALL** a hívó utasítás és a **RETURN**, ill. a **RETFIE** (lásd megszakítás) a

visszatérési utasítás. A többszörös megszakításhívásra a 30. ábra mutat példát

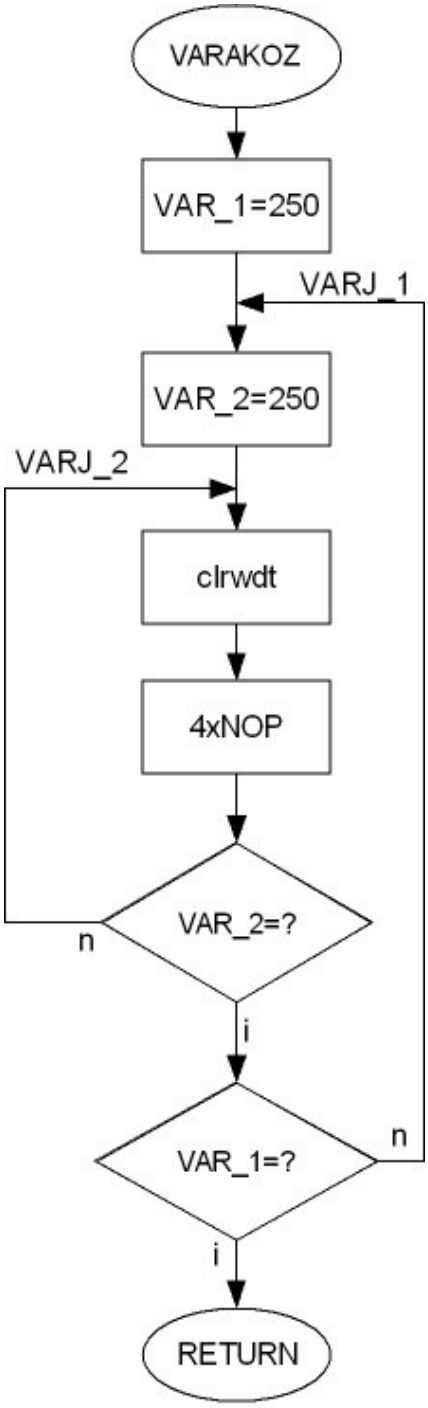
. A szubrutinhívás bizonyos „adminisztrációval” is jár, mint már említettük. Ugyanis, ha a főprogramból – vagy másik szubrutinból – hívunk egy szubrutint, akkor, ha a szubrutinban is használunk olyan regisztert, ami a főprogramban is szerepel, akkor az felülíródik! Első látásra az tűnik célravezetőnek, hogy használjunk különböző regisztereket, ami nem is jelent nagy gondot, mivel RISC processzor lévén sok az általános célú regiszter. Az egycímes struktúrából következően azonban a munkaregisztert (**W**) nem tudjuk kikerülni, ez szinte midig szükséges. Ugyanez a helyzet a jelzőbitekkel is, a szubrutin végrehajtása során felülíródnak. Ez a két dolog különösen a megszakítások kezelése esetén fog problémát okozni, ott majd részletesen kitérünk rá. Térjünk vissza a futófény programunkra. Összefoglalva tehát nincs más feladatunk, minthogy az 500ms idejű várakozást egy szubrutinban megírjuk! A program folyamatábrája a 31. ábrán látható. A világítódiódákat sorban egymás után kapcsoljuk be, közöttük pedig meghívjuk a *VARAKOZ* nevű szubrutint. Ezt egyelőre csak egy „dobozzal” jelöltük. Ennek a feladata, hogy létrehozza az 500ms-os időzítés. A kérdés, már csak az, hogyan hozzuk létre ezt az időtartamot. A mikrovezérlőben minden utasításnak van egy végrehajtási ideje. Ezt az időt a mikrovezérlőre kapcsolt órajel frekvenciája és az utasításhoz szükséges órajelciklusok száma határozza meg. A RISC mikroprocesszorokra az a jellemző, hogy az utasítások hossza 1 órajelciklus. Ez a PIC mikrovezérlőknél is így van, kivételt csak azok az utasítások jelentenek, ahol a PC tartama módosul. Ez abból adódik, hogy a processzor a futószalag elvet –



31. ábra

pipeline – használja. Ez azt jelenti, hogy az utasítások feldolgozása átlapoltan történik, amíg a következőt lehívjuk (fetch), addig az előzőt végrehajtjuk (execution).

Azonban, ha az utasításban a programszámláló értéke módosul (pl. feltételes ugró utasítás), akkor hiába hívtuk le a soron következő utasítást, a program nem azzal folytatódik! Ilyenkor az utasítás hossza két ciklus lesz. A 2. táblázatban látható, hogy a már használt BTFSC és BTFSS is ilyen utasítás. Az órajelet jelen esetben egy 4MHz-es kvarcoszcillátor biztosítja. Ez nagyon pontos frekvenciát biztosít (10^{-9}), tehát alkalmas lesz időzítési feladat ellátására. Az oszcillátor frekvenciájából áll elő a 4 fázisú órajel, ami az utasítás-feldolgozást ütemezi (lehívás, dekódolás, végrehajtás, kiírás). Ezekből kiszámítva egy utasítás végrehajtási ideje: 1µs. Tehát minden utasítás végrehajtása alatt eltelik 1µs. Mivel nekünk csak az utasítás végrehajtási ideje a fontos, ezért olyan utasítást célszerű választani, ami egyébként semmilyem műveletet nem végez! Erre alkalmas a **NOP** (No Operation) utasítás, ami annyit jelent, hogy nincs kijelölt műveletvégzés. A NOP utasítás gyökerei a gépi kódú programozásban keresendők. A gépi kódú programozásban ugyanis nem lehet szimbólumokkal hivatkozni – nincsenek címkék – a program egyes belépési pontjaira, a programozónak kell ezeket kiszámítania. Ez eddig nem is lenne baj, azonban amikor egy új utasítást kell beszúrni a programba, akkor a mögötte lévő összes cím eltolódik, az összes ugrási címet újra kell számolni. Ezért azokra a helyekre, amiről a programozó úgy gondolta, hogy oda biztosan kell még néhány sor, oda **NOP** utasításokat írt. Később



32. ábra

ezeket tetszés szerinti utasításokra cserélhette, anélkül, hogy a címek eltolódtak volna. Az 500ms időzítés létrehozásához: $\frac{500 \cdot 10^{-3}\,\mathrm{s}}{10^{-6}\,\mathrm{s}} = 500000$ utasításciklusra van szükség! Nyilván nem írhatunk le ennyi **NOP** utasítást, hiszen ez még a memóriában sem férne el! Ehelyett más megoldást kell keresnünk. Kézenfekvőnek tűnik, hogy egy adott számú **NOP** utasítást *n*-szer hajtsunk végre, ahol az *n* változó egy tetszőleges regiszter. Ezt a fajta megoldást nevezik ciklusszervezésnek. Ilyen struktúrát láthatunk a 23. ábra két jobboldali folyamatábráján. A felső ún. hátultesztelős, még az alsó elöltesztelős. Használjuk most a hátultesztelős megoldást. Szavakban kifejezve a feladat tehát az, hogy töltsünk fel egy változót az adott *n* értékre, futtassuk le a **NOP** utasításokat, csökkentsük 1-gyel a változó értékét, vizsgáljuk meg, hogy 0 lett-e a változó, mert ha igen, akkor kilépünk a ciklusból, máskülönben ismételünk. Most már csak erre alkalmas utasítást kell keresnünk. Tanulmányozva a 2. táblázat utasításait találunk olyat, amely ezt a két lépést egyesíti magában. Az utasítás neve: **DECFSZ** (Decrement File Skip if Zero) – csökkenti az adott fájlregiszter értékét, és ha nulla lett a soron következőt átlépi. Ezzel a ciklusunk már kialakítható utasítás szinten is. Már csak egy problémánk maradt. Az összes regiszterünk 8 bit széles, azaz 2^8=256 számértéket tudunk megkülönböztetni egymástól. Ez azt jelenti, hogy még így is több mint 1953 db **NOP**, vagy egyéb más utasítást kellene a ciklusmagba beleírnunk! A probléma tehát csak két egymásba ágyazott ciklussal oldható meg. Az így kialakított **VARAKOZ** szubrutinnak a folyamatábrája a 32. ábrán látható. A folyamatábrában található egy eddig nem említett utasítás, a **CLRWDT** (Clear Watchdog Timer). Erre a későbbiek során lehet szükség, amikor nem engedhető meg semmilyen körülmények között sem, hogy a program „lefagyjon”. Itt most ennek nincs jelentősége, csak azért említjük meg, hogy a tanulók megszokják ennek használatát. A szubrutinunk kialakításához két regisztert – változót – használtunk fel. A két regiszternek a **VAR_1** és **VAR_2** elnevezést adtuk. Ezt a két regisztert le kell foglalni az általános célú regiszterek (**GPR**). Ezt a legegyszerűbben a **CBLOCK** direktívával tehetjük meg. A 25. ábrán látható, hogy ez a terület a $0C_H$ címen kezdődik. Ne felejtsük el a felsorolás végén lezárni a blokkot az **ENDC** direktívával. Ezek alapján a tanulók önálló munka keretében készítsék a program forráskódját. Az egyik

lehetséges megoldás a következőképpen néz ki:

```
;Egyszerű futófény

;A panelon a JMP2-őt zárjuk rövidre, a JMP1-et pedig 2-3 állásba rakjuk fel.
;Ilyenkor a kétszínű LED-ek aktívak, a hétszegmenses kijelzők ki vannak
;kapcsolva.
;Az MPLAB-ban a "default radix"-ot hexa-ra kell állítani
;Ebben az esetben a külön nem jelölt számok hexában értendők, a B'szám'
;binárisként,
;a .szám (jó a D'szám' is, de az előbbi gyorsabb), decimálisként értelmezett

;Zöld színű futófény balról jobbra a 4 LED-en.

;V1.0
;2006.02.21.
;Juhász Róbert

        LIST    P=16F84
#INCLUDE    "P16F84.INC"                ;Használd az ebben lévő szimbólumokat
        __CONFIG _XT_OSC&_CP_OFF&_WDT_OFF    ;Kvarc oszcillátor,
;kódvédelem ki, wdt ki
        CBLOCK      0X0C
        VAR_1
        VAR_2
        ENDC
;-------------------------------------------------------------------------------------------------------
        ORG   0                     ;Kezdődjön 0h címen a program
        GOTO START                  ;Ugorj a START címkére
        ORG   4                     ;Megszakításoknak lefoglalt cím
;-------------------------------------------------------------------------------------------------------
START
        BANKSEL     TRISA           ;Váltsunk a TRISA-t tartalmazó bankba
        MOVLW       B'11111111'
        MOVWF       TRISA           ;PORTA INPUT LESZ
        MOVLW       B'00000000'
        MOVWF       TRISB           ;PORTB OUTPUT LESZ
        BANKSEL     PORTA               ;Váltsunk a PORTA-t tartalmazó bankba
```

```
;-------------------------------------------------------------------------------------------------------------
FUTO
        MOVLW       B'00000001'  ;Első zöld led
        MOVWF       PORTB        ;Kiírás a PORTRA, LED bekapcsolása

        CALL  VARAKOZ            ;500ms várakozás

        MOVLW       B'00000100'  ;Második zöld led
        MOVWF       PORTB        ;Kiírás a PORTRA, LED bekapcsolása

        CALL  VARAKOZ            ;500ms várakozás

        MOVLW       B'00010000'  ;Harmadik zöld LED
        MOVWF       PORTB        ;Kiírás a PORTRA, LED bekapcsolása

        CALL  VARAKOZ            ;500ms várakozás

        MOVLW       B'01000000'  ;Negyedik zöld LED
        MOVWF       PORTB        ;Kiírás a PORTRA, LED bekapcsolása

        CALL  VARAKOZ            ;500ms várakozás
        GOTO FUTO
;-----------------------------------------------------------------
VARAKOZ
        MOVLW       .250
        MOVWF       VAR_1        ;VAR_1 kezdeti értéke
VARJ_1
        MOVLW       .250
        MOVWF       VAR_2        ;VAR_2 kezdeti értéke
VARJ_2
        clrwdt                   ;WatchDog törlése
        NOP
        NOP
        NOP
        NOP
        DECFSZ      VAR_2,F      ;VAR_2 csökkentése, 0?
        GOTO VARJ_2              ;NEM, tovább
```

```
        DECFSZ      VAR_1,F     ;VAR_1 csökkentése, 0?
        GOTO VARJ_1             ;NEM, tovább
        RETURN                  ;IGEN, kiszáll
;-----------------------------------------------------------------
        END                 ;Forrásprogram vége (kötelező)
```

3.5 Ugrótáblák

A negyedik foglalkozás keretében összefoglaljuk eddigi ismereteinket és továbbfejlesztjük futófény programunkat látványos elemekkel. Ezek a programok szintén érdekesek a tanulók számára, ugyanakkor sok új programozási fogást is megtanulhatunk segítségükkel.

Először is térjünk vissza az alap futófény programunkra. Ezt a problémát úgy oldottuk meg, hogy az egyes kombinációkat (a futófény minden egyes állását) adott időközönként kiküldtük a portra. Ez négy LED-nél nem jelent gondot, azonban pl. 16 LED esetén már nem „elegáns” megoldás. Célszerű ilyenkor automatizálni a folyamatot. Itt jönnek segítségünkre az ún. *léptető* és *forgató* utasítások. Ezek különböző aritmetikai és logikai feladatok elvégzésére is alkalmasak. Tekintsük át először ezeket az utasításokat a 33. ábra alapján!

Léptető és forgató utasítások:

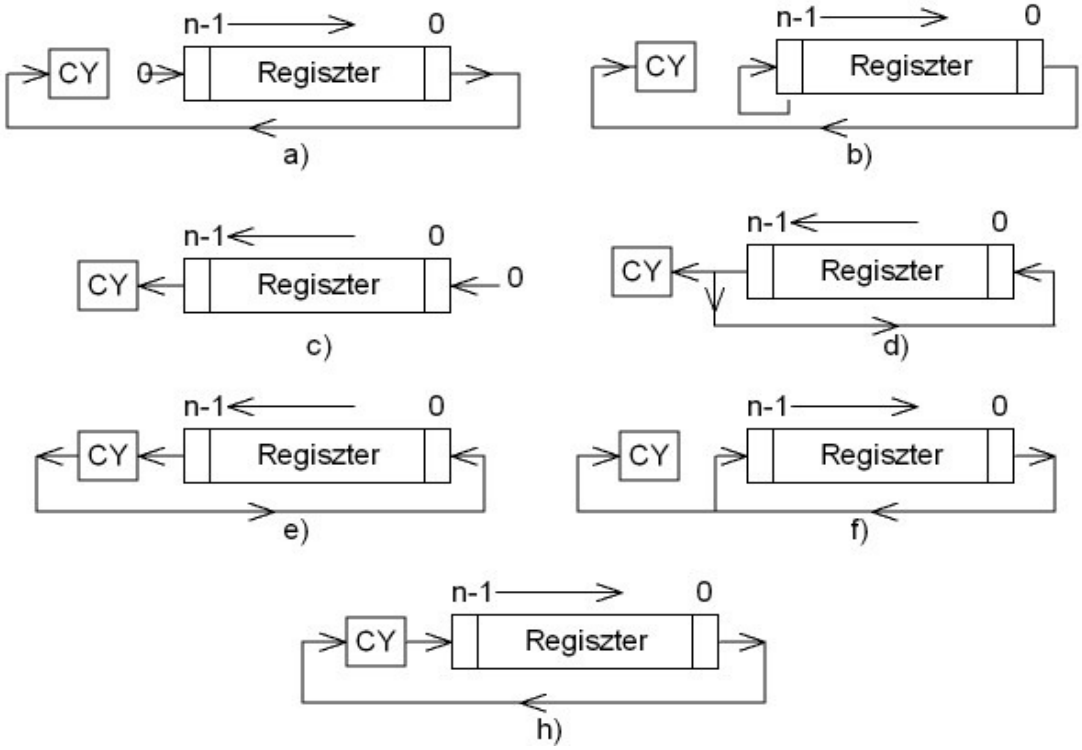
- **SRL** (Shift Right Logical) – logikai léptetés jobbra (a) ábra): minden bit egyet jobbra lép, a legfelső bit helyébe (n-1) „0” lép, a legalsó bit pedig az átvitel (CY) helyére lép,
- **SRA** (Shift Right Arithmetical) – aritmetika léptetés jobbra (b) ábra): minden bit egyet jobbra lép a legalsó bit az átvitel (CY) helyére íródik, a legfelső bit önmagába íródik. Vizsgáljuk meg, hogy milyen aritmetikai műveletet valósít meg a jobbra léptetés. Ábrázoljuk 5 biten a 10-es decimális számot és léptessük jobbra: 01010_B=10, léptetés után 00101_B=5, azaz kettővel elosztottuk a számot! Most már csak azt tisztázzuk, miért kell a legfelső bitet önmagába visszaírni. A megoldást a negatív számok ábrázolásában találjuk meg. A számítógépekben a negatív számok ábrázolására az ún. kettes komplemenst használják. Ennek lényege, hogy a legmagasabb helyiértékű bit jelöli, hogy pozitív, vagy negatív a szám. A nulla a

pozitív, az 1 a negatív számot jelenti. A mi példánkban a 2^4 helyiértékű bit az előjel, ami ha egy, akkor -2^4=-16-ot ér. Léptessük jobbra a -10-et: 10110_B, léptetés után 11011_B=-5. Most már érthető,

- **SLL** (Shift Left Logical) – logikai léptetés balra (c) ábra): minden bit egyet balra lép, az alsó bit helyébe „0” lép, a legfelső bit (n-1) az átvitelbit (CY) helyére íródik
- **SLA** (Shift Right Arithmetical) – aritmetikai léptetés jobbra (c) ábra): minden bit egyet balra lép, az alsó bit helyébe „0” lép, a legfelső bit (n-1) az átvitelbit (CY) helyére íródik. A különbség az előzőhöz képest csupán annyi, hogy ez állítja a „overflow” (túlcsordulás) bitet is. Ennek a bitnek a szerepe a már említett kettes komplemensű számábrázolásnál nyilvánul meg. Példákon keresztül nézzük, hogy meg miért van erre szükség. 4 biten 2-es komplemens kódban ábrázolva adjuk össze a -3-at és a -4-et, majd a -6-ot és a -7-et:

1101	1010
+1100	+1001
11001	10011

Az első esetben az átvitel felesleges, hiszen a keletkezett eredmény e nélkül is helyes (-8+1=-7), a második esetben átvitel nélkül pozitív



33. ábra

az eredmény, ami nem helyes. Ebből látszik, hogy az átvitelbit (carry) komplemens kódú ábrázolásnál nem alkalmas annak megmutatására, hogy az eredmény elfér-e az adott ábrázolási tartományban. Ehelyett a túlcsordulást (overflow) kell használni, amely a legmagasabb, és a közvetlenül előtte lévő helyiértékű bitekkel végzett kizáró-vagy művelet eredménye:

1101	1010
+1100	+1001
11001	10011
↓↓	↓↓
11	10

Ami az első esetben, $1 \oplus 1 = 0$, ami azt jelenti, hogy nincs túlcsordulás, míg a második esetben $1 \oplus 0 = 1$, azaz túlcsordulás keletkezett. Ilyenkor az overflow bitet is figyelembe kell venni, az eredmény helyesen: -16+2+1=-13.

