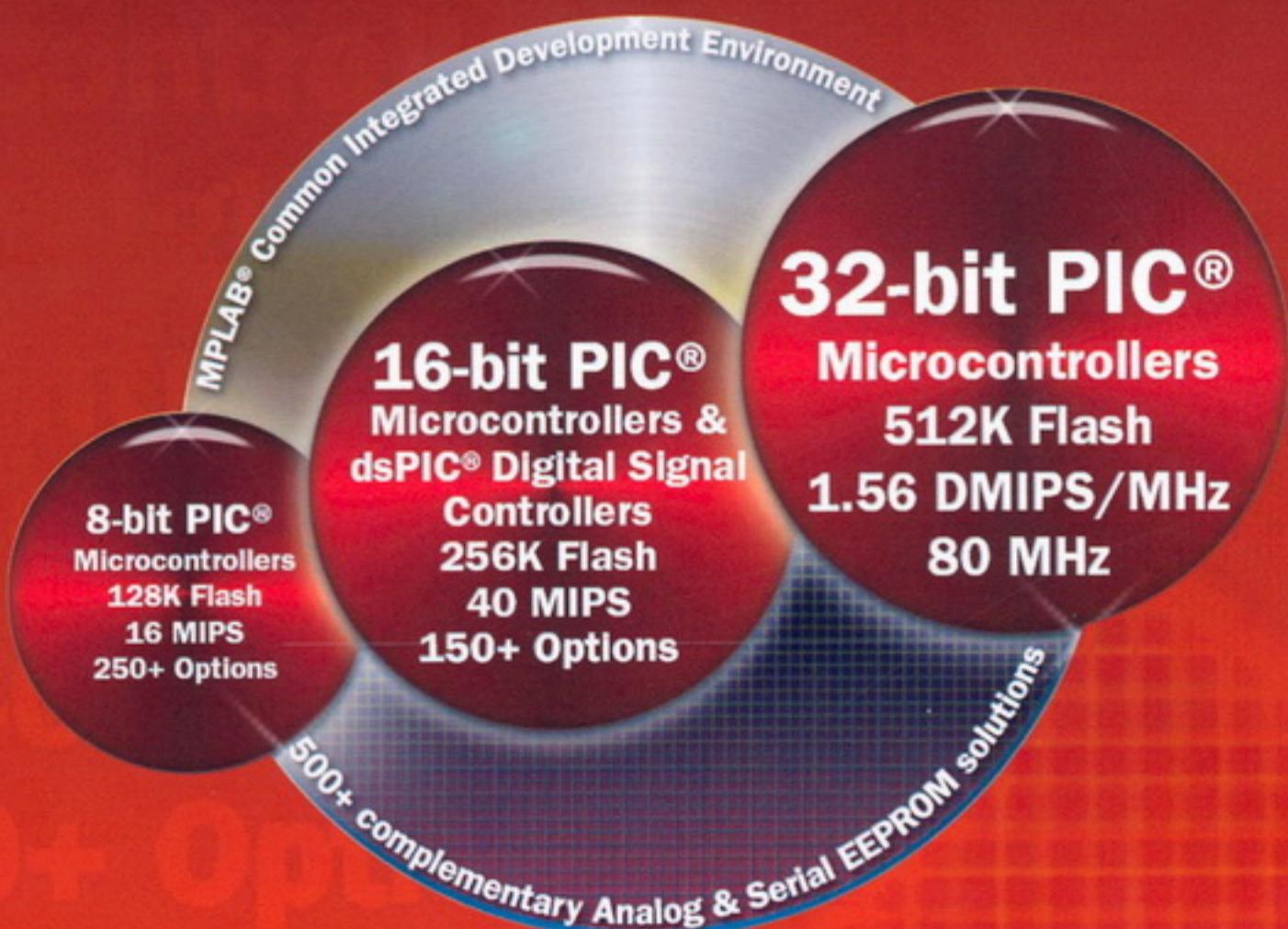


Kónya László–Kopják József

PIC MIKROVEZÉRLŐK ALKALMAZÁSTECHNIKÁJA

PIC PROGRAMOZÁS C NYELVEN



Harmadik, bővített kiadás

chipCAD
DISTRIBUTION

KÓNYA LÁSZLÓ–KOPJÁK JÓZSEF

**PIC MIKROVEZÉRLŐK
ALKALMAZÁSTECHNIKÁJA**

**PIC PROGRAMOZÁS
C NYELVEN**

HARMADIK, BŐVÍTETT, ÁTDOLGOZOTT KIADÁS

BUDAPEST
2009

© ChipCAD Elektronikai Disztribúció Kft., 2000, 2003, 2009
Minden jog fenntartva.