- **RL** (Rotate Left) – forgatás balra (d) ábra): minden bit egyet balra lép, a legnagyobb helyiértékű bit (n-1) belép az alsó bit, valamint az átvitelbit helyére
- **RLC** (Rotate Left trought Carry) – forgatás balra átvitelbiten keresztül (e) ábra): minden bit egyet balra lép, a legnagyobb helyiértékű bit (n-1) belép az átvitel helyére, az utóbbi pedig a legkisebb helyiértékű bit helyére
- **RR** (Rotate Right) – forgatás jobbra (f) ábra): minden bit egyet jobbra lép, a legkisebb helyiértékű bit belép a legfelső (n-1) bit, valamint az átvitelbit helyére
- **RRC** (Rotate Right trought Carry) – forgatás jobbra átvitelbiten keresztül (g) ábra): minden bit egyet jobbra lép, a legalacsonyabb helyiértékű bit belép az átvitel helyére, az utóbbi pedig a legnagyobb helyiértékű (n-1) bit helyére

A PIC mikrovezérlőkbe, mivel ezek csökkentett utasításkészletű processzorok, csak a minimálisan szükséges utasításokat építették be ezek közül. A 29. ábrát tanulmányozva könnyen belátható elég csak a forgatás jobbra, ill. balra átvitelbiten keresztül utasításokat beépíteni, mert ezek segítségével bármelyik megoldható. A fotófény megvalósításához a forgatás balra (**RL**), vagy jobbra (**RR**) utasítás a megfelelő. Azonban a mikrovezérlőnkbe csak a forgatás

átvitelbiten keresztül utasítás van beépítve. A két utasítás elnevezése **RRF** és **RLF**, ami nem egészen találó, hiszen a carry-n keresztül forgat. Az a feladat tehát, hogy a forgatás átvitelbiten utasítást átalakítsuk forgatás utasításra. Balra forgatás esetén a 33 e) ábrát kell a 33 d) szerintire átalakítani. A két ábrát tanulmányozva ehhez az szükséges, hogy az átvitelbit értéke a forgatás előtt ugyanaz legyen, mint a legfelső bit értéke. A feladat megoldását bízzuk a tanulókra önálló munka keretében. A feladat megoldható a legfelső bit tesztelésével, és ennek megfelelően a carry törlésével, vagy 1-be állításával. Azonban létezik egy sokkal elegánsabb megoldás, amit mindenképpen mutassunk meg a tanulóknak. A lényege ennek az, hogy az első forgatás eredményét a munkaregiszterbe írjuk, ilyenkor a port nem változik, viszont a carry értéke a legfelső bit lesz. Ezután végrehajtunk még egy forgatást, és az eredményt most már a helyére írjuk:

```
RLF     PORTB,W     ;carry=legfelső bit
RLF     PORTB,F     ;forgatás a portra
```

Természetesen, ha azt szeretnénk, hogy egy adott szín fusson körbe, akkor ahhoz két ismételt forgatást kell végrehajtani. Mindez ciklusba foglalva:

```
        MOVLW   B'01000000'
        MOVWF   PORTB       ;első LED be
FOROG
        CALL    VARJ        ;várakozás
        RLF     PORTB,W     ;carry=legfelső bit
        RLF     PORTB,F     ;forgatás a portra
        RLF     PORTB,W     ;carry=legfelső bit
        RLF     PORTB,F     ;forgatás a portra
        GOTO    FOROG       ;végtelen hurok
```

Miután a tanulók megoldották ezt a feladatot, rátérhetünk a foglalkozás második felére, aminek keretében fényjátékot készítünk, mely során ismét sok hasznos új ismeretre tehetnek szert a tanulók. A feladat lényege az, hogy a kapcsolók állásától függően 16 különböző futófényt valósítsunk meg az alábbi táblázat alapján:

Kapcsolók	Feladat
0000	Összes LED kikapcsolt
0001	Piros futófény balról jobbra
0010	Zöld futófény balról jobbra
0011	Sárga futófény balról jobbra
0100	Piros oda-vissza futófény
0101	Zöld oda-vissza futófény
0110	Sárga oda-vissza futófény
0111	Piros „lyuk” fut balról jobbra
1000	Zöld „lyuk” fut balról jobbra
1001	Sárga „lyuk” fut balról jobbra
1010	Balról jobbra sorban bekacsolnak (úgy is marad) a piros LED-ek
1011	Balról jobbra sorban bekacsolnak (úgy is marad) a zöld LED-ek
1100	Balról jobbra sorban bekacsolnak (úgy is marad) a sárga LED-ek
1101	Balról jobbra bekapcsolnak, majd vissza elalszanak a pirosak
1110	Balról jobbra bekapcsolnak, majd vissza elalszanak a zöldek
1111	Szabadon választott variációk

A kapcsolók állásának lekérdezését, és ettől függően programelágazás létrehozását már tanultuk. Azonban ez a struktúra 4 kapcsoló esetén már meglehetősen bonyolulttá válik, nem is beszélve arról, ha növeljük a kapcsolók számát. A 19. ábrán látható folyamatábrákat tanulmányozva sokkal célszerűbbnek látszik az indexelt struktúra használata ezen feladat megoldásához. Ennek az a lényege, hogy a kapcsolóállások lehetséges kombinációinak megfelelően ugyanannyi programágat hozunk létre. Mindig az az ág fog lefutni, amennyi a kapcsolók állásának számértéke. Ehhez először a kapcsolóálláshoz tartozó megfelelő számértéket kell kiolvasni a **PORTA** regiszteréből:

7	6	5	4	3	2	1	0
PORTA							
-	-	-	K1	K2	K3	K4	

Látható, hogy a 4 kapcsoló a **PORTA** 4-1 lábára van kötve, azaz számértékileg nincs a helyén. Ez azonban a nemrég tanult forgató utasítással megoldható. Ezen kívül van még egy probléma, nevezetesen az, hogy a hasznos 4 bit mellet marad 1 bit (az "A" port 5 bites), aminek az értéke tetszőleges lehet, ami a számértékünket meghamisíthatja. A felesleges biteket tehát valamilyen módon

ki kell szűrni. Erre ad megoldást az ún. maszkolás. Az eredeti regiszterünket rákülgjük egy olyan maszkra, ami csak a számunkra hasznos helyen engedi át a biteket, a többit nullává teszi. Ez tulajdonképpen bitenkénti ÉS kapcsolatot jelent az eredeti és a maszk regiszter között!

0	**0**	**0**	**x**	**0**	**1**	**0**	**1**	**PORTA**
0	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**maszk**
-	**-**	**-**	**-**	↓	↓	↓	↓	
tilt				**itt átenged**				
0	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**eredmény**

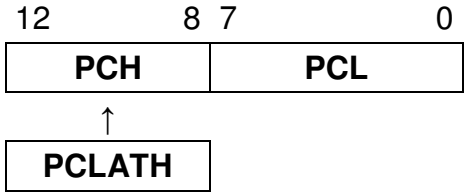
Mindez utasításszinten csupán ennyi:

```
RRF     PORTA,W     ;PORTA beolvasása, helyére forgatás
ANDLW   B'00001111' ;maszk
MOVWF   A_LATCH     ;eredmény tárolása
```

Az indexelt struktúra megvalósítását ún. ugrótábla segítségével oldjuk meg. Ennek lényege a programszámláló megfelelő számértékre való beállítása lesz. Az egyes alprogramokat megírjuk szépen egymás után, majd az ugrótáblában a kapcsoló állásának számértékétől függően az adott programra ugrunk. A gyakorlatban ez a következőképpen néz ki:

```
        ORG     300
UGRO_TABLA
        ADDWF   PCL,F       ;PC feltöltése
        GOTO    SEMMI       ;W=0 esetén ugrás a semmi címkére
        GOTO    EGY         ;W=1 esetén ugrás az EGY címkére
        GOTO    KETTO       ;W=2 esetén ugrás a kETTO címkére
        ……
        GOTO    TIZENOT     ;W=15 esetén ugrás a TIZENOT címkére
```

A tábla tulajdonképpen azt csinálja, hogy a programszámláló aktuális értékéhez hozzáadja a **W**-ben lévő számértéket (tehát a táblára ugrás előtt a kapcsoló értékét a **W**-be kell tölteni). Amennyiben ez 0, akkor a **GOTO SEMMI** sorra ugrik, és így tovább. Az összeadásban a **PCL** regiszter, a programszámláló alsó 8 bitje szerepel, ugyanis a 16F84 aritmetikai és logikai egysége csak nyolcbites operandusokkal képes műveleteket végezni. Ezért a 13 bites programszámlálót – 2 kB memória címzésére alkalmas – két részre bontották:



A programszámláló szubrutinhíváskor, visszatéréskor 13 bitesként viselkedik, a **PCH** értéke automatikusan állítódik. A **PCH** regiszterhez a felhasználó csak a **PCLATH** (Program Counter Latch High) átmeneti regiszteren keresztül fér hozzá. Amikor írjuk a **PCL** regisztert a **PCH** a **PCLATH** regiszterből töltődik fel, ezért a tábla hívása előtt a **PCLATH** regisztert fel kell tölteni az ugrótábla memóriabeli címének felső felével! Ez nagyon fontos dolog, ezért nyomatékosan hívjuk fel rá a tanulók figyelmét. Több tábla esetén minden táblára ugrás előtt be kell állítani a **PCLATH** regisztert. A beállítások elmaradása katasztrofális hibákhoz vezet! Jelen esetben az ugrótábla a 300_H címre lett befordítva, ennek a felső fele 3, tehát ezt kell betölteni a **PCLATH** regiszterbe. Ezt legegyszerűbben a **HIGH** direktívával tehetjük meg:

```
MOVLW   HIGH  UGRO_TABLA    ;a cím felső fele
MOVWF   PCLATH              ;PCLATH feltöltése
```

Még egy fontos dologra fel kell hívni a tanulók figyelmét. Az ugrótábla nem véletlenül lett a 300_H címre fordítva, hiszen láthattuk a **PCLATH** menet közben nem változik, így az ugrótáblánkban maximálisan 255 bejegyzés lehet. Abban az esetben, ha nem adunk meg kezdőcímet az ugrótáblánknak, akkor lehet, hogy laphatárra kerül, ami súlyos hibákhoz vezet! Ugyanígy fontos megjegyezni, ha az ugrótábla nincs teljesen kitöltve, – ez nagyon gyakori eset – akkor a tábla hívása előtt a **W**-ben nem lehet nagyobb szám, mint ahány bejegyzés van a táblában. Ezt a feltételt mindenképpen biztosítani kell, mert ez is számos rejtélyes hiba forrása lehet!

Most már a tanulók elegendő ismerettel rendelkeznek ahhoz, hogy a fényjáték programot önálló munka keretében teljes dokumentálással – fontos, hogy rászoktassuk a tanulókat erre is – együtt elkészítsék! A legjobb megoldásokat díjazzuk!

3.6 Megszakítások kezelése

Elérkezett az ötödik, alkalom a kezdők számára a mikrovezérlők oktatásának keretében. Röviden fussuk át az eddig tanultakat, kérdezzük meg a tanulókat, volt-e valamilyen problémájuk az eddig elhangzottakkal.

A feladat során, a próbapanelen – első generációs v2 verzió – található két darab hétszegmenses kijelző működtetésére írunk programot. Tapasztalataim szerint ez mindig felkelti a tanulók kíváncsiságát, legtöbben ugyanis mindjárt az elején erre akarnak programot írni. Az előzőekhez hasonló itt is sok új ismeretanyagot sajátítanak el a tanulók játékos formában.

A próbapanel kapcsolási rajzát tanulmányozva láthatjuk, hogy a megfelelő szegmensek párhuzamosan vannak a **PORTB**-re kapcsolva. Ahhoz, hogy a két kijelzőn eltérő karakter legyen látható, tehát valamilyen „trükkhöz" kell folyamodnunk. A megoldás abban rejlik, hogy „becsapjuk" a szemünket! A két különböző kijelzést egymás után felváltva rakjuk ki a kijelzőkre, miközben a másik kijelző sötét. Ebben az esetben természetesen az egyes kijelzőkön nem folyamatos, hanem villogó kijelzést kapunk. Az emberi szem azonban a villogást csak kb. 50Hz frekvenciáig érzékeli, felette folyamatosnak véli a kijelzést. Az ilyen váltott kijelzéses megoldást *multiplex* üzemmódnak nevezzük! A kijelzők multiplexelésének folyamatosan, az egész program során működnie kell. A feladat az, hogyan oldjuk meg ezt olyan módon, hogy a többi program feldolgozásának egészét a legkevésbé zavarja.

Erre ad megoldást a mikrovezérlők megszakítási rendszere. Ez a későbbiek során nagyon fontos lehet, sok problémakör egyszerűbb megoldását teszi majd lehetővé. Vizsgáljuk meg először tehát, hogy mi is az a magszakítás.

A számítógépek működése során nagyon gyakran következnek be olyan események, amelyek a feldolgozás szempontjából váratlannak tekintendők. Ezeket, a váratlan eseményeket úgy kell tudni lekezelni, hogy a feldolgozás egészét a legkevésbé zavarják. A keletkező események lehetnek:

- Szinkron események: a program futása szempontjából jól meghatározható helyen és időpontban várható események, azaz a program futása során mindig ugyanott következnek be (pl. túlcsordulás).

- Aszinkron várható események: a program futása során várható a bekövetkezése, de időpontja ismeretlen (pl. egy gomb lenyomása)

Az események kiváltó oka lehet:

- Maga a program: pl. futás közbeni hiba, megszakítás utasítás elhelyezése a programban, melynek kezelésére egy kiszolgáló eljárást indít el a processzor
- Hardver: pl. valamilyen periféria, amely az adatátvitel idejére a futó program ideiglenes felfüggesztését kezdeményezi

Az ilyen események feldolgozására szolgálnak a megszakítások, a megszakítási rendszer. A megszakítási kérelem tulajdonképpen egy jelzés a processzor számára valamely esemény bekövetkezéséről. A megszakítás a futó folyamat felfüggesztése a megszakítási kérelem hatására, annak kiszolgálása céljából.

A megszakítási kérelem feldolgozására egy hardver-szoftver együttes szolgál, amely elvégzi a kérelem kiértékelését és az ehhez szükséges tevékenységet. A megszakítási kérelem kiszolgálása tehát az a folyamat, amelyet a gép a megszakítási kérelem hatására elvégez.

Az utasítás-végrehajtástól független, külső események által, a program végrehajtását akadályozó eseményeket *megszakításoknak* (INTERRUPT), míg az utasítások szabályszerű végrehajtását akadályozó eseményeket *kivételeknek* (EXCEPTION) nevezzük. *Megszakítás* esetén a processzor a végrehajtás alatt álló utasítást szabályszerűen befejezi, majd a megszakítás kiszolgálása után a soron következő utasítással folytatja. *Kivétel* esetén a kiváltó esemény kiszolgálása után a processzor ismét megkísérli a megszakított utasítás végrehajtását.

Egyes esetekben a megszakítás lehetősége engedélyezhető (enable), vagy tiltható (disable). Az engedélyezés, vagy tiltás egy regiszter megfelelő bitjeinek beállításával történik, amit megszakítás *maszkolásnak* nevezünk. A hardver megszakítások között létezik olyan is, amit a felhasználó nem tilthat le (NMI – Non Maskable Interrupt), ez valamilyen súlyos hardverhiba esetén következik be. A szoftver megszakítások nem maszkolhatók!

A megszakítások kiszolgálása során több probléma is felmerül, amelyre megoldást kell keresnünk. Ezek a következők:

- A megszakítási kérelem keletkezési helyének megállapítása (ki kérte

a megszakítást)

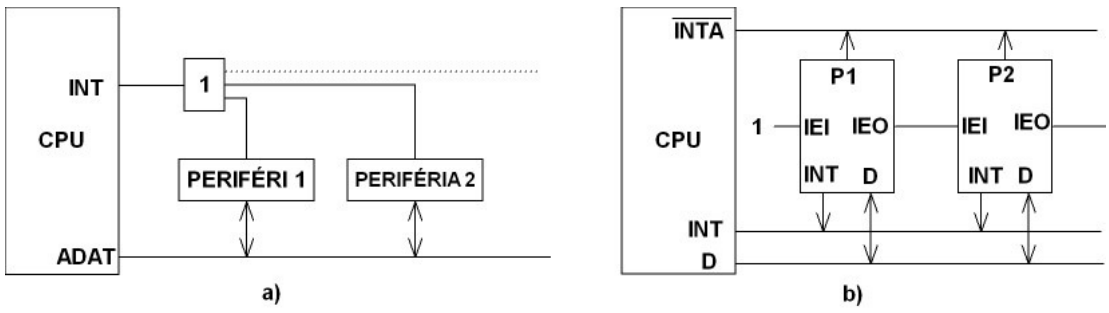
- Megszakítások maszkolása
- A megszakítási kérelem kiszolgálási sorrendjének megállapítása (prioritás)
- Többszörös megszakításkezelés megoldása

A megszakítási kérelem keletkezési helyének megállapítása történhet szoftveres, vagy hardveres úton.

Szoftver módszer: egy program – amely általában az operációs rendszer részét képző rutin – bizonyos időközönként megvizsgálja a szóba jöhető eszközök állapotjelzőjét. Ezt lekérdezéses megszakításkezelésnek – *polling interrupt* – nevezzük.

Hardver módszerek: egy megszakítás-vezérlő áramkör szabályozza program segítségével, vagy anélkül a megszakítások kiszolgálását. A megszakítási rendszer rendelkezhet egy vagy több megszakítási vonallal.

Egy megszakítási vonal esetén: az egyes eszközök (perifériák) megszakítási kérelmei egy közös gyűjtő VAGY kapura futnak be. Bármelyik kérelem estén a kapu kimenete jelzést ad a processzor egyetlen megszakítási vonala (INT) felé. Ez a jelzés egy kiszolgáló rutint indít el, amely sorra megvizsgálja az eszközök állapotjelzőit (megszakítási FLAG-bitek). Ilyen rendszer látható a 34. a) és b) ábrán.



34. ábra

A b) ábrán az ún. Daisy Chain látható. A perifériák egymás után vannak láncolva. A láncban elfoglalt helyük egyben a prioritási sorrendet is meghatározza. A legmagasabb prioritású (P1) periféria IEI – Interrupt Enable Input – lába logikai 1-re van kötve, így ennek mindig engedélyezett a megszakítási kérelme. Az IEO – Interrupt Enable Output – lábon kiadott logikai 0-val a mögötte lévő eszközöket bármelyik periféria letilthatja.

Több megszakítási vonal esetén: minden eszköz saját megszakítást kérő vezetékkel rendelkezi, így a kérelem helye egyértelműen meghatározható, és a hozzá tartozó kiszolgáló rutin elindítható.

Vektoros módszer: a megszakítást kérő eszköz valamilyen módon a kiszolgáló rutin kezdőcímét határozza meg a megszakítás-vezérlő és a processzor számára. Erre a következő eljárások ismertek:

- A megszakítást kérő eszköz az őt kiszolgáló rutint hívó utasítást adja át a processzornak (pl. CALL INT_RUT)
- A kérő eszköz annak a tárolóhelynek a címét adja át a processzornak, amelyben a kiszolgáló rutint hívó utasítás található
- A megszakítást kérő eszköz az őt kiszolgáló rutinnak a kezdőcímét adja át a processzornak, amely azt, mint a soron következő utasítás címét betölti az utasításszámlálóba
- A megszakítást kérő eszköz egy sorszámot ad át a processzornak, amely sorszám a megszakításokat kiszolgáló rutinok kezdőcímét tartalmazó táblában, a megszakítási vektortáblában kijelöli az adott eszközt kiszolgáló rutin kezdőcímét

A megszakítások prioritásának kezelésére, és a többszörös megszakításkezelésre kidolgozott eljárások:

- Forgó (Rotary): mindig a legutoljára kiszolgált eszköz lesz a legutolsó a prioritási sorrendben. Többszörös megszakításkezelésre nincs lehetőség, mivel a kiszolgáló program letiltja az újabb megszakítási kérelmek kiszolgálását
- Teljesen egymásba skatulyázott (Full Nested): többszintű rendszerekben a megszakítást kiszolgáló rutin is megszakítható bizonyos szabályok figyelembe vételével:
  - A kiszolgáló rutin a vele egyező vagy alacsonyabb prioritású kérelmeket letiltja
  - A kiszolgáló rutin a folyamat elején ideiglenesen eggyel alacsonyabb szintre sorolja magát, azaz a vele egyező, vagy nagyobb prioritásúak megszakíthatják a kérelmet
  - A kiszolgáló rutin ideiglenesen új prioritásokat rendel az egyes eszközökhöz

Az általános alapelvek tisztázása után nézzük meg, hogyan alakították ki a

megszakítási rendszert a 16F84-en! A mikrovezérlőnk egy megszakítási vonallal rendelkezik (30. a) ábra). Az ide érkező kérelmet a processzor minden utasításlehíváskor az órajel T_2 fázisában megvizsgálja. Többszörös megszakítás nincs, prioritást nem kezel. Megszakítás keletkezésekor a CALL 4 utasítás fut le. A 16F84 a következő megszakítási forrásokkal rendelkezik:

- Külső megszakítás az RB0/INT lábon
- TMR0 túlcsordulás
- Változás a PORTB felső 4 bitjén
- A belső EEPROM írása befejeződött

A megszakítást vezérlő regiszter (**INTCON**) tartalmazza az egyes megszakítások jelzőbitjeit. Ebben a regiszterben találhatók még az egyes megszakítások engedélyező (maszk) bitjei is, valamint a globális engedélyező bit. A belső EEPROM írásának befejeztét jelző **EEIF** bit az **EECON1** regiszterben található!

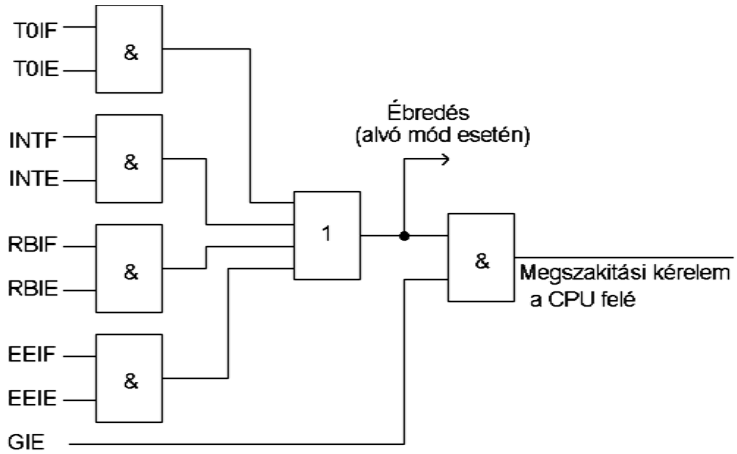
A globális megszakítást engedélyező bit a **GIE** (**INTCON<7>**) engedélyezi (**GIE=1**) az összes nem maszkolt megszakítást, illetve ha a **GIE**=0, akkor pedig mindentől függetlenül az összes megszakítást letiltja. Amikor egy megszakítási kérelem elfogadásra kerül, akkor a **GIE** bit 0 lesz, letiltva ezáltal az összes többi megszakítást. A **PC** tartalma (visszatérési cím) elmentődik a verembe, majd feltöltődik a **0004h** *megszakítási vektorral*. Mivel az összes megszakítás esetén a 0004h címre fut a program, a felhasználónak a jelzőbitek tesztelésével kell a megszakítás forrását megállapítania (polling):

```
ORG     4

BTFSC   INTCON,T0IF
GOTO    T0_INT_RUT
BTFSC   INTCON,RBIF
GOTO    RB_CHANGE_INT_RUT
.
.
```

A kiszolgáló rutinban a felhasználónak törölnie kell a megfelelő jelzőbitet, mert ellenkező esetben állandóan erre a rutinra futna a program. A megszakítási rutinból a **RETFIE** (nem **RETURN**) utasítással kell visszatérni, mert ez az az utasítás, ami újra engedélyezi a megszításokat (**GIE=1**). Ez utóbbi kettő

figyelmen kívül hagyása sok hiba forrása lehet, ezért erre mindenképpen hívjuk fel a tanulók figyelmét. A megszakítási logika a 35. ábrán látható:



35. ábra

Külső megszakítás az **RB0/int** lábon: az **RB0/INT** lábon érkező megszakítás élvezérelt, a megszakítás a jel felfutó élére történik, ha az **INTEDG** bit (**OPTION_REG<6>**) 1-be van állítva, illetve lefutó élre következik be, ha 0-ba van állítva. Amikor a beállításnak megfelelő változás történik az **RB0/INT** lábon, akkor az **INTF** bit 1-be billen (**INTCON<1>**). A megszakítást engedélyezni az **INTE** bit (**INTCON<4>**) 1-be billentésével lehet, tiltását pedig a bit 0-ba állításával lehet elérni. Az **INTF** jelzőbitet szoftverből, a megszakítást kiszolgáló rutinban kell törölni, mielőtt újra engedélyeződik a megszakítás (**RETFIE**). Az RB0/INT lábon érkező megszakítás felébresztheti a processzort alvó módból (**SLEEP**), ha előtte az **INTE** bit 1-be volt állítva.

TMR0 megszakítás: a TMR0 megszakítást a **TMR0** regiszter túlcsordulása (FFh→00h). Túlcsorduláskor a **T0IF** (**INTCON<2>**) 1-be vált. A megszakítást engedélyezni/tiltani a **T0IE** (**INTCON<5>**) bit 1-be állításával ill. törlésével lehet.

PORTB megszakítás: amikor a **PORTB<7:4>** bitek valamelyikén változás történik az **RBIF** bit (**INTCON<0>**) 1-be vált. A megszakítást engedélyezni/tiltani az **RBIE** (**INTCON<3>**) bit 1-be állításával ill. törlésével lehet.

EEPROM megszakítás: amikor a belső EEPROM adatmemória írási ciklusa befejeződik az **EEIF** bit (**EECON1<4>**) 1-be vált. A megszakítást engedélyezni/tiltani a **EEIE** (**INTCON<6>**) bit 1-be állításával ill. törlésével lehet. Mivel a megszakítás aszinkron módon bármikor bekövetkezhet a főprogram

futása során, így a megszakítási rutin a főprogramban is használatos regisztereket felülírja. Kézenfekvő megoldásnak kínálkozik, hogy más-más regisztereket használunk, azonban a munkaregisztert (**W**) és a státuszregisztert (**STATUS**) nem kerülhetjük ki. A helyzetet tovább bonyolítja, hogy a státuszregiszter mentésénél csak olyan utasítást használhatunk, amely nem állítja a flag-biteket! A 2. táblázatot tanulmányozva láthatjuk, hogy két olyan mozgató utasítás van, amely nem állít státuszbitet (**SWAPF, MOVWF**). A következő mintapélda elmenti és visszaállítja a **STATUS** és a **W** regiszterek tartalmát. A felhasználónak definiálni kell a **W_TEMP** és a **STATUS_TEMP** regisztereket, ahol átmenetileg tárolódnak az adatok.

A példa a következő lépéseket tartalmazza:

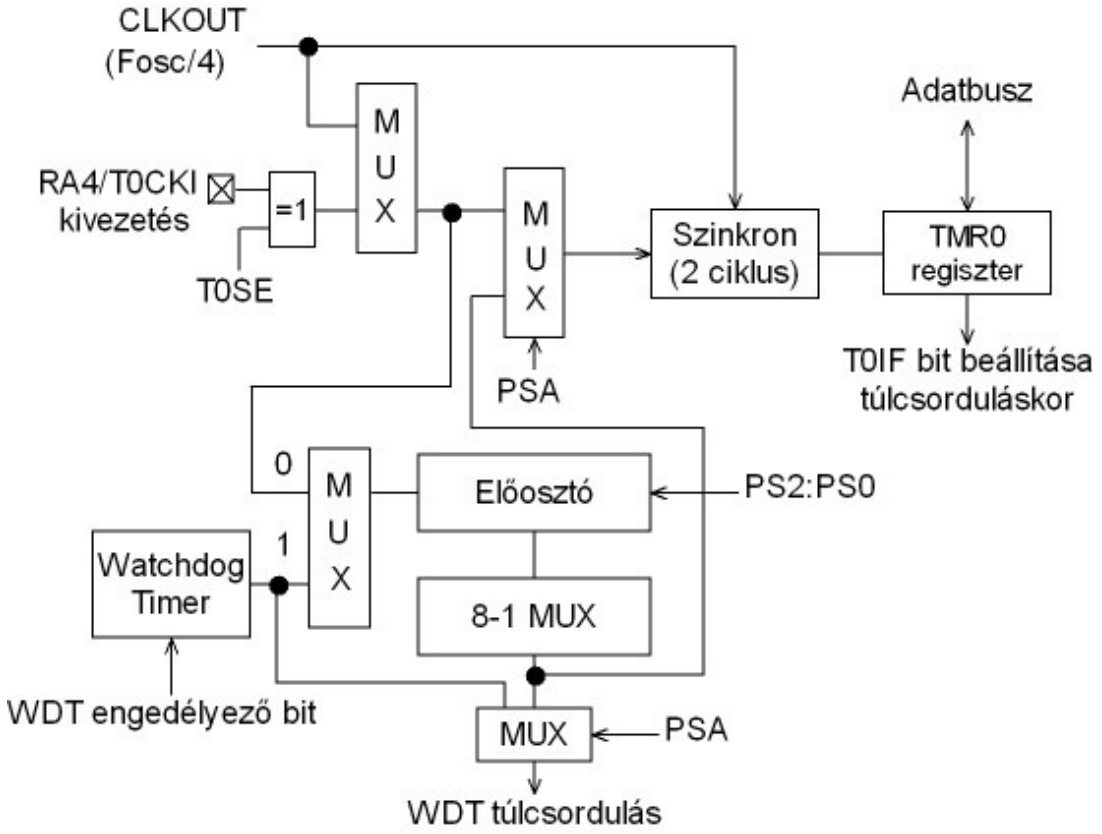
a) W regiszter mentése
b) A STATUS regiszter mentése a STATUS_TEMP regiszterbe
c) A megszakítási rutin végrehajtása
d) A STATUS (a bankválasztó bitek is) regiszter visszaállítása
e) A W regiszter visszaállítása

```
PUSH    MOVWF   W_TEMP          ;W mentése
        SWAPF   STATUS,W
        MOVWF   STATUS_TEMP     ;STATUS mentése
ISR_RUTIN:
                :
                :
POP     SWAPF   STATUS_TEMP,W
        MOVWF   STATUS          ;STATUS visszaállítása
        SWAPF   W_TEMP,F
        SWAPF   W_TEMP,W        ;W visszaállítása
        RETFIE
```

A regiszterek mentésének elmaradása szintén számos rejtélyes hiba forrása lehet! A tanulók figyelmét hívjuk fel erre a fontos tényre!