A jelen könyvet vagy annak részeit a Kiadó engedélye nélkül bármilyen formában vagy eszközzel reprodukálni tilos.

Felelős kiadó:

Dr. Holman Tamás, a ChipCAD Kft. ügyvezető igazgatója
1094 Budapest, Tüzoltó u. 31.
Postacím: 1476 Budapest Pf. 303.
Telefon: (1) 231-7000
Telefax: (1) 231-7011
E-mail: info@chipcad.hu
Internet: <http://www.chipcad.hu>

ISBN 978-963-06-6720-3

Lektorálta: ChipCAD Kft.
Borítóterv: Heiling Zsolt
Nyomdai előkészítés: Tertia Kiadó
Szedési munkálatok: Zimler Tamás
Nyomdai kivitelezés: Fólium Nyomda

E könyv szerzői igen különböző emberek – életkorban, tapasztalatban. Egy dolog azonban biztosan közös bennük: mindenkiten szeretnének köszönetet mondani mindazoknak, akik hozzájárultak ahhoz, hogy ez a könyv, amit most az Olvasó a kezében tart, megjelenhetett.

Kollégák, volt és jelenlegi tanítványok, munkatársak – a ChipCad Kft. dolgozói –, álljon itt a szerzők köszönete jeléül egy korántsem teljes névsor: Bors Ernő, Füzi Attila, Götz Tibor, Küllős Máté és még sokan mások.

A Szerzők néhány téma kidolgozásában nagyon értékes ötleteket és segítséget kaptak tőlük – és főleg sok türelmet, építő kritikát, amivel hozzájárultak a könyv színvonalához.

Külön köszönet illeti dr. Holman Tamás igazgató urat azért, hogy lehetővé tette a könyv megírását, és immár a harmadik kiadás megjelenését.

És természetesen a Szerzők külön-külön is hálával és köszönettel gondolnak a családra, a társakra, a barátokra, akik megteremtették azt a nyugodt, alkotó környezetet, ami az anyaggyűjtéshez, a könyv megírásához szükséges volt, és elfogadták, hogy a szabad időt sokszor nem velük, hanem e könyv írásával töltötték.

2009. február

DR. KÓNYA LÁSZLÓ, KOPJÁK JÓZSEF

TARTALOMJEGYZÉK

| | |
|---|----|
| Előszó | 13 |
| I. rész PIC MIKROVEZÉRLŐK ALKALMAZÁSTECHNIKÁJA (KÓNYA LÁSZLÓ) | |
| 1. Ismétlés | 19 |
| 1.1. A mikroprocesszor működése | 19 |
| 1.2. Mikroszámítógépek, mikrokontrollerek | 20 |
| 2. A PIC mikrovezérlök felépítése, fejlődése | 23 |
| 2.1. Aritmetikai-logikai egység (ALU) | 23 |
| 2.2. Programmemória | 24 |
| 2.2.1. Programmemória-típusok | 24 |
| 2.3. Adatmemória | 26 |
| 2.4. A program- és az adatmemória kapcsolata | 27 |
| 2.5. Megszakítás | 29 |
| 2.6. Rendszerelemek | 32 |
| 2.6.1. Oszcillátor, órajel | 32 |
| 2.6.2. Reset | 34 |
| 2.6.3. Watchdog | 38 |
| 2.6.4. Microchip nanowatt technológia | 39 |
| 2.6.5. Alacsony tápfeszültség figyelése (LVD) | 47 |
| 2.7. A PIC mikrovezérlök fejlődése | 48 |
| 2.8. 8 bites mikrovezérlő-családok (8/12, 8/14, e8/14, 8/16) | 49 |
| 2.8.1. PIC10F/PIC12F/PIC16F – 8/12-es család (Baseline Flash Microcontrollers) | 50 |
| 2.8.2. PIC12F/PIC16F – 8/14-es család (Mid Range Microcontrollers) | 53 |
| 2.8.3. PIC12F/PIC16F – e8/14-es család (Enhanced Mid Range Microcontrollers) | 55 |
| 2.8.4. PIC18F – 8/16-os család (High Performance Microcontrollers) | 58 |
| 2.9. 16/24-es mikrovezérlök, PIC24/PIC30/dsPIC33 családok | 66 |
| 2.9.1. A PIC24-es termékvonal | 67 |
| 2.9.2. dsPIC termékvonal: DSP PIC | 72 |
| 2.10. A 32/32 bites mikrovezérlök – PIC32-es család | 74 |
| 2.11. Összefoglalás | 80 |
| 3. Digitális perifériák | 81 |
| 3.1. I/O portok | 83 |
| 3.1.1. Bemeneti változások kezelése a 16/24 bites kontrollereknél | 85 |
| 3.1.2. Az érintésérzékelők kezelése | 85 |
| 3.1.3. Funkciók más lábakhoz rendelése – Peripheral Pin Select (PPS) | 88 |
| 3.2. Párhuzamos I/O port | 90 |
| 3.3. Időzítő/eseményszámláló modulok | 93 |
| 3.4. Naptár/óra modul (RTCC) | 95 |
| 3.5. A CCP modulok fejlődése | 96 |
| 3.6. Az Input Capture modul | 97 |
| 3.7. Output Compare (OC) /PWM modul | 98 |