A kijelzőnk működtetésére a TMR0 megszakítást fogjuk felhasználni. A **TIMER0** működhet számlálóként vagy időzítőként. Időzítőként akkor működik, ha **T0CS** bitet (**OPTION_REG<5>**) 0-ba állítjuk, számlálóként pedig akkor, ha 1-be billentjük. Időzítő módban minden utasításciklus eggyel növeli a **TMR0** regiszter értékét (feltéve, hogy nincs előosztás). A **TMR0** regiszter tartalmát a felhasználó is módosíthatja. Számláló módban a **TMR0** regiszter értéke

növekszik minden felfutó vagy lefutó él hatására, amely az **RA4/T0CKI** lábon történik. A felfutó vagy lefutó él kiválasztása a **T0SE** bittel (**OPTION_REG<4>**) történik. Nullába állítva ezt a bitet felfutó élre történik, 1-be állítva pedig lefutó élre történik a növelés. A modulhoz tartozik egy ún. előosztó is. Ez a 8 bites számláló regiszter lehet a Timer0 modul előosztója, vagy a Watchdog Timer utóosztója, mint ahogy a 36. ábrán látható. Mivel csak egy regiszterünk van, amely megosztott a Timer0 modul és a Watchdog Timer között, ezért, ha az előosztót a Timer0 modulhoz rendeljük, akkor a Watchdog Timer-nek nincs osztója, s ez ugyanígy fordítva is igaz. Az előosztó nem írható és nem is olvasható. A **PSA** bit (**OPTION_REG<3>**) határozza meg, hogy az előosztó

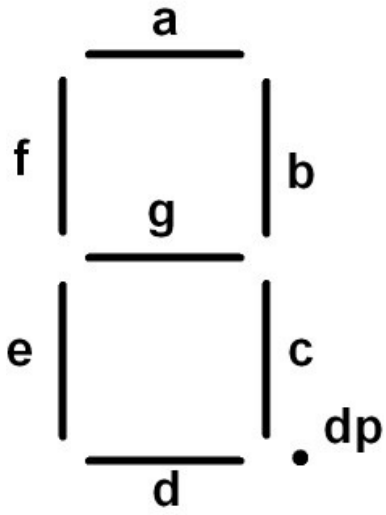


36. ábra

hová kapcsolódik. Ha a **PSA** bit értéke 0, akkor az osztó a Timer0 modulhoz, ha 1, akkor a Watchdog Timer-hez kapcsolódik. Az osztás mértékét a **PS2:PS0** bitek határozzák meg. Ha az osztó a Timer0 modulhoz kapcsolódik, akkor az értékek 1:2, 1:4….1:256 között alakulnak, ha a Watchdog Timer-hez kapcsolódik, akkor pedig 1:1, 1:2….1:128 közötti értékeket vehet fel.

A Timer0 tehát beállítástól függően adott időközönként megszakítást okoz. A

megszakítási rutin feladata az lesz, hogy a két kijelzőre felváltva kiírja az adatokat. Megfelelő időköz megválasztásával a szemünket „becsapjuk”, s úgy fog tűnni, hogy egyszerre mindkét kijelző világít. A hétszegmenses kijelzők közös anódjait a T_1 és T_2 tranzisztorok kapcsolják. Ezek PNP tranzisztorok, ezért a bekapcsolásukhoz az **RA0** és **RA1** lábra 0-t kell adni! Ezek alapján a megszakítási rutinunk a következőképpen nézhet ki:



37. ábra

```
T0_INT
        BCF     INTCON,T0IF     ;Megszakítási
flag törlése (kötelező)
        BTFSS   MELYIK          ;Melyik kijelzőre írunk
        GOTO    KIJEL2          ;2-es kijelző
        BSF     ANOD2           ;2-es kijelzö KI
        BCF     ANOD1           ;1-es BE
        MOVF    ADAT1,W
        MOVWF   PORTB           ;Kiírás a kijelzőre
        BCF     MELYIK          ;Váltás a másik kijelzőre
        GOTO    POP             ;Kilépés
KIJEL2
        BSF     ANOD1           ;1-es kijelzö KI
        BCF     ANOD2           ;2-es BE
        MOVF    ADAT1,W
        MOVWF   PORTB           ;Kiírás a kijelzőre
        BSF     MELYIK          ;Váltás a másik kijelzőre
        GOTO    POP             ;Kilépés
```

A megszakítási rutinhoz definiálnunk kell még néhány szimbólumot és regisztert:

```
#DEFINE  ANOD1   PORTA,0
#DEFINE  ANOD2   PORTA,1
#DEFINE  MELYIK  FLAG,0
```

illetve a konstansblokkban:

```
CBLOCK   0X0C
```

```
.
.
ADAT1
ADAT2
FLAG
.
.
ENDC
```

A kijelzők kiválasztására szolgáló bit létrehozásához (**MELYIK)** definiáltuk a **FLAG** regisztert. Ennek a 0-s bitjét használjuk most fel. Ezt a regisztert a későbbiekben is felhasználhatjuk, hiszen maradt még 7 felhasználható bitje.

A kijelzendő értékekhez célszerű még készítenünk egy adattáblát! Ez a tábla fogja tartalmazni, hogy az adott számjegyhez a hétszegmenses kijelzőn melyik szegmenseket kell bekapcsolni. A kijelző felépítése és szegmenseinek elnevezése a 37. ábrán látható. A szegmensek sorban „dp”-től az „a”-ig az **RB7-RB0** lábakra vannak kötve (a kiosztás a v2-es próbapanelra vonatkozik). Mivel a kijelző közös anódos, ezért a megfelelő szegmensre alacsony szintet kell kapcsolni ahhoz, hogy világítson. Például az 1-es szám megjelenítéséhez a „b” és a „c” szegmenseket kell bekapcsolni, azaz a **PORTB** 6-os és 5-ös lábára magas, a többire alacsony szintet kell kapcsolni! Az egyes számértékekhez tartozó kódok a 4. táblázatban láthatók. Ezen kódok alapján készíthetjük el az adattáblát, amely nagymértékben hasonlít a már ismert ugrótáblához. Itt is azt használjuk ki, hogy a programszámláló aktuális tartalmához hozzáadjuk a munkaregiszter (**W**) tartalmát. Így a W értékétől függően a megfelelő sorra ugrik a program. Már csak azt kell megoldani, hogy a program az ezekbe a sorokba írt kódokkal térjen vissza. Ezt legegyszerűbben a **RETLW** (Return with Literal in W) utasítással tehetjük meg. A hasonlóság miatt az adattáblára is ugyanazok a szabályok vonatkoznak, mint az ugrótáblára:

Karakter	Kód	
	Bináris	Hexa
0	11000000	C0
1	11111001	F9
2	10100100	A4
3	10110000	B0
4	10011001	99
5	10010010	92
6	10000010	82
7	11111000	F8
8	10000000	80
9	10010000	90
pont	01111111	7F

4. táblázat

- A 8 bites **ALU** miatt csak a **PCL**-t tudjuk írni, a **PCH**-t a hívás előtt be kell állítani
- Mivel a **PCH** nem állítódik, ezért a tábla nem lehet laphatáron!
- A hívás előtt a W-ben nem lehet nagyobb szám, mint ahány bejegyzés van a táblában!

Ezek alapján elkészíthetjük az adattáblánkat:

```
        ORG     300
SZEGMENS_TABLA
        ADDWF   PCL,F       ;PC feltöltése
        RETLW   B'00011111' ;visszatérés a 0 kódjával
        RETLW   B'10010000' ;visszatérés az 1 kódjával
        .
        .
        RETLW   B'11111110' ;visszatérés a pontkódjával
```

Illetve választhatunk egy másik megoldást is, amihez a **DT** (Define Table) direktívát használjuk. Itt az adatokat vesszővel egymástól elválasztva folyamatosan írhatjuk:

```
        ORG     300
SZEGMENS_TABLA
        ADDWF   PCL,F       ;PC feltöltése
        DT      0C0,OF9,0A4,0B0,99,92,82,0F8,80,90
```

Itt arra kell ügyelni, hogy szám csak számmal kezdődhet, ezért FE helyett 0FE-t kell írni. Ezt a megoldást inkább akkor érdemes használni, ha pl. szöveget akarunk kiírni. Teljes megoldást itt sem adok, mindenkinek a fantáziájára bízom a megoldást. Hagyjuk a tanulókat önállóan dolgozni. Az első feladatként célszerű azt elkészíttetni a gyerekekkel, hogy az első kijelzőre az 1-es számot, a második kijelzőre pedig a 2-es számot írja ki a program. Érdemes kétsugaras oszcilloszkóppal megvizsgálni az anódokat kapcsoló jeleket. Szintén tanulságos kísérlet lehet a diákoknak az, ha lelassítjuk a multiplexet. Ilyenkor már szabad szemmel is láthatóvá válik, hogyan kapcsolja a program a kijelzőket.

Ezek után már nagyon sokféle feladatot adhatunk a diákoknak, amit a hétszegmenses kijelzővel oldhatnak meg:

- Számlálás 0-99-ig
- Számlálás egyik kijelzőn előre, a másikon hátra
- Kapcsolók értékének kiírása

- Időzítő
- Dobókocka
- Kijelzés fényerejének változtatása, stb.

3.7 Áttérés a 18-as mikrovezérlőkre

A 18xxx sorozatú mikrovezérlők alapjaikban hasonlítanak a már megismert 16F84-es mikrovezérlőre, de tartalmaznak nagyon sok újdonságot. Ezeket elsősorban a felhasználók tapasztalataira építve dolgozta ki a Microchip.

3.7.1 Memóriaszervezés

Sok gondot okozott a felhasználók körében a bankváltásból, illetve a programmemória lapozásából adódó hibák. Ezek helytelen használata rengeteg „rejtélyes” hibához vezetett. Ennek elkerülése érdekében a gyártó átdolgozta a mikrovezérlő memóriaszervezését.

A programmemória szélességét megváltoztatták 14 bitről 16 bitre, így ez immár közvetlenül kiolvasható vált, és 2 bájtot lehetett elhelyezni egy címen. A programszámláló hosszát 21 bitre növelték, amivel 2 Mszó címezhető meg:

20 16	15 8	7 0
PCLATU	**PCLATH**	**PCL**

Emiatt szakítani kellett az egy utasítás egy szó elvével, ugyanis 21 bit nem fér el 1 16 bites szóban. Cserébe viszont nem kell lapozni, mert a CALL és a GOTO is átfogja a teljes memóriát.

A programmemória olvasására új táblakezelést dolgoztak ki. A címzést a TBLPTR táblamutató regiszter végzi, amellyel a teljes memória átfogható. A 8 bites architektúra miatt a TBLPTR is 3 részre tagolódik:

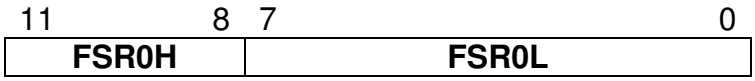
20 16	15 8	7 0
TBLPTRU	**TBLPTRH**	**TBLPTRL**

A kiolvasás a TBLRD* utasítással történik, és a kiolvasott adat a TABLAT regiszterbe kerül. Az alábbiakban egy példát láthatunk a táblakezelésre hétszegmenses kijelzőhöz:

```
;TÁBLAKEZELÉS
MOVLW   UPPER   KAR1_TABLA
MOVWF   TBLPTRU         ; ITT A "HIGH" FELETT "UPPER" IS VAN
MOVLW   HIGH  KAR1_TABLA
MOVWF   TBLPTRH
```

```
MOVLW   .5
MOVWF   TBLPTRL         ;TÁBLA 5. SORÁNAK CÍMZÉSE
TBLRD*                  ;TÁBLA OLVASÁS
MOVF    TABLAT,W        ;AZ EREDMÉNY A "TABLAT"-BAN,
                        ;AMIT MOST A W-BE MOZGATUNK
;használhatunk még:
;       TBLRD*+                 ;növeli a mutatót kiolvasás után
;       TBLRD*-                 ;CSÖKKENTI
.
.
;itt 16 bites a programmemória, így 2 bájt fér el
        ORG     800
KAR1_TABLA
        DB      B'00111111',B'00000110'   ;0-1
        DB      B'01011011',B'01001111'   ;2-3
        DB      B'01100110',B'01101101'   ;4-5
        DB      B'01111101',B'00000111'   ;6-7
        DB      B'01111111',B'00000010'   ;8-9
        DB      B'00000110',B'01101111'   ;10-11
```

Az adatmemória is jelentős változáson ment keresztül. Az adatok tárolására egy 4096 bájtos fájlregisztertömb szolgál, amely indirekt módon a 12 bites FSR regiszterrel címezhető meg. Így az FSR regiszter átfogja a teljes memóriát:



A tapasztalatok azt mutatták, hogy az FSR regiszterek segítségével leggyakrabban a sorban egymás utáni adatmemória-regisztereket címezzük – pl. soros vonalon bejövő adatok tárolása – ezért az FSR regiszterek számát kezelését kibővítették. A 18-as családban 3db FSR: az FSR0, FSR1 és FSR2 található. A hozzájuk tartozó árnyékregiszterek pedig: INDF0, INDF1, INDF2. A ciklikus kezelhetőség érdekében – pl. törlünk egy adott memóriaterületet – bevezettek olyan regisztereket, amelyekhez való hozzáférés automatikusan növeli, vagy csökkenti az adott FSR tartalmát:

POSTINCn	[FSRn] = [FSRn+1]	végrehajtás *után* az FSR 1-gyel nő
POSTDECn	[FSRn] = [FSRn-1]	végrehajtás *után* az FSR 1-gyel csökken

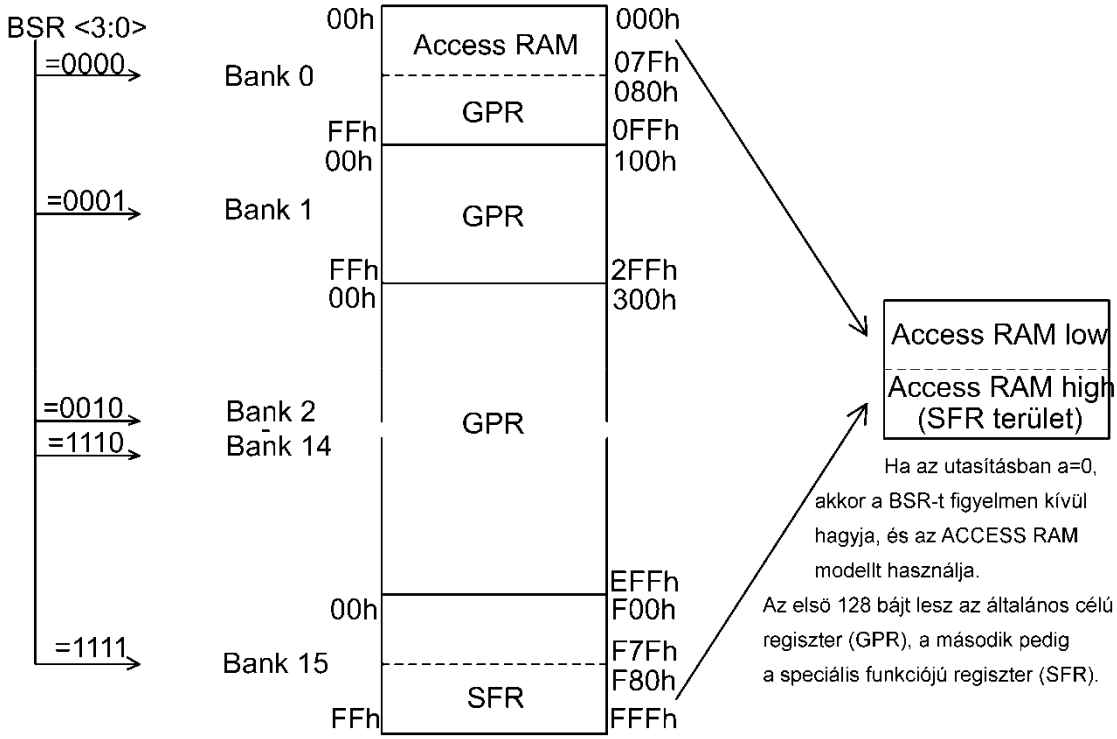
PREINCn [FSRn] = [FSRn+1] végrehajtás *előtt* az FSR 1-gyel nő

PLUSWn [FSRn] = [FSRn+[WREG]] végrehajtás *előtt* az FSR tartalmához a WREG tartalmát adjuk hozzá

Az FSR regiszterek kezdőértékének feltöltésére az **LFSR** utasítás szolgál, így meg tudunk adni közvetlenül 12 bites címet. A következőkben példát láthatunk az FSR használatára. A rutin a programmemória egy részét törli:

```
;törli az adatmemória 0-3 bankját
TOROL
        CLRF    POSTINC0
        BTFSS   FSR0H,2
        GOTO    TOROL           ;BANK0-3 TOROL
        RETURN
```

Mivel az architektúra, mint említettem 8 bites, felmerül a kérdés, hogy akkor az utasításokban – pl. a PORTA regiszterbe akarunk adatot mozgatni – hogyan képezzük le a 12 biten címezhető regisztereket. Erre két megoldás létezik a 18-as sorozatban. Az egyik megoldásban 4096 bájtot 16db 256 bájtos bankra



38. ábra

osztjuk (38. ábra). A bankok kiválasztása a 4 bites **BSR** (Bank Select Register) segítségével történik. Amennyiben ezt a modellt használjuk, különös figyelmet fordítsuk a BSR megfelelő használatára, ugyanis a bankváltások elmaradása, ill. helytelen használata számos hiba forrása lehet. Amennyiben a felhasználó nem szeretne ilyen sok regisztert utasításból elérni, akkor használhat egy egyszerűsített modellt, ahol nem kell bankot váltania! Ezt a modellt **Access RAM**-nak nevezi a gyártó. Ilyenkor az adatmemória első 128 bájtja (000-07F) lesz az általános célú regiszter – GPR – illetve a 4 kilobájtos memória utolsó 128 bájtja (F80-FFF) lesz az SFR (speciális funkciójú regiszter).