| | |
|--|-----|
| 3.8. Motor Control PWM (MCPWM) | 101 |
| 3.9. Forgásérzékelés – QEI | 102 |
| 3.10. EEPROM modul | 104 |
| 3.11. LCD vezérlés | 105 |
| 3.12. DMA – közvetlen memóriahozzáférés | 107 |
| 3.13. JTAG peremfigyelő modul | 108 |
| | |
| 4. Analóg perifériák | 110 |
| 4.1. A D/A átalakítás elve: 4 bites D/A átalakító | 110 |
| 4.2. Meredekség (slope) A/D átalakítók | 111 |
| 4.3. Fokozatos közelítésű A/D átalakítók | 112 |
| 4.3.1 Fokozatos közelítésű átalakítók fejlődése | 115 |
| 4.4. Delta-sigma átalakítók | 118 |
| 4.5. Analóg komparátorok | 119 |
| 4.6. Mechatronika | 120 |
| | |
| 5. Kommunikációs perifériák | 123 |
| 5.1. SPI | 123 |
| 5.1.1 16 bites SPI működési módok | 124 |
| 5.2. I2C | 125 |
| 5.3. Aszinkron soros átvitel | 126 |
| 5.4. UNIO busz | 127 |
| 5.5. LIN | 130 |
| 5.6. CAN | 131 |
| 5.7. USB | 137 |
| 5.7.1. MPLAB Starter Kit PIC24-es mikrovezérlőkhöz | 139 |
| 5.7.2. Keretprogram az USB eszközök kezelésére | 140 |
| 5.8. Data Converter Interface (DCI) | 141 |
| 5.9. Ethernet | 142 |
| 5.9.1. Az Ethernet sikere | 143 |
| 5.9.2. Microchipmegoldások | 144 |
| 5.10. CRC ellenőrző modul | 147 |
| | |
| 6. PIC programfejlesztő eszközök | 149 |
| 6.1. MPLAB | 150 |
| 6.1.1. Fejlesztési lépések – projektek | 151 |
| 6.1.2. MPLAB installálása, törlése | 151 |
| 6.1.3. Projekt létrehozása | 152 |
| 6.1.4. Projekt és munkakörnyezet | 153 |
| 6.1.5. MPLAB – használat | 153 |
| 6.1.6. Szimulátor használata és nyomkövetés | 160 |
| 6.1.7. Stimulus használata | 161 |
| 6.1.8. Aszinkron stimulus | 161 |
| 6.1.9. Szinkron stimulus | 162 |
| 6.1.10. Vizsgálóeszközök a szimulátorban | 165 |
| 6.2. VDI – Visual Device Initializer | 169 |
| 6.3. Application Maestro | 169 |
| 6.4. Kódmodul könyvtár (Code Module Library) | 171 |
| 6.5. MINDI – a Microchip webalapú analóg szimulátora | 172 |
| 6.6. MAPS – Microchip Advanced Product Selector | 172 |

| | |
|---|-----|
| 7. Hardver fejlesztőeszközök PIC mikrovezérlőkhöz | 173 |
| 7.1. ICD2 | 173 |
| 7.1.1. A debugger működése | 173 |
| 7.1.2. Az ICD használata | 174 |
| 7.1.3. ICD2 debugger | 174 |
| 7.1.4. Korlátozások az ICD használatakor | 175 |
| 7.2. ICD3 | 176 |
| 7.3. Real ICE Emulátor | 176 |
| 7.4. PICkit 1 | 177 |
| 7.5. PICkit 2 | 178 |
| 7.6. Explorer 16 Demo Board | 181 |
| 7.7. PIC programozók | 183 |
| | |
| 8. PIC programozás assembler segítségével | 185 |
| 8.1. Utasításkészletek – programozói modell | 185 |
| 8.2. Assembler programozás | 186 |
| 8.3. Programfejlesztés linker használatával | 186 |
| 8.3.1 Map fájl használata | 189 |
| 8.4. Többmodulos programozás | 190 |
| 8.5. Assembler mintapéldák | 192 |
| 8.5.1 minta00.asm | 194 |
| 8.5.2 blink01.asm | 196 |
| 8.5.3 bcnt_02.asm | 198 |
| 8.5.4 rotate_03.asm | 200 |
| 8.5.5 dled_04.asm | 201 |
| 8.5.6 blinkpol_05.asm | 201 |
| 8.5.7 ittabla_06.asm | 202 |
| 8.5.8 adtabla_07.asm | 204 |
| 8.5.9 indir_08.asm | 205 |
| 8.5.10 stmach_09.asm | 206 |
| 8.5.11 preempt_10.asm | 207 |
| 8.5.12 bilkez_11.asm | 208 |
| 8.5.13 eeprom_12.asm | 209 |
| 8.5.14 frekgen_13.asm | 211 |
| 8.5.15 kodzar_14.asm | 212 |
| 8.5.16 adkod_15.asm | 213 |
| 8.5.17 jatek_16.asm | 214 |
| | |
| 9. PIC programozás Basic-ben | 215 |
| 9.1. PICBASIC™ programozás | 215 |
| 9.2. Ajánlott programozási stílus | 217 |
| 9.3. Megoldások Basic programozáshoz | 219 |
| | |
| II. rész | |
| PIC PROGRAMOZÁS C NYELVEN (KOPJÁK JÓZSEF) | |
| Bevezetés | 223 |
| | |
| 10. Az első program | 225 |
| 10.1. A fordító könyvtárszerkezete | 225 |
| 10.2. Az első projekt létrehozása | 225 |

| | |
|---|-----|
| 10.3. „Szia világ!” | 226 |
| 10.3.1 Fordítás | 227 |
| 10.3.2 Futtatás és szimuláció | 230 |
| 10.4. PORTA beállítása | 230 |
| 10.4.1 És a LED-ek felvillannak | 232 |
| 10.5. Azoknak, akik csak az assemblyben bíznak | 234 |
| 11. Egy kis matematika | 236 |
| 11.1. Változók | 236 |
| 11.1.1 Egész számok | 236 |
| 11.1.2 Változók a memóriában | 238 |
| 11.1.3 Lebegőpontos valós számok | 238 |
| 11.2. Operátorok | 239 |
| 11.3. Aritmetikai operátorok | 239 |
| 11.4. Elválasztást végző operátorok | 240 |
| 11.5. Inkrementáló és dekrementáló operátor | 241 |
| 11.6. A megfelelő változótípus kiválasztása | 243 |
| 11.6.1 A legkisebb változótípus | 243 |
| 11.6.2 Egy lépéssel feljebb | 244 |
| 11.6.3 Irány a long! | 245 |
| 11.6.4 És végül a 64 bit | 246 |
| 11.6.5 Elhagyjuk az egész számokat | 246 |
| 11.6.6 Mérési eredmények | 246 |
| 11.6.7 Mikor melyik változótípust érdemes választani? | 248 |
| 12. Körbe-körbe | 249 |
| 12.1. A ciklus fogalma | 249 |
| 12.2. Relációs operátorok | 250 |
| 12.3. Logikai operátorok | 251 |
| 12.4. While ciklus | 251 |
| 12.4.1 Egy kis dinamizmus | 254 |
| 12.4.2 A villogó LED-ek | 255 |
| 12.5. For ciklus | 255 |
| 12.6. Do-while ciklus | 258 |
| 12.7. Break utasítás | 259 |
| 12.8. Continue utasítás | 260 |
| 13. Amikor dönteneni kell | 261 |
| 13.1. If utasítás | 261 |
| 13.2. If-else utasítás | 262 |
| 13.3. Bitoperátorok | 262 |
| 13.4. Rekurzív értékkedő operátorok | 264 |
| 13.4.1 Egy kis „partihangulat” | 264 |
| 13.5. Feltételes operátor | 266 |
| 13.5.1 Bemeneteink is vannak | 267 |
| 13.6. Switch-case szerkezet | 271 |
| 13.6.1 Csak haladóknak | 274 |
| 14. Vissza a változókhöz | 275 |
| 14.1. Típuskonverzió | 275 |
| 14.1.1 Magyar jelölés | 278 |
| 14.2. Mutatók | 278 |