Most már csak egy kérdést kell tisztázni: honnan tudja a mikrovezérlő címaritmetikája, hogy melyik modellt használjuk? A memóriamodell megadásához 16 bites utasítás kódszavából használtak fel egy bitet. Ennek a neve: „a” (acces). Amennyiben a=0, akkor az **Access RAM**-ot használja a mikrovezérlő, ha a=1, akkor pedig a BSR regiszter választja ki azt a bankot, amit használni fogunk. Az utasításban ezután nem csak a célt, hanem a memóriamodellt is meg kell adni (pl ADDWF REG,W,0). Szerencsére az MPLAB alapból azt feltételezi, hogy az **Access RAM**-ot használjuk, így ezt külön jelölni nem kell, csak akkor kell megadni az access bitet, ha a másik memóriamodellt használjuk.

3.7.2 Utasításkészlet

Egyrészt a megváltozott architektúra, másrészt az egyszerűbb kezelhetőség – szegényes utasításkészlet – miatt az utasításkészletet kibővítették. Ez eredményezett ötletes utasításokat, de véleményem szerint felesleges utasításokat is. A kevesebb sokszor több szokták mondani, ez itt is igaz, nem biztos, hogy a 16-os sorozat úgymond „szegényes” utasításkészlete ne lett volna hatékony!

Mint említettem a PIC 18xxx mikrovezérlők kiterjesztett utasításkészlettel rendelkeznek. A legtöbb utasítás egyszavas (16 bit), de létezik 3 kétszavas utasítás is. Mindegyik egyszavas utasítás egy műveleti kódból (MK) – amely megadja az utasítás típusát – és egy vagy két operandusból – amelyre a művelet vonatkozik – áll.

Az utasításkészlet 4 alapkategóriára osztható:

- **Bájt orientált** művelet
- **Bit orientált** művelet
- **Konstans** művelet
- **Vezérlő** utasítás

A **bájt orientált** utasítások operandus mezője 3 részből áll:

1. Fájlregiszter (jele: **f**)
2. Az eredmény helye (jele: **d**)
3. „Acces" memória (jele: **a**)

A fájlregisztert leíró „f" azt a regisztert jelenti, amire az utasítás irányul.

A cél leíró (d) a művelet eredményének helyét adja meg. Amennyiben a d=0 (vagy d=W) az eredmény a WREG-ben keletkezik, amikor pedig a d=1 (vagy d=F) akkor abban a fájlregiszterben, amire az utasítás vonatkozik.

A **bit orientált** utasítások operandus mezője is 3 részből áll:

1. Fájlregiszter (jele: **f**)
2. A fájlregiszter egy bitje (jele: **b**)
3. „Access" memória (jele: **a**)

A bitet megadó „b" azonosító a fájlregiszteren belül annak a bitnek a sorszáma, amire a művelet vonatkozik.

A **konstans** műveletek operandus mezője a következő részekből áll:

- A konstans értéke, amit a fájlregiszterbe töltünk (jele: **k**)
- A felhasznált FSR regiszter, amibe a konstanst töltjük (jele: **f**)
- Nincs operandus (jele: **-**)

A **vezérlő** utasítások operandus mezője pedig a következő részekből áll:

- Programmemória cím (jele: **n**)
- A CALL vagy a RETURN utasítás módja (jele: **s**)
- Tábla írás és olvasás módja (jele: **m**)
- Nincs operandus (jele: **-**)

A kétszavas utasítások 32 bitből állnak. A második szó felső 4 bitje 1-es. Amennyiben az első szó is utasításként értelmezett, akkor NOP hajtódik végre. Az egyszavas utasítások 1 gépi ciklus alatt hajtódnak végre, kivéve azokat az elágaztató utasításokat, amelyeknél a PC tartalma módosul. A kétszavas utasítás két gépi ciklus alatt hajtódik végre. Minden utasítás 4 oszcillátor periódust igényel. Ez azt jelenti, ha 4MHz a oszcillátor frekvenciája, akkor 1

utasítás 1µs alatt hajtódik végre. Feltételes ugró utasítás esetén amikor a PC tartalma módosul az utasítás végrehajtásához 2µs szükséges. Kétszavas elágaztató utasításoknál, ha a feltétel teljesül a végrehajtási idő 3µs lesz. A példákban használt formátum „nnnh” hexadecimális számot jelent, ahol a „h” hexát jelent. A következő táblázatokból megismerhetjük a 18-as mirovezérlők utasításkészletét.

Mező	Leírás
a	RAM hozzáférési bit a=0: a RAM terület az ún. „Access” RAM-ban van (a BSR-t nem kell figyelembe venni) a=1: a RAM bankot a BSR jelöli ki
bbb	Bit cím a 8 bites fájlregiszterben
d	Célkiválasztó bit: d=0(W): az eredmény a WREG-be kerül d=1(F): az eredmény a fájlregiszterbe (f) kerül
dest	Cél, akár a WREG, akár egy fájlregiszter
f	8 bites fájlregiszter neve (pl PORTB) v. címe(0xFF)
f_s	12 fájlregiszter címe (0x000-0xFFF). A forráscím tkp.
f_d	12 fájlregiszter címe (0x000-0xFFF). A célcím tkp.
k	Konstans mező, konstans adat, vagy címke (8, 16, ill. 20 bites lehet)
label	Címke neve
mm * *+ *- +*	A TBLPTR regiszter kezelésének módja tábla íráskor és olvasáskor: Nem változik a TBLPTR értéke (se íráskor, se olvasáskor) Az olvasás vagy írás után 1-gyel növekszik a TBLPTR Az olvasás vagy írás után 1-gyel csökken a TBLPTR Az olvasás vagy írás előtt 1-gyel növekszik a TBLPTR
n	Relatív cím (2-es komplemensben adott szám) a feltételes elágaztató utasításoknál, vagy közvetlen cím szubrutin hívásnál, visszatéréskor, ugráskor.
PRODH	A szorzás eredményének felső bájtja
PRODL	A szorzás eredményének alsó bájtja
s	Fast call/return módot kiválasztó bit s=0: nincs mentés az árnyékregiszterekbe s=4: mentés/töltés az árnyékregiszterekbe (fast módus)
u	Nem használt, vagy nem változik
WREG	Munkaregiszter (akkumulátor)
x	Figyelmen kívül hagyható (0 v. 1) A fordító nullát generál. Ez a többi Microchip szoftverrel való kompatíbilitáshoz szükséges.
TBLPTR	21 bites táblamutató (egy programmemória beli címre mutat)
TABLAT	A táblából kiolvasott értéket tárolja
TOS	TOP OF STACK, a verem teteje
PC	Programszámláló
PCL	Programszámláló alsó bájtja
PCH	Programszámláló felső bájtja
PCLATH	Programszámláló felső bájtját tároló regiszter
PCLATU	Programszámláló legfelső bájtját tároló regiszter
GIE	Általános megszakítás engedélyezés
WDT	Watchdog Timer
$\overline{\text{TO}}$	Time-out bit
$\overline{\text{PD}}$	Power-down bit
C, DC, Z, OV, N	Az ALU státuszbitjei: átvitelbit, fél átvitel, zéróbit, túlcsordulásbit, előjelbit
[ ]	Feltételes
( )	Tartalom
→	Hozzárendelés
< >	Bitterület a regiszterben
€	Halmaz eleme
Dőlt betű	A felhasználó által definiált

Mnemonik Operandus	Leírás	Ciklus	16 bites kód MSB LSB	Állított jelzőbitek	Megjegyzés
Bájt orientált fájlregiszter műveletek					
ADDWF f,d,a	W és f összeadása	1	0010 01da ffff ffff	C,DC,Z,OV,N	1,2
ADDWFC f,d,a	W, f és az átvitelbit összeadása	1	0010 00da ffff ffff	C,DC,Z,OV,N	1,2
ANDWF f,d,a	W és f ÉS kapcsolata	1	0001 01da ffff ffff	Z,N	1,2
CLRF f,a	f törlése	1	0110 101a ffff ffff	Z	2
COMF f,d,a	f komplementálása	1	0001 11da ffff ffff	Z,N	1,2
CPFSEQ f,a	WREG és f összehasonlítása, átlép ha =	1 (2 v.3)	0110 001a ffff ffff	nincs	4
CPFSGT f,a	WREG és f összehasonlítása, átlép ha f>	1 (2 v.3)	0110 010a ffff ffff	nincs	4
CPFSLT f,a	WREG és f összehasonlítása, átlép ha f<	1 (2 v.3)	0110 000a ffff ffff	nincs	4
DECF f,d,a	f csökkentése	1	0000 01da ffff ffff	C,DC,Z,OV,N	1,2,3,4
DECFSZ f,d,a	f csökkentése, átlép ha 0	1 (2 v.3)	0010 11da ffff ffff	nincs	1,2,3,4
DCFSNZ f,d,a	f csökkentése, átlép ha nem 0	1 (2 v.3)	0100 11da ffff ffff	nincs	1,2
INCF f,d,a	f növelése	1	0010 10da ffff ffff	C,DC,Z,OV,N	1,2,3,4
INCFSZ f,d,a	f növelése, átlép ha 0	1 (2 v.3)	0011 11da ffff ffff	nincs	4
INFSNZ f,d,a	f növelése, átlép ha nem 0	1 (2 v.3)	0100 10da ffff ffff	nincs	1,2
IORWF f,d,a	WREG és f VAGY kapcsolata	1	0001 00da ffff ffff	Z,N	1,2
MOVF f,d,a	f mozgatása	1	0101 00da ffff ffff	Z,N	1
MOVFF f_s, f_d	f_s (forrás)mozgatása 1. szó f_d -be (cél) 2. szó	2	1100 ffff ffff ffff 1111 ffff ffff ffff	nincs	
MOVWF f,a	WREG mozgatása f-be	1	0110 111a ffff ffff	nincs	
MULWF f,a	WREG és f összeszorzása	1	0000 001affff ffff	nincs	
NEGF f,a	f kettes komplemensének képzése	1	0110 110a ffff ffff	C,DC,Z,OV,N	1,2
RLCF f,d,a	f forgatása balra átvitelbiten keresztül	1	0011 01da ffff ffff	C,Z,N	
RLNCF f,d,a	f forgatása balra átvitelbit kihagyásával	1	0100 01daffff ffff	Z,N	1,2
RRCF f,d,a	f forgatása jobbra átvitelbiten keresztül	1	0011 00da ffff ffff	C,Z,N	
RRNCF f,d,a	f forgatása jobbra átvitelbit kihagyásával	1	0100 00da ffff ffff	Z,N	
SET f	f 1-be állítása	1	0110 100a ffff ffff	nincs	
SUBFWB f,d,a	f kivonása a WREG-ből áthozattal	1	0101 01da ffff ffff	C,DC,Z,OV,N	1,2
SUBWF f,d,a	WREG kivonása f-ből	1	0101 11da ffff ffff	C,DC,Z,OV,N	
SUBWFB f,d,a	WREG kivonása f-ből áthozattal	1	0101 10da ffff ffff	C,DC,Z,OV,N	1,2
SWAPF f,d,a	f alsó és felső 4 bitjének felcserélése	1	0011 10da ffff ffff	nincs	4
TSTFSZ f,a	f tesztelése és átlépés, ha 0	1 (2 v.3)	0110 011a ffff ffff	nincs	1,2
XORWF f,d,a	WREG és az f kizáró-VAGY kapcsolata	1	0001 10da ffff ffff	Z,N	
Bit orientált fájlregiszter műveletek					
BCF f,b,a	f adott bitjének törlése	1	1001 bbba ffff ffff	nincs	1,2
BSF f,b,a	f adott bitjének 1-be állítása	1	1000 bbba ffff ffff	nincs	1,2
BTFSC f,b,a	f adott bitjének tesztelése és átlép, ha 0	1 (2 v.3)	1011 bbba ffff ffff	nincs	3,4
BTFSS f,b,a	f adott bitjének tesztelése és átlép, ha 1	1 (2 v.3)	1010 bbba ffff ffff	nincs	3,4
BTG f,d,a	f adott bitjének invertálása	1	0111 bbba ffff ffff	nincs	1,2

Megjegyzések:

1. Amikor a PORT értéke változik, és a művelet eredménye önmagába íródik (pl. MOVF PORTB,0,1 vagy MOVF PORTB,F,A) az az érték kerül beírásra, ami jelenleg a lábon mérhető. Pl. ha a tárolt adat a regiszterben 1, és a láb bemenetnek van konfigurálva, és ezt a lábat egy külső eszköz 0-ba húzza, akkor a regiszterbe 0 kerül visszaírásra.
2. Abban az esetben, ha a művelet a TMR0 regiszterre vonatkozik (d=1, vagy d=f), és az előosztó a TMR0-hoz van rendelve, akkor az előosztó törlődik.
3. Amikor a programszámláló (PC) változik az utasítás két ciklus hosszú lesz. A második ciklusban a NOP utasítás kerül végrehajtásra.
4. Némely utasítás 2-szó hosszúságú. A második szó ezekben az utasításokban NOP-ként hajtódik végre, kivéve ha az utasítás első szavának folytatása ez a 16 bit.

Mnemonik Operandus	Leírás	Ciklus	16 bites kód MSB LSB	Állított jelzőbitek	Megjegyzés
Vezérlő utasítások					
BC n	ugrás, ha az átvitelbit 1	1 (2)	1110 0010 nnnn nnnn	nincs	
BN n	ugrás, ha az előjelbit 1	1 (2)	1110 0110 nnnn nnnn	nincs	
BNC	ugrás, ha az átvitelbit 0	1 (2)	1110 0011 nnnn nnnn	nincs	
BNN n	ugrás, ha az előjelbit 0	1 (2)	1110 0111 nnnn nnnn	nincs	
BNOV n	ugrás, ha túlcsordulásbit 0	1 (2)	1110 0101nnnn nnnn	nincs	
BNZ n	ugrás, ha a zéróbit 0	2	1110 0001 nnnn nnnn	nincs	
BOV n	ugrás, ha túlcsordulásbit 1	1 (2)	1110 0100 nnnn nnnn	nincs	
BRA n	feltétel nélküli ugrás	1 (2)	1101 0nnn nnnn nnnn	nincs	
BZ n	ugrás, ha a zéróbit 1	1 (2)	1110 0000 nnnn nnnn	nincs	
CALL n,s	Szubrutin hívás 1. szó 2. szó	2	1110 110s kkkk kkkk 1111 kkkk kkkk kkkk	nincs	
CLRWDT -	Watchdog Timer törlése	1	0000 0000 0000 0100	$\overline{TO}$, $\overline{PD}$	
DAW -	WREG decimális korrekciója	1	0000 0000 0000 0111	C	
GOTO n	feltétel nélküli ugrás 1. szó 2. szó	2	1110 1111kkkk kkkk 1111 kkkk kkkk kkkk	nincs	4
NOP -	nincs kijelölt műveletvégzés	1	0000 0000 0000 0000	nincs	
NOP -	nincs kijelölt műveletvégzés	1	1111 xxxx xxxx xxxx	nincs	
POP -	kivétel a veremből	1	0000 0000 0000 0110	nincs	
PUSH -	beírás a verembe	1	0000 0000 0000 0101	nincs	
RCALL n	relatív szubrutin hívása	2	1101 1nnn nnnn nnnn	nincs	
RESET	Szoftveres reszet	1	0000 0000 1111 1111	mindegyik	
RETFIE s	visszatérés megszakítás engedélyezéssel	2	0000 0000 0001 000s	GIE/GIEH PEIE/GIEL	
RETLW k	visszatérés a WREG-ben egy konstanssal	2	0000 1100 kkkk kkkk	nincs	
RETURN s	visszatérés a szubrutinból	2	0000 0000 0001 001s	nincs	
SLEEP -	Szundi üzemmód	1	0000 0000 000 0011	$\overline{TO}$, $\overline{PD}$	

Megjegyzések:

1. Amikor a PORT értéke változik, és a művelet eredménye önmagába íródik (pl. MOVF PORTB,0,1 vagy MOVF PORTB,F,A) az az érték kerül beírásra, ami jelenleg a lábon mérhető. Pl. ha a tárolt adat a regiszterben 1, és a láb bemenetnek van konfigurálva, és ezt a lábat egy külső eszköz 0-ba húzza, akkor a regiszterbe 0 kerül visszaírásra.
2. Abban az esetben, ha a művelet a TMR0 regiszterre vonatkozik (d=1, vagy d=f), és az előosztó a TMR0-hoz van rendelve, akkor az előosztó törlődik.
3. Amikor a programszámláló (PC) változik az utasítás két ciklus hosszú lesz. A második ciklusban a NOP utasítás kerül végrehajtásra.
4. Némely utasítás 2-szó hosszúságú. A második szó ezekben az utasításokban NOP-ként hajtódik végre, kivéve ha az utasítás első szavának folytatása ez a 16 bit.