| | |
|---|-----|
| 14.3. Tömbök | 281 |
| 14.3.1 Jöjjenek újból a LED-ek | 282 |
| 14.3.2 Const előtag | 284 |
| 14.3.3 Többdimenziós tömbök | 285 |
| 14.3.4 Karakterláncok | 286 |
| 14.3.5 Készítsünk fényújságot! | 287 |
| 14.4. Mutatóaritmetika | 290 |
| 14.5. Kapcsolat a tömbök és a mutatók között | 290 |
| 14.6. Mutatótömbök | 292 |
| 14.6.1 Karakterláncokat tartalmazó tömbök | 293 |
| 14.7. Mutatók nagysága | 293 |
| 14.8. Sizeof() operátor | 294 |
| 14.9. Dinamikus memória kezelés | 294 |
| 15. Még mindig a változókról | 297 |
| 15.1. Változótípus-definíció typedef segítségével | 297 |
| 15.2. Felsorolástípus | 297 |
| 15.3. Struktúrák | 300 |
| 15.3.1 Struktúrák deklaráció | 300 |
| 15.3.2 Hivatkozás a struktúrák elemeire | 302 |
| 15.3.3 Struktúrák egybeágyazása | 303 |
| 15.3.4 Bitstruktúrák | 305 |
| 15.3.5 Előre deklárált bitek használata | 306 |
| 15.4. Unionok | 306 |
| 15.5. Operátorok kiértékelési sorrendje | 307 |
| 16. És a kép összeáll | 309 |
| 16.1. Függvények | 309 |
| 16.1.1 Függvények definíció | 310 |
| 16.1.2 Függvények deklaráció | 312 |
| 16.1.3 Függvények paraméterátadása | 313 |
| 16.1.4 Rekurzív függvényhívás | 317 |
| 16.1.5 Függvénymutatók | 318 |
| 16.2. Változók láthatósága és élettartama | 318 |
| 16.2.1 Lokális, automatikus változó | 318 |
| 16.2.2 Lokális, statikus változók | 319 |
| 16.2.3 Globális változók | 319 |
| 16.2.4 volatile előtag | 320 |
| 16.3. Több forrásállományból álló programok készítése | 320 |
| 16.3.1 extern előtag | 322 |
| 16.3.2 Statikus globális változók | 323 |
| 16.3.3 Statikus függvények | 323 |
| 16.4. Az előfordítónak szóló utasítások | 323 |
| 16.4.1 Szimbolikus konstansok és makrók definíció | 323 |
| 16.4.2 Előre definiált szimbólumok | 325 |
| 16.4.3 Fejlécállományok betöltése | 325 |
| 16.4.4 Feltételes fordítás utasításai | 325 |
| 16.4.5 Implementációfüggő utasítások | 326 |

| | | |
|----------|--|-----|
| 17. | Amikor több szálon futnak az események..... | 327 |
| 17.1. | Timer1 időzítő/számláló modul használata | 327 |
| 17.2. | Megszakításkezelés | 329 |
| 17.2.1. | A PIC24F család megszakításvezérlője | 329 |
| 17.3. | A megszakításrutin elkészítése | 332 |
| 17.4. | Energiatakarékos üzemmódok használata | 334 |
| 17.5. | Inline assembly | 335 |
| 17.6. | Analóg–digitális átalakító használata | 336 |
| 17.7. | Több megszakításforrás egyidejű használata | 340 |
| 18. | Átjárás a világok között..... | 346 |
| 18.1. | Áttérés a MICROCHIP C18 fordítójára | 346 |
| 18.1.1. | A fordító könyvtárszerkezete | 346 |
| 18.1.2. | Az első projekt elkészítése | 346 |
| 18.1.3. | Konfigurációs bitek beállítása | 348 |
| 18.1.4. | Regiszterek és bitek használata | 348 |
| 18.1.5. | Változótípusok | 348 |
| 18.1.6. | Mutatók | 349 |
| 18.1.7. | Tárolási hely meghatározása | 350 |
| 18.1.8. | Inline assembly | 350 |
| 18.1.9. | Megszakítások használata | 352 |
| 18.1.10. | Kétszintű megszakításrendszer használata | 355 |
| 18.2. | Áttérés a HI-TECH PIC-C fordítójára | 358 |
| 18.2.1. | A fordító könyvtárszerkezete | 358 |
| 18.2.2. | Az első projekt elkészítése | 358 |
| 18.2.3. | Konfigurációs bitek beállítása | 359 |
| 18.2.4. | Regiszterek és bitek használata | 360 |
| 18.2.5. | Változótípusok | 360 |
| 18.2.6. | Tárolási hely meghatározása | 361 |
| 18.2.7. | Mutatók | 361 |
| 18.2.8. | Inline assembly | 362 |
| 18.2.9. | Megszakítások használata | 362 |
| 18.2.10. | Több megszakításforrás használata | 364 |
| 18.3. | Áttérés a MICROCHIP C32 fordítójára | 366 |
| 18.3.1. | A fordító könyvtárszerkezete | 366 |
| 18.3.2. | Az első projekt elkészítése | 366 |
| 18.3.3. | Konfigurációs bitek beállítása | 368 |
| 18.3.4. | Regiszterek és bitek használata | 368 |
| 18.3.5. | Változótípusok és mutatók | 368 |
| 18.3.6. | Inline assembly | 369 |
| 18.3.7. | Megszakítások használata | 370 |
| 18.3.8. | Több megszakításvektor használata | 374 |
| 19. | Mellékletek | 377 |
| 19.1. | A Microchip internet-oldalán közvetlenül elérhető téma gyűjteménye | 377 |
| 19.2. | PIC mikrovezérlőkkel kapcsolatos fogalmak gyűjteménye | 378 |
| 19.3. | Egyéb irodalom | 387 |
| 19.4. | A bináris prefixum | 387 |
| 19.5. | Könyvajánló | 388 |

ELŐSZÓ

A ChipCad Kft. által kiadott, két kiadásban és nagy példányszámban megjelent „PIC mikrovezérlők alkalmazástechnikája” című könyvek népszerűsége indokolja, hogy – mivel az előző kiadás elfogyott – nekivágunk egy újabb, a jelen állapotokat is bemutató, korszerűbb könyv megírásának.

Ennek célja, hogy bemutassuk az utolsó, 2003. évi kiadás óta bekövetkezett, a PIC mikrovezérlőket érintő fejlesztéseket.

A második kiadás óta a Microchip az újabb PIC mikrokontrollereivel teljesen új területekre is belépett. 2004-ben indította a 16 bites dsPIC30 gyártását, amelyeket a kétszeres adatbusz-szélesség mellett DSP jelfeldolgozó képességgel is felruházott, és 2008-ra már 150 típusból álló PIC24 és dsPIC családdá szélesítette. 2007 végén jelentette be a Microchip a harminckét bites mikrokontrollereit, és már 2008 tavaszára megkezdte a PIC32 mikrokontrollerek sorozatyártását. 2008-ra nemcsak a hatmilliárd darabos összes gyártási volument lépte át a cég, de 500-nál is több mikrokontroller-típus gyártásával büszkélkedhet.

Könyünk előző kiadásai Magyarországon hézagpótlónak bizonyultak. Ez volt az első olyan könyv, amely a mikroprocesszoros fejlesztés lépései részletezte: a szükséges hardver és járulékos elemeinek kiválasztásától kezdve az assembleralapú programírás bemutatásáig. Részletesen írtunk az egyes részegységek működését meghatározó státus- és parancsbitekről, a mikrovezérlők működését meghatározó konfigurációs bitekről, azok használatáról.

A harmadik kiadásban nem kevesebb vállalkozunk, mint az olvasó számára kellő támaszt adni a hatalmassá bővülő PIC paletta használatához. A Microchip korán felismerte, hogy a világpiac számára úgy lesz képes sokfajta PIC mikrokontrollert készíteni, ha az azok fejlesztését és tervezését lehetővé tévő fejlesztőszerszámokat is elkészíti. Így a kilencvenes évek közepétől megfigyelhetjük, hogy programozó és hibavadász (debugger) eszközök és szoftverek készítésével segíti a tervezők munkáját. Az egyre összetettebb PIC eszközök programozásához nélkülözhetetlenné vált a magas szintű programozási nyelvek használata és szoftverkönyvtárakba szervezett eljárásokkal való támogatása. Az egyre komplexebb PIC mikrovezérlők által támásztott egyre magasabb szintű kihívásra való válaszként a könyvbe bekerült a C programozási nyelv használatát bemutató rész, amit Kopják József, a BMF Kandó Kálmán Villamosmérnöki Kar tanára írt tanfolyami-főiskolai oktatási tapasztalatainak felhasználásával.

A magas szintű programozási nyelvek használatát ma már nem lehet megkerülni, emiatt kiemelt hangsúlyt fektettünk a C programozási nyelvet elsajátítani szándékozó olvasók igényének kielégítésére. A könyv véges terjedelme miatt az MPLAB C30 fordítóprogrammal dolgoztunk, és a mintapéldákat az univerzális Explorer 16 kísérleti panelre készítettük el. A C programozást bemutató fejezetek felépítésénél a teljesen kezdő és a már C programozói jártasságot szerzett olvasók igényeit is figyelembe vette a szerző, így reményeink szerint minden szinten hasznos ismeretet nyújtunk.

A harmadik kiadásban az időközben óriási méretű terebélyesedő téma miatt kerüljük a korábbi kiadásokban megfogalmazottak közvetlen átemelését, helyette újraszerkesztett formában a korábbi PIC architektúrák jellemzőinek az összefoglalása mellett az új architektúrák és perifériakészletük összehasonlító ismertetését tüztük ki célul.