Mnemonik Operandus	Leírás	Ciklus	16 bites kód MSB LSB	Állított jelzőbitek	Megjegyzés
Konstans műveletek					
ADDLW	konstans hozzáadása a WREG-hez	1	0000 0000 kkkk kkkk	C,DC,Z,OV,N	
ANDLW	konstans illetve a WREG ÉS kapcsolata	1	0000 1011 kkkk kkkk	Z,N	
IORLW	konstans és a WREG VAGY kapcsolata	1	0000 1001 kkkk kkkk	Z,N	
LFSR	FSR feltöltése 2. szó egy konstanssal 1. szó	2	1110 1110 00ff kkkk 1111 0000 kkkk kkkk	nincs	
MOVLB	BSR feltöltése egy konstanssal <3:0>	1	0000 0001 0000 kkkk	nincs	
MOVLW	konstans betöltése a WREG-be	1	0000 1110 kkkk kkkk	nincs	
MULLW	konstans és a WREG összeszorzása	1	0000 1101 kkkk kkkk	nincs	
RETLW k	visszatérés a WREG-ben egy konstanssal	2	0000 1100 kkkk kkkk	nincs	
SUBLW	WREG kivonása a konstansból	1	0000 1000 kkkk kkkk	C,DC,Z,OV,N	
XORLW	WREG és a konstans kizáró-VAGY kapcsolata	1	0000 1010 kkkk kkkk	Z,N	
Adatmemória ↔ programmemória műveletek					
TBLRD*	tábla olvasás	2	0000 0000 0000 1000	nincs	
TBLRD*+	tábla olvasás utólagos növeléssel		0000 0000 0000 1001	nincs	
TBLRD*-	tábla olvasás utólagos csökkentéssel		0000 0000 0000 1010	nincs	
TBLRD+*	tábla olvasás előzetes növeléssel		0000 0000 0000 1011	nincs	
TBLWT*	tábla írás	2(5)	0000 0000 0000 1100	nincs	
TBLWT*+	tábla írás utólagos növeléssel		0000 0000 0000 1101	nincs	
TBLWT*-	tábla írás utólagos csökkentéssel		0000 0000 0000 1110	nincs	
TBLWT+*	tábla írás előzetes növeléssel		0000 0000 0000 1111	nincs	

Megjegyzések:

1. Amikor a PORT értéke változik, és a művelet eredménye önmagába íródik (pl. MOVF PORTB,0,1 vagy MOVF PORTB,F,A) az az érték kerül beírásra, ami jelenleg a lábon mérhető. Pl. ha a tárolt adat a regiszterben 1, és a láb bemenetnek van konfigurálva, és ezt a lábat egy külső eszköz 0-ba húzza, akkor a regiszterbe 0 kerül visszaírásra.
2. Abban az esetben, ha a művelet a TMR0 regiszterre vonatkozik (d=1, vagy d=f), és az előosztó a TMR0-hoz van rendelve, akkor az előosztó törlődik.
3. Amikor a programszámláló (PC) változik az utasítás két ciklus hosszú lesz. A második ciklusban a NOP utasítás kerül végrehajtásra.
4. Némely utasítás 2-szó hosszúságú. A második szó ezekben az utasításokban NOP-ként hajtódik végre, kivéve ha az utasítás első szavának folytatása ez a 16 bit.

3.7.3 Megszakítási rendszer

A 18-as sorozatban a megszakításkezelés is alapvető változáson ment keresztül. A 16-os mikrovezérlőknél a már futó interrupt megszakítására nem volt lehetőség. Itt kétszintű megszakítási rendszert találhatunk. Létezik egy alacsony és egy magas prioritási szint. Minden egyes megszakítási forrásnál meg tudjuk adni, hogy az alacsony, vagy magas prioritású legyen (IP bit). Alapesetben a mikrovezérlő egyszintű megszakítási rendszert – a kétszintűt külön be kell állítani – használ, ilyenkor csak magas prioritású megszakítások léteznek, amelyeket más IT rutin nem szakíthat meg. A kétszintű megszakítást az **RCON** regiszter **IPEN** bitjével lehet engedélyezni. Ilyenkor megváltozik a **GIE** és a **PEIE** bit szerepe. A **GIE** lesz a **GIEH**, a **PEIE** pedig a **GIEL**. A **GIEH** bittel lehet engedélyezni-tiltani az összes magas, a **GIEL** bittel pedig az összes alacsony prioritású megszakítást. A két szint miatt két megszakítási vektor létezik. A magas prioritású megszakítások a 0008h címre, az alacsonyak pedig a 0018h címre érkeznek. A következő programrészlet bemutatja a megszakítási vektorok elhelyezkedését, és a rutinok meghívását:

```
;*************************************************************************
;Magas proiritású megszakítási vektor
; Az itt lévő kód akkor hajtódik végre, amikor magas prioritású megszakítás történik, vagy
; bármely megszakítás, ha a prioritások nincsenek engedélyezve.

            ORG     0x0008

            BRA     HighInt                 ;ugrás a magas prioritású
;megszakítási rutinra

;*************************************************************************
;Alacsony proiritású megszakítási vektor és rutin
; Az itt lévő kód akkor hajtódik végre, amikor alacsony prioritású megszakítás történik.
; Az ittlévő programot el lehet távolítani, ha nem használjuk az alacsony prioritású ;megszakítást.
;A MAGAS PRIORITÁSÚ MEGSZAKÍTÁS FELÜLÍRJA A FAST REGISZTERT, EZÉRT ;SZOFTVERESEN KELL MENTENI, HA HASZNÁLJUK AZ ALACSONY PRIORITÁSÚ ;MEGSZAKÍTÁST!
            ORG     0x0018

            MOVFF    STATUS,STATUS_TEMP ;STATUS regiszter mentése
            MOVFF   WREG,WREG_TEMP      ;munka regiszter mentése
            MOVFF   BSR,BSR_TEMP        ;BSR regiszter mentése

;           *** ide kerül az alacsony prioritású megszakítás rutin***
```

```
            MOVFF   BSR_TEMP,BSR            ;BSR regiszter visszaállítása
            MOVFF   WREG_TEMP,WREG          ;munka regiszter visszaállítása
            MOVFF   STATUS_TEMP,STATUS      ;STATUS regiszter visszaállítása
            RETFIE

;*************************************************************************
;Magas prioritású megszakítás rutin
;A magas prioritású megszakítás rutint ide írjuk, hogy elkerüljük az ütközést
;az alacsony prioritású megszakítási vektorral.

HighInt:

;           *** ide kerül a magas prioritású megszakítás rutin***
;CALL       INT_SERV,FAST
;MEGSZ. RUTIN HIVASA STATUS, BSR, WREG MENTESEVEL

            BTFSC   PIR1,RCIF
            CALL    RC_INT,FAST

            BTFSC   INTCON,INT0IF
            CALL    HALL_INT,FAST

            BTFSC   PIR1,ADIF
            CALL    AD_INT,FAST

            RETFIE  FAST        ;STATUS, BSR, WREG visszaállítása
;*************************************************************************************************
```

Látható, hogy a megszakítási rutin hívásánál használhatjuk azt a lehetőséget, hogy a rutinra ugrás előtt automatikusan elmentjük (FAST opció) a BSR, a STATUS és a WREG tartalmát az árnyékregiszterekbe, majd pedig a visszatérésnél helyreállítjuk. A 16-os mikrovezérlőknél erre külön rutint kellett írni, aminek elhagyása, ill. hibás használata rengeteg „rejtélyes” hiba forrása volt. Sajnos, ha kétszintű megszakítást alkalmazunk, akkor az alacsony prioritásúra ugyanúgy programot kell, mert csak egy árnyékregisztertömböt építettek be.

3.7.4 Konfigurációs bitek

A 18-as sorozatnál lényegesen nagyobb feladatot jelent a konfigurációs bitek beállítása. Ez köszönhető annak, hogy lényegesen több a periféria – sok a többfunkciós kivezetés – másrészt a meglévő szolgáltatások is kibővültek (pl. watchdog timer). Szerencsére a konfigurációs bitek beállításához segítséget kaphatunk az adott mikrovezérlőhöz a gyártó által adott .inc fájlban. Ezek a fájlok a *C:\Program Files\Microchip\MPASM Suite könytárban* találhatóak. Az

alább látható programrészlet példát mutat a 18-as mikrovezérlő konfigurációs bitjeinek beállítására:

```
;konfigurációs bitek
; A CONFIG direktívával lehet megadni a konfiguráció adatait.
; A szimbólumok azonosak a P18F442.INC fájlban megadottakkal.
; The PIC18FXX2 adatlapja tartalmazza konfigurációs bitek leírását.
; A következő sorokat az alkalmazáshoz illeszkedően meg kell változtatni.
    ;CONFIG  OSC=XT          ;kvarc oszcillátor
    CONFIG   OSC=INTIO67    ;belső kvarc oszcillátor RA6-7 láb üzemel
    CONFIG   FCMEN=ON       ;oszcillátor monitorozás be
    CONFIG   IESO=OFF       ;belső-külső oszcillátor váltás ki
    CONFIG   PWRT=ON        ;oszcillátor bekapcsolása akkor, ha stabil a táp
    CONFIG   BOREN=ON       ;kritikus tápfeszültség reszet be
    CONFIG   BORV=2         ;reszet szintje
    CONFIG   WDT=ON         ;watchdog timer be
    CONFIG   WDTPS=128      ;128-as előosztó
    CONFIG   MCLRE=OFF      ;reszet láb ki, MCLR I/O lábként üzemel
    CONFIG   LPT1OSC=OFF    ;másodlagos oszcillátor ki
    CONFIG   PBADEN=OFF     ;PORTB0-4 digitális I/O
    CONFIG   STVREN=OFF     ;reszet verem túl-alul csordulás hatására ki
    CONFIG   LVP=OFF        ;alacsony feszültségű programozás ki
    CONFIG   XINST=OFF      ;kiterjesztett utasításkészlet ki, indexelés ki
    CONFIG   DEBUG=OFF      ;nyomkövetés ki
    CONFIG   CP0=OFF        ;kódvédelem ki
    CONFIG   CP1=OFF
    CONFIG   CPB=ON         ;boot-blokk védelem (1. 128 sor) be
    CONFIG   CPD=OFF        ;EEPROM védelem ki
    CONFIG   WRT0=OFF       ;programmemória írásvédelem ki
    CONFIG   WRT1=OFF
    CONFIG   WRTB=OFF       ;tábla írásvédelem ki
    CONFIG   WRTC=OFF       ;konfigurációs bitek írásvédelme ki
    CONFIG   WRTD=OFF       ;EEPROM írásvédelme ki
    CONFIG   EBTR0=OFF      ;tábla kiolvasás védelem ki
    CONFIG   EBTR1=OFF
    CONFIG   EBTRB=OFF
```

Ezzel befejeződött az utolsó témakörünk is. A feladatok megoldásához, a

tanításhoz és a tanuláshoz is sok örömöt kívánok, hiszen megfelelően végezve a tanulás és a tanítás is öröm!

5. Kitekintés

Az idő nem áll meg. A technika, a technológia folyamatosan változik. Különösen igaz ez az informatika területére. Amit ma papírra vetek, az lehet, hogy holnapra már elévült. Az oktatásban ezt a hihetetlen ütemű változást nagyon nehéz követni. Ennek ellenére egy jó pedagógusnak mindent el kell követnie ennek érdekében. Ehhez szükséges lenne, hogy szoros kapcsolatban legyen a tanár az iparral. Enélkül a pedagógus egy zárt világba kerül, sokkal kevésbé lesz képes megújulni, új dolgokat bevinni az oktatásba. Tapasztalataim szerint a megszokás nagy úr. Kevesen hajlandóak a régi jól bevált módszerüket, tananyagukat, stb. újabbra cserélni. Még akkor sem teszik ezt, ha köztudomásúan korszerűtlen is. Ez a megújulás különösen nehéz akkor, ha a pedagógusnak nincs élő tapasztalata ezekről az új ismeretekről. Véleményem szerint bizonyos időnként lehetőséget kellene biztosítani a pedagógusok számára ezen ismeretek befogadására.

Az itt ismertetett eszközök is folyamatosan fejlődnek, készülnek újabb berendezések. A honlapomon ezt a változást igyekszem mindig folyamatosan követni. A legnagyobb ellenségem az idő! Ezek a munkák sok idejét elveszik az embernek. Szerencsére volt diákjaim között találtam többeket is, akik ebben a munkában segítségemre voltak. Itt szeretnék köszönetet mondani Kerényi Pálnak és Varga Lászlónak, akik nyomtatott áramkörök tervezésében, mintaprogramok írásában, valamint ötletgazdaként folyamatosan segítik munkámat.

Időközben elkészült a próbapanel második és harmadik generációs fejlesztése is. A második generációs a 16F87X sorozatú mikrovezérlőkhöz készült. Az ötletet a győri mikrovezérlő versenyek gyakorlati feladatai adták, ez alapján készült a kapcsolás. Felépítésében lényeges különbség, hogy 4 db hétszegmenses kijelzőt tud kezelni. Néhány mintapélda a próbapanelhoz:

- közlekedési lámpa vezérlése
- reakcióidő mérése
- digitális óra
- számológép, stb.

A harmadik generációs panel már a legújabb fejlesztésű 18xx2 sorozatú mikrovezérlőkhöz készült. Ez egy generációváltást jelent a PIC mikrovezérlőknél. Nagyon sok hasznos újítást tartalmaz elődjeihez képest. Természetesen ennek megfelelően a programozása is változik. A legújabb próbapanel két darab PIC18F442-es (452-es) mikrovezérlőt képes fogadni! Újdonság a panelon, hogy a mikrovezérlőket nem szükséges kivenni a foglalatból, az ICSP (In Circuit Serial Programming) csatlakozón keresztül égethető. A panelon megmaradt mindaz, ami régről jól bevált – 4db hétszegmenses kijelző, 4db kétszínű LED, 4db kapcsoló – valamint kiegészült rengeteg új alkalmazással:

- 8 db piros LED
- 1 db RGB LED
- 1 db 4x16-os LCD kijelző
- 1db 3x4-es mátrix tasztatúra
- 2 db soros EEPROM
- 4 db potenciométer az analóg bemenetekre kapcsolva
- PWM kimenet
- Soros vonali csatlakozás a PC felé

A két mikrovezérlő miatt lehetőség nyílik kommunikációs csatornák kiépítésére is. Ennek szellemében összeköttetés van közöttük soros vonalon és I^2C buszon is. Ennek a panelnak jelenleg csak a prototípusa van kész, a végleges verzióval a tesztelések miatt még várni kell. Folyamatban van a PIC18xx2-es sorozat angol nyelvű leírásának fordítása is, azonban ez vélhetően hosszú folyamat lesz, hiszen több mint 300 oldalas dokumentációról van szó.

Írásommal igyekeztem a legkorszerűbb ismereteket nyújtani a pedagógusok és a diákok számára egyaránt. Remélem munkám elérte célját, és sok mindenkivel sikerült megszerettetnem ezt a szép szakmát!

Irodalomjegyzék

Cserny László (2001): *Mikroszámítógépek*
LSI Oktatóközpont, Vác

Dr. Kónya László (2003): *PIC mikrovezérlők alkalmazástechnikája*.
ChipCAD Elektronikai Disztribúció Kft, Budapest.