Bár a második kiadás a mai napig érvényes és fontos ismereteket tartalmaz, nyomtatásban való ismétlés helyett a teljes második kiadást elhelyezzük a könyv DVD-mellékletén, elektronikus formában. A DVD-re kerül a könyv mintaprogramjainak, a Microchip fejlesztőeszközeinek és szoftvereinek a gyűjteménye. Az olvasók azonnal használható szoftvereket

kapnak a könyv mellé, amelyekhez indulásként a Microchip nagy sorozatban gyártott, emiatt olcsó készülékeit javasoljuk megvásárlásra a könyv mintaprogramjainak a kipróbálásához és az ismeretek elmélyítéséhez. Ezek:

- PIC Kit2 Starter Kit
- PIC18F4X20 Starter Kit
- Explorer 16 Kit

Az előző két kiadáshoz hasonlóan a harmadik kiadásban sem vállalkozunk arra, hogy adatlapokat pótolunk, viszont célként tüztük ki, hogy könyvünk olvasói eligazodjanak az angol nyelvű adatlapokban és műszaki specifikációkban. Segítjük a gyártó honlapján való eligazodást és a tervezéshez nélkülözhetetlen adatok megtalálását, azok megértését.

Az előbbiek alapján nyilvánvaló, hogy a könyv első fele a PIC-ek fejlődését bemutató részből, illetve a C nyelvű programozással foglakozó részből áll. Az első rész fejezetei a következő témákat tartalmazzák:

- Az **első fejezet** ismétlés, egy rövid összefoglaló a mikrovezérlők működéséről, részegységeiről, felépítéséről, az utasításvégrehajtás folyamatáról.
- A **második fejezet** témaja a PIC mikrovezérlők felépítésének, fejlődésének a bemutatása. Megvizsgáljuk, hogyan fejlődött a program- és adatmemória, a megszakítás, valamint a legfontosabb rendszerelemek: órajel, RESET, watchdog, alacsony tápfeszültség figyelése a továbbfejlesztések során. Jelentősége miatt részletesen foglalkozunk a tokok fogyasztásának nagymértékű csökkentését lehetővé tevő nanowatt technológia bemutatásával. Külön kitérünk a PIC mikrovezérlők fejlődésének a bemutatására, amely jelenleg már négy különböző típusú nyolc bites adatszélességű, négy 16 bites adatszélességű, valamint egy 32 bites adatszélességű termékvonalból áll. Mivel a 16 bites és 32 bites mikrovezérlőnek magyar nyelvű leírása még nincs, ezért ezeket részletesebben bemutatjuk.
- A **harmadik fejezetben** a digitális perifériákat, illetve azok fejlődését mutatjuk be. Ezek a perifériák: az I/O portok, bemeneti változások kezelése, illetve az érintőkapcsolók bemutatása. Párhuzamos I/O port, számlálók, capture/compare/PWM modulok, forgásérzékelés, LCD vezérlés.
- A **negyedik fejezetben** az analóg egységek bemutatásával folytatódik a perifériák tár-gyalása, bemutatásra kerülnek a különböző típusú A/D átalakítók, fejlődésük, valamint az analóg komparátorok.
- Az **ötödik fejezet** a kommunikációs perifériáké. A szokásos soros megoldások mellett (SPI, UART, I2C) bemutatjuk az újnak tekinthető LIN, CAN, DCI, Ethernet modulokat.
- **Hatodik fejezet:** A Microchip nem csupán a termékpallettát, hanem a programfejlesztő eszközeit is igen intenzíven fejleszti. A legfontosabb fejlesztői környezet továbbra is az ingyenes, a cég honlapjáról letölthető MPLAB IDE (integrált fejlesztői környezet) program maradt. A beépített, jelentősen kibővített szimulátorával a kifejlesztendő alkalmazást tesztelhetjük, ellenőrizhetjük, az alkalmazás áramkörí megalosítását megelőzzen. Az MPLAB-hoz több új hardverelemet illesztettek: emulátorokat, debuggereket, különféle áramkörököt tartalmazó demokártyákat, amelyek mindegyikére jellemző a nem magas ár.
- A mikrovezérlő alapú alkalmazásfejlesztéshez hardvereszközökre: demokártyákra, emulátorokra, debuggerekre, programozókra is szükség van. Ezeket a **hetedik fejezet** foglalja össze.
- A **nyolcadik fejezetben** egy assembler nyelvű példasor segítségével kívánjuk az assembler programozáson alapuló programfejlesztés alapjait elsajátítatni. Ez tizenhat, fokozatosan növekvő nehézségű mintapélda segítségével valósul meg. A feladatokat a **PicKit2** hibakereső/programozó segítségével a **PICF4XK20**-as gyakorlópanel felhasználásával oldjuk meg, eljutva a többmodulos programozás alapjainak a megértéséhez.
- PIC mikrovezérlőket magas szintű nyelven is programozhatunk. A programozói körökből induló tévhittel szemben **BASIC**-ben is nagyon hatékonyan és gyorsan lehet programozni. A **kilencedik fejezet** microEngineering Labs, Inc. cég PicBasic, illetve PicBasic Pro fordítóprogramját mutatja be röviden.