PIC16F84 DATA SHEET
1998 Microchip Technology Inc., Arizona.

http://www.ic-prog.com

http://www.picallw.com

http://www.microchip.com

http://www.chipcad.hu

Mellékletek

1. Univerzális programozó készülék

1.1 Általános leírás

Az ismertetésre kerülő programozó készülék egyszerűsége ellenére széles körben alkalmazható. Többféle programozó szoftverrel is – IC-PROG, PICALL – képes együttműködni. A programozó nagyon sokféle PIC mikrovezérlő égetésére alkalmas, itt látható egy kivonatos lista ezekből:

- **12xx-es sorozat**

PIC12F508	PIC12F609	PIC12HV615	PIC12F675
PIC12F509	PIC12HV609	PIC12F629	rfPIC12F675
PIC12F510	PIC12F615	PIC12F635	PIC12F68

- **16xx-es sorozat**

PIC16F54	PIC16F690	PIC16F874	PIC16C66
PIC16F57	PIC16F72	PIC16F874A	PIC16C67
PIC16F59	PIC16F73	PIC16F876	PIC16C620/A
PIC16F505	PIC16F74	PIC16F876A	PIC16C621/A
PIC16F506	PIC16F76	PIC16F877	PIC16C622/A
PIC16F610	PIC16F77	PIC16F877A	PIC16CE623
PIC16HV610	PIC16F716	PIC16F882	PIC16CE624
PIC16F616	PIC16F737	PIC16F883	PIC16CE625
PIC16HV616	PIC16F747	PIC16F884	PIC16C71
PIC16F627	PIC16F767	PIC16F886	PIC16C72
PIC16F627A	PIC16F777	PIC16F887	PIC16C72a
PIC16F628	PIC16F785	PIC16F913	PIC16C73
PIC16F628A	PIC16HV785	PIC16F914	PIC16C73A/B
PIC16F630	PIC16F83	PIC16F916	PIC16C74
PIC16F631	PIC16F84	PIC16F917	PIC16C74A/B
PIC16F636	PIC16F84A	PIC16F946	PIC16C76
PIC16F639	PIC16F87	PIC16C61	PIC16C77
PIC16F648A	PIC16F88	PIC16C62	PIC16C710
PIC16F676	PIC16F818	PIC16C62A/B	PIC16C711
PIC16F677	PIC16F819	PIC16C63	PIC16C712
PIC16F684	PIC16F870	PIC16C63A	PIC16C716
PIC16F685	PIC16F871	PIC16C64	PIC16C745
PIC16F687	PIC16F872	PIC16C64A	PIC16C765
PIC16F688	PIC16F873	PIC16C65	PIC16C773
PIC16F689	PIC16F873A	PIC16C65A/B	PIC16C774

PIC16C923	PIC16C924	PIC16C925	PIC16C926

- **18xx-es sorozat:**

PIC18F1220	PIC18F4410	PIC18F8310	PIC18F67J11
PIC18F1230	PIC18F442-4439	PIC18F8390	PIC18F67J50
PIC18F1320	PIC18F4420	PIC18F8410	PIC18F67J60
PIC18F1330	PIC18F4423	PIC18F8490	PIC18F83J11
PIC18F1330	PIC18F4431	PIC18F8520	PIC18F83J90
PIC18F2220	PIC18F4450	PIC18F8525	PIC18F84J11
PIC18F2221	PIC18F4455	PIC18F8527	PIC18F84J90
PIC18F2320	PIC18F4458	PIC18F8585	PIC18F85J10
PIC18F2321	PIC18F448	PIC18F8620	PIC18F85J11
PIC18F2331	PIC18F4480	PIC18F8621	PIC18F85J15
PIC18F2410	PIC18F4510	PIC18F8622	PIC18F85J50
PIC18F242-2439	PIC18F4515	PIC18F8627	PIC18F85J90
PIC18F2420	PIC18F452-4539	PIC18F8680	PIC18F86J10
PIC18F2423	PIC18F4520	PIC18F8720	PIC18F86J11
PIC18F2431	PIC18F4523	PIC18F8722	PIC18F86J15
PIC18F2450	PIC18F4525	PIC18F24J10	PIC18F86J16
PIC18F2455	PIC18F4550	PIC18F25J10	PIC18F86J50
PIC18F2458	PIC18F4553	PIC18F44J10	PIC18F86J55
PIC18F248	PIC18F458	PIC18F45J10	PIC18F86J60
PIC18F2480	PIC18F4580	PIC18LF24J10	PIC18F86J65
PIC18F2510	PIC18F4585	PIC18LF25J10	PIC18F87J10
PIC18F2515	PIC18F4610	PIC18LF44J10	PIC18F87J11
PIC18F252-2539	PIC18F4620	PIC18LF45J10	PIC18F87J50
PIC18F2520	PIC18F4680	PIC18F63J11	PIC18F87J60
PIC18F2523	PIC18F4682	PIC18F63J90	PIC18F96J60
PIC18F2525	PIC18F4685	PIC18F64J11	PIC18F96J65
PIC18F2550	PIC18F6310	PIC18F64J90	PIC18F97J60
PIC18F2553	PIC18F6390	PIC18F65J10	PIC18F13K50
PIC18F258	PIC18F6410	PIC18F65J11	PIC18LF13K50
PIC18F2580	PIC18F6490	PIC18F65J15	PIC18F14K50
PIC18F2585	PIC18F6520	PIC18F65J50	PIC18LF14K50
PIC18F2610	PIC18F6525	PIC18F65J90	PIC18F23K20
PIC18F2620	PIC18F6527	PIC18F66J10	PIC18F24K20
PIC18F2680	PIC18F6585	PIC18F66J11	PIC18F25K20
PIC18F2682	PIC18F6620	PIC18F66J15	PIC18F26K20
PIC18F2685	PIC18F6621	PIC18F66J16	PIC18F43K20
PIC18F4220	PIC18F6622	PIC18F66J50	PIC18F44K20
PIC18F4221	PIC18F6627	PIC18F66J55	PIC18F45K20
PIC18F4320	PIC18F6680	PIC18F66J60	PIC18F46K20
PIC18F4321	PIC18F6720	PIC18F66J65	
PIC18F4331	PIC18F6722	PIC18F67J10	

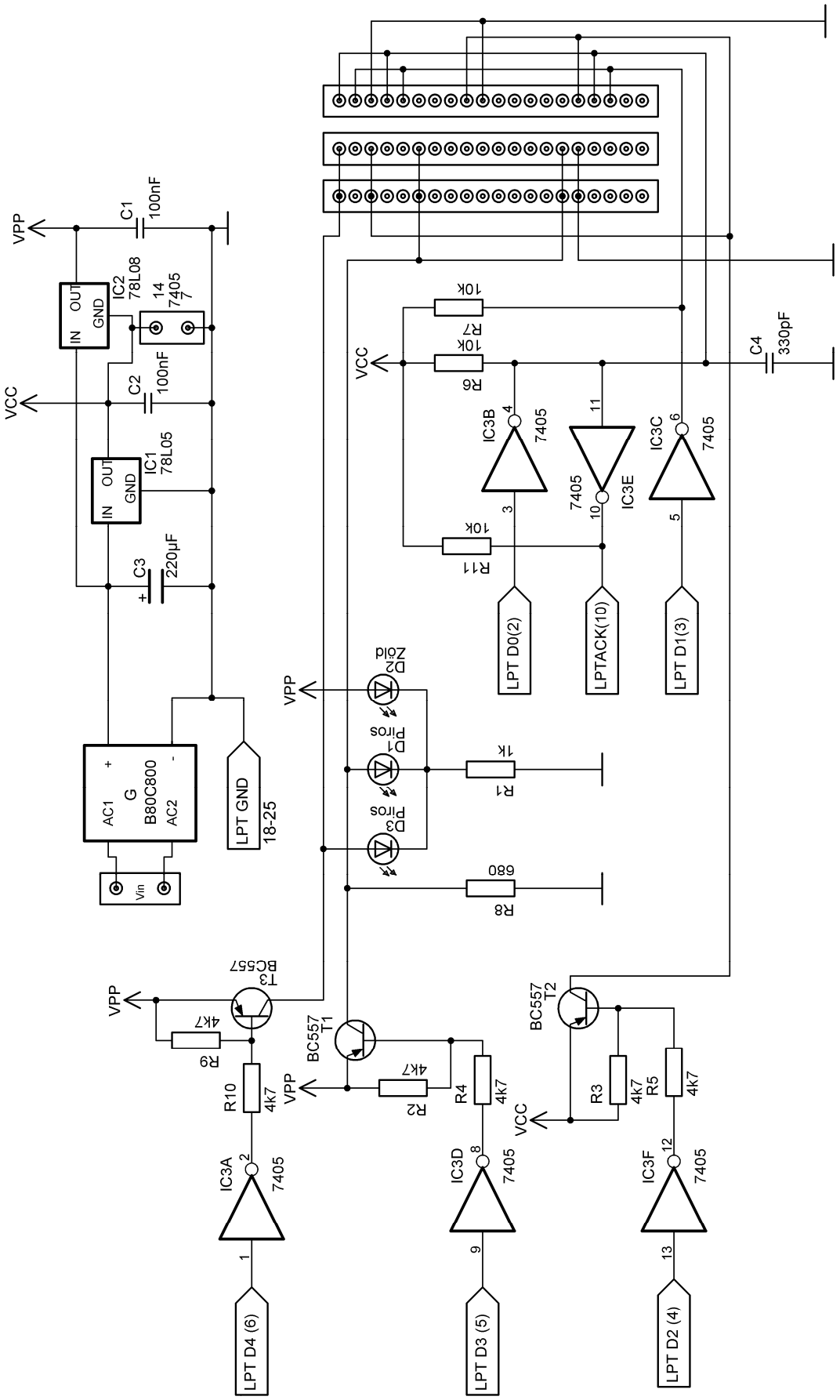
1.2 A programozó működése

A programozó működése az 1. ábrán követhető nyomon. A bejövő váltakozófeszültséget a G_1 graetz-híd egyenirányítja, illetve egyenáramú táplálás esetén a polaritásfüggetlenséget biztosítja. Az egyenirányított jel simítását a C_3 kondenzátor végzi. Az IC_1 feszültségstabilizátor a programozni kívánt mikrovezérlő stabil +5V-os tápellátását biztosítja. Az IC_1 „tetejére ültetett" 8V-os IC_2 stabilizátor az égetéshez szükséges minimum 12,5V-os programozófeszültséget (V_{pp}) állítja elő. A C_1 és a C_2 kondenzátorok a stabilizátorok begerjedését akadályozzák meg. A 3 db PNP tranzisztor kapcsolóelemként működik, a programozófeszültségeket és a tápfeszültséget kapcsolják a mikrovezérlőre. A T_2 kapcsolja a tápfeszültséget. 40 és 28 lábú tokok esetén a T_3 kapcsolja, a többinél pedig a T_1 kapcsolja a programozófeszültséget. A programozó a vezérlést a számítógép párhuzamos portjáról kapja. Az IC_3 – 6 db invertáló vagy neminvertáló buffert tartalmaz – feladata az, hogy megfelelő leválasztást biztosítson a számítógép és az égető között. Fontos, hogy a programozó szoftver beállításainál a megfelelő buffert válasszuk ki mind a PICALL-ban, illetve az IC-PROG-ban. A három LED (D_1-D_3) a programozó működéséről ad tájékoztatást. A D_3 zöld LED a tápfeszültség meglétét jelzi, a két piros (D_1, D_2) pedig a programozás idején világít. A D_3 a 40 és a 28 lábú tokok esetében, a D_2 pedig az összes többinél jelez.

1.3 A programozó megépítése, élesztése

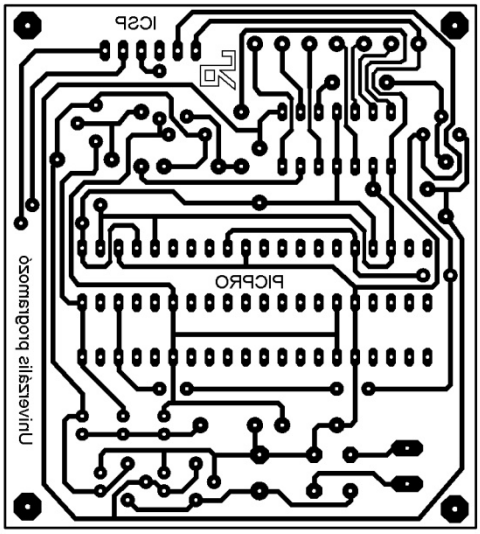
A 2. ábrán látható általam tervezett nyomtatási rajz alapján fotózásos, vagy egyéb más technológiával készítsük el a nyomtatott áramkört. Az elkészült nyákot fény felé tartva ellenőrizzük, hogy nincs-e rajta szakadás vagy zárlat. Fúrás után tisztítsuk le, és vonjuk be valamilyen forraszthető védőlakkal, hogy a rézfólia ne oxidálódhasson.

Az alkatrészeket készítsük elő a beültetéshez. A beültetést a 3. ábra alapján végezhetjük el. Először az átkötéseket forrasszuk be. Ezután következnek az ellenállások, majd a 3 sor csatlakozó hüvely, valamint a graetz-híd. Az ICSP csatlakozó tüskesorából vegyük ki a negyediket, úgy forrasszuk be. Az IC_3-at forrasszuk be ezután (nem szükséges foglalatba rakni), majd folytassuk a sort a

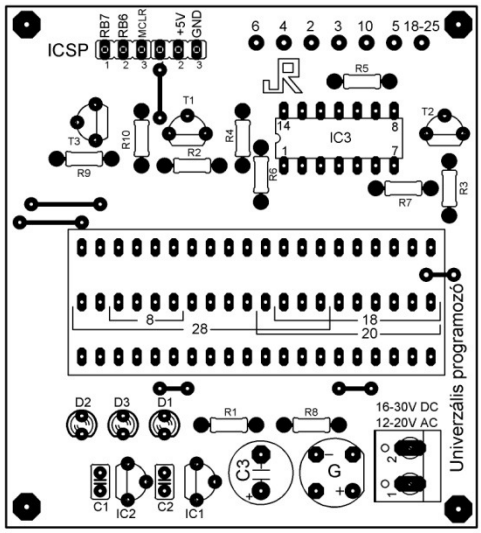

VPP
C1
100nF
IC2
78L08
IN
OUT
GND
14
7405
7
C2
100nF
VCC
IC1
78L05
C3
220µF
AC1
G
B80C800
AC2
Vin
LPT GND
18-25
10K
R7
R6
C4
330pF
IC3B
IC3E
IC3C
7405
11
3
10
5
R11
LPT D0(2)
LPTACK(10)
LPT D1(3)
D2
Zöld
D1
Piros
D3
1K
R1
680
R8
BC557
T3
T1
T2
4K7
R9
R10
R2
R4
R3
R5
IC3A
IC3D
IC3F
2
8
12
1
9
13
LPT D4 (6)
LPT D3 (5)
LPT D2 (4)

1. ábra

két stabilizátor IC-vel és a 3 darab tranzisztorral. Rakjuk be a gerjedésgátló kondenzátorokat, valamint a tápfeszültség csatlakoztatására szolgáló sorkapcsot. Ültessük be a három LED-et egyforma magasságban, ügyelve a polaritásukra, végezetül a C_3 pufferkondenzátort is tegyük a helyére. Miután mindezzel végeztünk, készítsük a számítógéphez csatlakozó kábelünket. Ehhez a kereskedelemben kapható 8x0,22-es, vagy 2x0,5+6x0,22-s sodrott rézvezetéket használjunk. A hossza kb. 1,5 méter legyen. A vezetéket csupaszoljuk le, és egy ér kivételével ónozzuk elő. A 25 pólusú csatlakozó használni kívánt lábait szintén futtassuk be forrasztóónnal. Ezután egy csupasz vezetékdarabbal kössük össze 18-25-ig a csatlakozó lábait. Forrasszunk egy-



2. ábra



3. ábra

egy vezetéket a csatlakozó 2-es, 3-as, 4-es, 6-os, 10-es és az összekötött 18-25-ös lábaihoz. Célszerű egy papírra felírni, hogy milyen színű vezetékeket forrasztottunk az egyes lábakhoz, mert ez megkönnyíti az azonosítást. A vezeték másik végét forrasszuk be a nyákba, a beültetési rajzon lévő feliratok a csatlakozó megfelelő kivezetéseit jelölik. Multiméter segítségével ellenőrizzük az összeköttetések helyességét. Szereljük össze a csatlakozót – ne feledkezzünk meg a kiszakadás-gátlóról sem – befejezésül.

Az elkészült áramkörünkre kapcsoljunk tápfeszültséget. A zöld LED-nek ilyenkor világítania kell. Multiméterrel ellenőrizzük a feszültséget az IC_1 (5V) és az IC_2 kimenetén (13V). Csatlakoztassuk a programozót a számítógéphez, és pl. a PICALL programmal próbáljunk meg beégetni egy programot a mikrovezérlőnkbe. Ügyeljünk arra, hogy megfelelően pozícionáljuk a

mikrovezérlőt (3. ábra). A mikrovezérlő kivételét a programozóból csipesszel, vagy a speciálisan erre készített IC kiszedővel végezzük. Szabadkézzel sose próbálkozzunk, mert a hirtelen kiugró IC lábai az ujjunkba állhat, balesetet okozva ezzel!

1.4 Alkatrészjegyzék

Ellenállás:

R_1=1 kΩ

$R_{2, 3, 4, 5, 9, 10}$=4,7 kΩ

$R_{6, 7}$=10 kΩ

R_8=680 Ω

R_{11}^*=10 kΩ

Kondenzátor:

$C_{1, 2}$=100nF

C_3=220 µF/50V

C_4^*=330 pF

Integrált áramkör:

IC_1=78L05

IC_2=78L08

IC_3=74LS05

Tranzisztor:

$T_{1, 2, 3}$=BC557

Graetz:

G=B80C1500

LED:

$D_{1,3}$=∅3 mm piros LED

D_2=∅3 mm zöld LED

Egyéb:

3 db 20 lábú csatlakozó hüvely

1 db 6 lábú 6/3 mm-es tüskesor

1,5 m 8x0,22-es 8 eres vezeték

1 db D-SUB 25 dugó (printer)

1 db 2-es nyák sorkapocs

Megjegyzés: a *-gal jelölt alkatrészeket nem kell beültetni

2. Mikrovezérlő próbapanel

2.1 Általános leírás

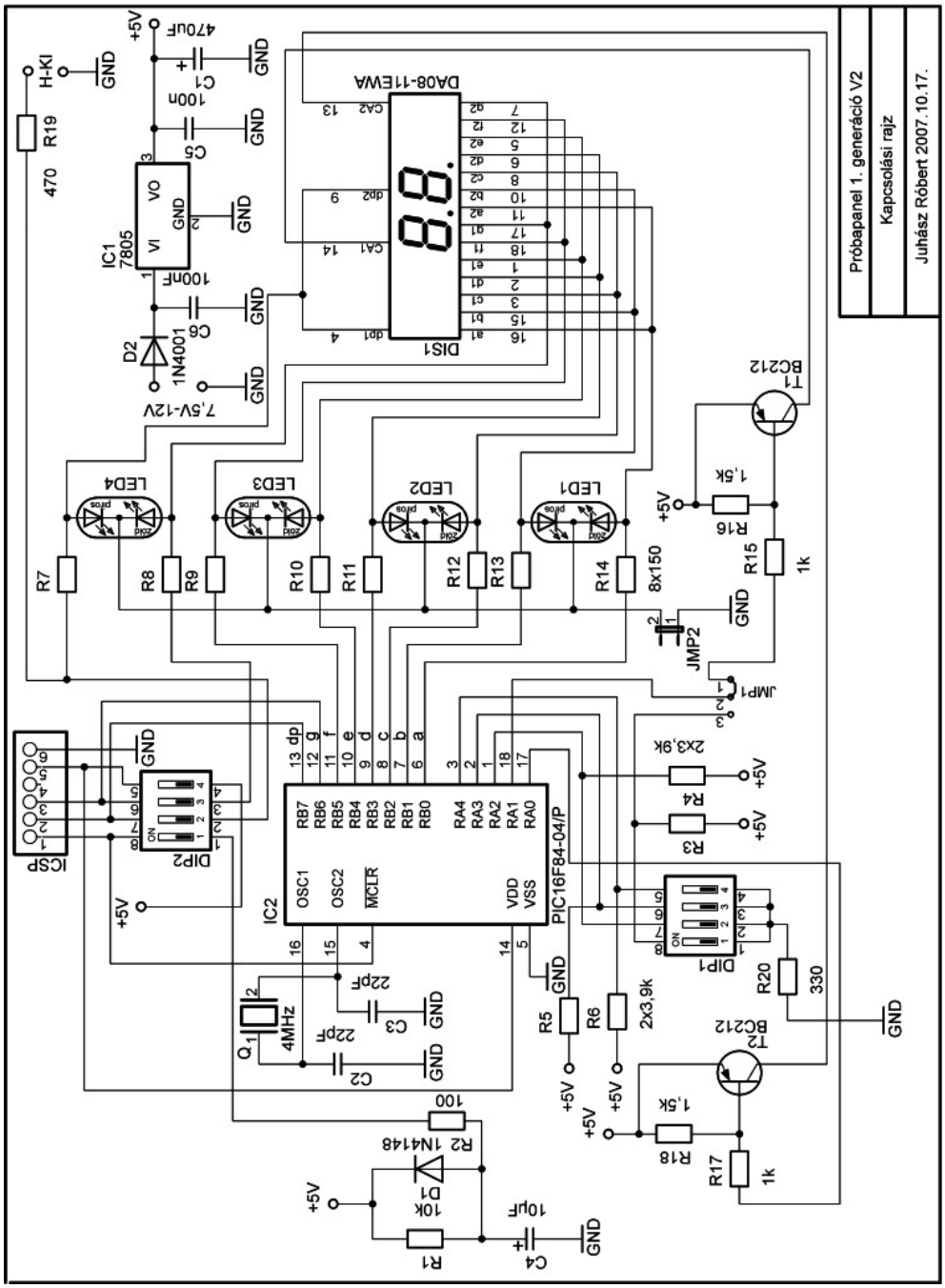
Nem elég csak megírni a programokat, hanem azokat ki is kell próbálni. Az igazi élményt ugyanis az adja a tanulóknak, ha látják működni alkotásaikat. A szimuláció sohasem helyettesítheti a gyakorlatot! A próbapanel a PIC16F84-es mikrovezérlőre megírt programok tesztelésére szolgál. Felépítése egyszerű, házilag is könnyen kivitelezhető, mégis sokféle funkciót egyesít magában. A próbapanel elsősorban kezdők számára készült. A panelen különböző feladattípusok gyakorlására nyílik lehetőség (ezt természetesen bárki bővítheti ízlése szerint, az itt felsoroltak csak mintául szolgálnak):

- Kimenetek be- és kikapcsolása
- Kapcsolók lekérdezése, pergésmentesítés
- Hétszegmenses kijelző illesztése, multiplex üzemmód
- Megszakítások kezelése
- Ugrótáblák, adattáblák használata
- Számkonverziók (bináris - BCD, hexadecimális - decimális, stb.)
- PWM jel előállítása (fényerő-szabályozás, hanggenerálás)

2.2 A próbapanel működése

A próbapanel kapcsolása a 4. ábrán látható, működését ez alapján érthetjük meg. A bejövő egyenfeszültséget az IC_1 stabilizálja (5V) az áramkörök számára, a D_2 dióda a fordított polaritás ellen véd. A C_1 kondenzátor pufferként működik, a C_5 és a C_6 az IC_1 begerjedését akadályozza meg. A mikrovezérlő órajelét a Q, C_2 és C_3 elemek biztosítják.

A 4MHz-es kvarckristályból a mikrovezérlő egy négyfázisú órajelet állít elő, így a működési frekvenciája 1MHz lesz (1 gépi ciklus 1μs). A programok írásánál, a konfigurációs biztosítékoknál XT oszcillátort kell beállítani. Fontos szempont a mikrovezérlős áramkörök tervezésénél, hogy a kvarckristályt és a két kondenzátort minél közelebb kell elhelyezni a mikrovezérlőhöz, mert ellenkező esetben az oszcillátor nem indul be!



4. ábra

A mikrovezérlő biztos reszetelését a tápfeszültség bekapcsolásakor az R_1, R_2, D_1 komplexum biztosítja.

A mikrovezérlő B portjára (RB7-RB0) csatlakozik a 4 db kétszínű LED (LED_1-LED_4), a hétszegmenses kijelzők katódjai (dp, g-a), valamint az RB7 láb külső eszközök csatlakoztatására alkalmas. A kétszínű LED-ek anódjai korlátozó-ellenállásokon (R_7-R_{14}) kapcsolódnak a PORTB-hez. Ugyanezen ellenállások a hétszegmenses kijelző LED-jeinek korlátozó ellenállásai is. A kétszínű LED-ek közösített katódjait a JMP2-es jumperrel lehet bekapcsolni. A hétszegmenses

kijelző használatakor ezt a jumpert el kell távolítani! A LED-ek bekapcsolása aktív 1-es szinttel történik. Az RB0 kapcsolja a LED_1 (zöld) anódját, az RB2 a LED_1 (piros) anódját, stb.

A DA08-11EWA típusú közös anódos kijelző két darab hétszegmenses kijelzőt tartalmaz. A kijelzők megfelelő kivezetései párhuzamosan vannak kapcsolva. Ahhoz, hogy a két kijelzőre különböző értékeket lehessen kiírni, a kijelzőket multiplex üzemmódban kell üzemeltetni. A multiplex üzemmódot a kijelzők (1-2) anódjainak megfelelő kapcsolgatásával lehet elérni. Az anódokat a mikrovezérlő RA0-RA1 lábára csatlakozó PNP tranzisztorokkal (T_2-T_1) lehet bekapcsolni. A közös anódos jelleg miatt mind az anódokat, mint a szegmenseket aktív nullával lehet működtetni. Az RA1 láb kettős funkciót lát el, a megfelelő működést a JMP1-es jumperrel lehet beállítani, jelen esetben 1-2 állásba kell tenni!

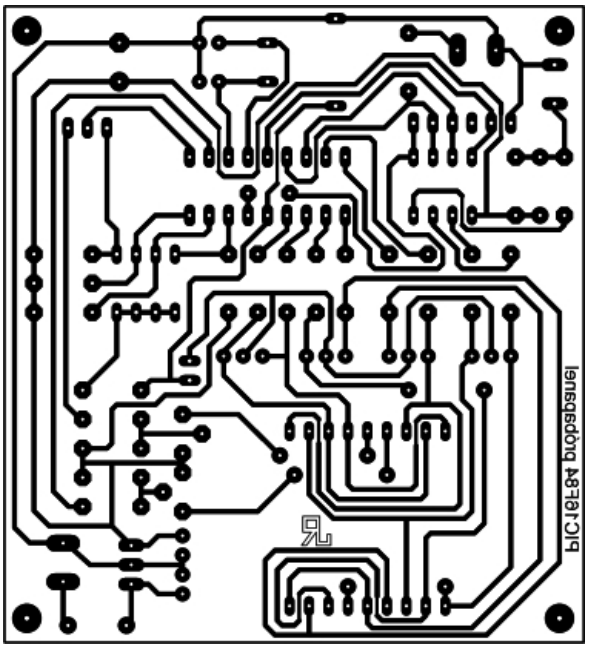
Az A portra csatlakozik egy 4-es DIP1 kapcsoló felhúzóellenásokon keresztül. Bekapcsolt állapotban a PORTA lábaira logikai nullát ad. Mind a négy kapcsoló használatához a JMP1-et 2-3 állásba kell kapcsolni. A DIP2-es kapcsoló kikapcsolt állapotában programozni lehet az ICSP-n keresztül, bekapcsolt állapotban pedig tesztelni lehet a programot.

A külső kimenet bekapcsolását a JMP3 rövidre zárásával lehet elérni.

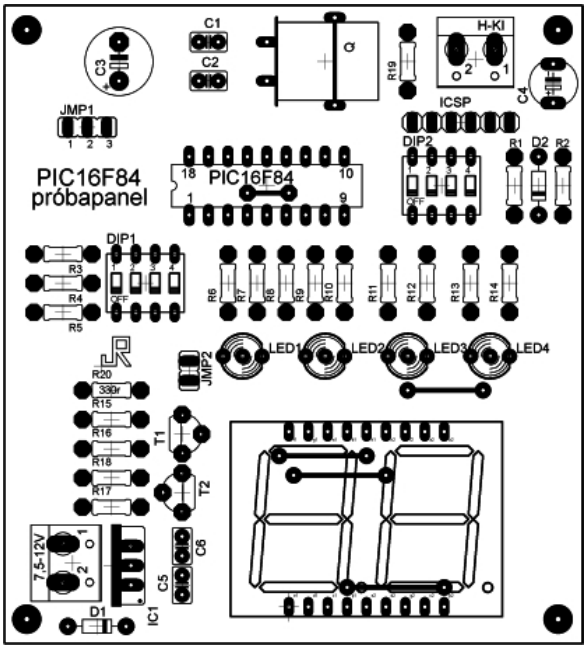
<table>
<tr><th rowspan="3">PORT</th><th colspan="5">JUMPER</th></tr>
<tr><th colspan="2">JMP1</th><th colspan="2">JMP2</th><th>JMP3</th></tr>
<tr><th>1-2</th><th>2-3</th><th>BE</th><th>KI</th><th>BE</th></tr>
<tr><td>RB7</td><td></td><td></td><td>LED4 P</td><td>LD 1-2 dp</td><td>H-KI</td></tr>
<tr><td>RB6</td><td></td><td></td><td>LED4 Z</td><td>LD 1-2 g</td><td rowspan="8"></td></tr>
<tr><td>RB5</td><td></td><td></td><td>LED3 P</td><td>LD 1-2 f</td></tr>
<tr><td>RB4</td><td></td><td></td><td>LED3 Z</td><td>LD 1-2 e</td></tr>
<tr><td>RB3</td><td></td><td></td><td>LED2 P</td><td>LD 1-2 d</td></tr>
<tr><td>RB2</td><td></td><td></td><td>LED2 Z</td><td>LD 1-2 c</td></tr>
<tr><td>RB1</td><td></td><td></td><td>LED1 P</td><td>LD 1-2 b</td></tr>
<tr><td>RB0</td><td></td><td></td><td>LED1 Z</td><td>LD 1-2 a</td></tr>
<tr><td>RA4</td><td colspan="2">DIP 4</td><td colspan="3" rowspan="5">a piros szín aktív 1-et (dőlt betű)
a kék aktív 0-át jelent (normál)</td></tr>
<tr><td>RA3</td><td colspan="2">DIP 3</td></tr>
<tr><td>RA2</td><td colspan="2">DIP2</td></tr>
<tr><td>RA1</td><td>LD 2 anód</td><td>DIP 1</td></tr>
<tr><td>RA0</td><td colspan="2">LD 1 anód</td></tr>
</table>

2.3 A próbapanel megépítése, élesztése

A nyomtatott áramköri terv alapján (5. ábra) készítsük el a panelt pl. fotózásos technológiával. A letisztított panelt fény felé tartva ellenőrizzük, hogy nincs-e rajta szakadás, vagy zárlat. Fúrjuk ki, és ismételt tisztítás után vonjuk be forraszthatóvédőlakkal, majd szárítsuk meg.



5. ábra



6. ábra

Az alkatrészjegyzék alapján készítsük elő az alkatrészeket beültetésre. A beültetést a 6. ábra szerint végezzük.

Először ültessük be az összes ellenállást – ügyelve az olvasási irányra –, majd a D_1 és D_2 jelű diódát. Folytassuk a sort a mikrovezérlő 18 lábú foglalatával, figyelve a megfelelő pozícióra. Ezután ültessük be a kvarcot és a hozzá tartozó két kondenzátort, valamint a C5 és C6 gerjedésgátló kondenzátort. A magas kvarcot fektetve ültessük és egy szigeteletlen vezetéket ráforrasztva, rögzítsük. Következhetnek a jumper tüskék, az ICSP tüskesora – jobbról a harmadik lábát húzzuk ki fogóval – és a két tranzisztor. A tranzisztorok után ültessük be a két DIP kapcsolót és a 7-szegmenses kijelzőt, mindkettőnél ügyeljünk a helyes pozícióra. A 4 LED magasságát úgy állítsuk be, hogy azonos legyen a kijelzőével, a pozicionálást itt is figyeljük. Végül forrasszuk be a két sorkapcsot, a két elektrolit-kondenzátort – polaritásra ügyelve – majd a

stabilizátort.

Az elkészült panelra kapcsoljunk megfelelő egyenfeszültséget (7,5-12V) és a mikrovezérlő foglalatánál az 5-ös és a 14-es láb között mérve ellenőrizzük a +5V-os tápfeszültséget. A reszet lábon (4-es) szintén +5V-ot kell mérnünk.

Abban az esetben, ha mindent rendben találunk egy PIC16F84-es mikrovezérlőbe égessük be a teszt.hex állományt (megtalálható a http://plc.mechatronika.hu oldalon). Feszültségmentes állapotban helyezzük a PIC-et a própabanelba, és végezzük el a tesztelést az alábbiak szerint:

1. **Piros futófény balról jobbra a 4 LED-en**
   - JMP1 jumper 1-2 pontját zárjuk rövidre
   - JMP2 jumpert zárjuk rövidre
   - JMP3 jumpert hagyjuk üresen
   - A DIP mind a 4 kapcsolóját állítsuk felső állásba
   - Kapcsoljunk az áramkörre 8V egyenfeszültséget

2. **Zöld futófény balról jobbra a 4 LED-en**
   - A DIP 4-es kapcsolóját állítsuk alsó állásba

3. **Sárga futófény balról jobbra a 4 LED-en**
   - A DIP 4-es kapcsolóját állítsuk vissza felső állásba
   - A DIP 3-as kapcsolóját kapcsoljuk le

4. **Hétszegmenses kijelzők tesztelése**
   - A JMP2-es jumpert vegyük le
   - A DIP 3-as és 4-es kapcsolóját tegyük alsó állásba

   Helyes működés esetén a tizedespont és a szegmensek a kijelző mindkét felén sorban s világítani kezdenek.

2.4 Alkatrészjegyzék

Ellenállás:

R_1=10kΩ

R_2=100Ω

R_{3-6}=3,9kΩ

R_{7-14}=150Ω

R_{15-17}=1kΩ

R_{16-18}=1,5kΩ

R_{19}=470Ω

Kondenzátor:

C_1=470µF

$C_{2,\ 3}$=22pF

C_4=10µF

$C_{5,\ 6}$=100nF

Dióda:

D_1=1N4148

D_2=1N4004

LED:

LED_1-LED_4=∅5mm kétszínű LED

Kijelző:

LD=DA08-11EWA hétszegmenses kijelző

Tranzisztor:

$T_{1,\ 2}$=BC212

Kvarc:

Q=4MHz

Jumper:

JMP_1=3-as jumper tüske

$JMP_{2,\ 3}$=2-es jumper tüske

Kapcsoló:

DIP1=4-es DIP kapcsoló

DIP2=4-es DIP kapcsoló

Integrált áramkör:

IC_1=7805

IC_2=PIC16F84-04/P

Egyéb:

3db jumper

6-os tüskesor (6/3)