A második rész alapvetően a PIC-ek C-ben történő programozásával foglalkozik. A 8 bites PIC18-as architektúra és az ezt követő 16 és 32 bites adatszélességű PIC mikrovezérlők mindegyike a C programnyelven írt alkalmazások futtatására lett optimalizálva. Ez a rész a C nyelvű programozás alapjainak bemutatása mellett kitér a PIC alkalmazások C nyelven történő programozási megoldásainak a bemutatására is.

- A **tizedik fejezet** végigvezeti a „tanulót” az első program megírásának buktatóin, s közben bemutatja az alapokat: az új projekt létrehozását és konfigurálását, a fordítás és a szimuláció különbségeit, az „A” port inicializálását, valamint a C kód visszafejtését assembly kódra.
- A **tizenegyedik fejezetben** már túl vagyunk az első programunk megírásán, elindulunk a nyelv alaposabb megismerése felé. A C nyelvben, mint a magas szintű nyelvek többségében, a változók és a hozzá tartozó aritmetikai műveletek a nyelv alapkövei. Ennek a fejezetnek az a célja, hogy mi is megismerekjük ezekkel az alapkövekkel: az egész típusú és a lebegőpontos változókkal, az operátorokkal (aritmetikai, elválasztást végző, inkrementáló és dekrementáló), és képesek legyünk a változók teljesítményének összehasonlítására. Szó esik a töréspontos futtatásról is.
- A **tizenkettédik fejezet** mélyebben bevezeti az olvasót az operátorok birodalmába: relációs operátorok, logikai operátorok; megismerkedhet továbbá a ciklusszervezés alapelveivel: while ciklus, for ciklus, do-while ciklus, és természetesen itt kerülnek ismertetésre a ciklusfutást módosító utasítások is.
- A **tizenharmadik fejezet** előtt már megismerkedtünk a ciklusokkal, itt megismerkedünk a feltételes elágazásokkal is. Az eddigi példákban nem kellett feltételek alapján különböző feladatokat végrehajtani. Most megnézzük, hogyan lehet megoldani azt, ha valamilyen feltétel alapján különböző részeket kell a programunknak végrehajtania.
- A **tizennegyedik fejezet** a mélyebb ismeretek megszerzésének területe. A 11. fejezetben már foglalkoztunk a változókkal, de akkor csak érintőlegesen tekintettük át őket. Most egy kicsit mélyebben a dolgok mögé nézünk, és megismerkedünk a C nyelv változói és a hozzájuk kapcsolódó operátorok által nyújtott lehetőségekkel. Természetesen ez a fejezet sem mentes a mintapéldáktól. A tömbök segítségével a ledsorra egy újabb programot készítünk.
- A **tizenötödik fejezet** a C nyelv összetett adatstruktúrák kezelésével kapcsolatos szolgáltatásait tekinti át. Megismerkedünk a typedef, enum, struct és a union kifejezéssel. Megismerkedünk a bitstruktúrák kialakításának módszerével és a mikrokontrollerek speciális funkcióregisztereit bitenkénti elérésének lehetőségével.
- A **tizenhatodik fejezet** bemutatja a C nyelv egyik nagyon fontos elemét, a függvényeket és azok használatát. Ezek után megismerkedünk a több modulból álló programok elkezdtésekének menetével. A fejezet végén az előfordítónak szóló utasításokkal foglalkozunk. Ezzel a fejezettel zárul az általános C nyelv bemutatása, a könyv hátralévő két fejezete már a különböző architektúrákhöz tartozó fordítók hardverspecifikus alkalmazásaival foglalkozik.
- A **tizenhetedik fejezet** már a gyakorlat világába vezeti az olvasót: most, hogy megismerkedtünk a C nyelv által nyújtott lehetőségekkel, elkészítjük az első, a mikrokontroller belső perifériái által nyújtott lehetőségeket is kihasználó programunkat. Most sem hagyjuk el a jól bevált ledsorunkat, egy újabb futófényprogramot hozunk létre. Először megismerkedünk a **Timer1** időzítő modul beállításával és használatával, majd a következő lépésben kiegészítjük a programunkat egy megszakításrutinnal. A megszakításrutin elkészítése után a programunkat energiatakarékkossá alakítjuk át, majd legvégi állíthatóvá tesszük a futófénynünk sebességét a fejlesztőpanelen található potenciometré segítségével.
- A **tizennyolcadik fejezet** rövid útmutatót nyújt azok számára, akik a Microchip 16 bites PIC mikrokontrollereitől eltérő, különböző bitszélességű architektúráakra szeretnének C nyelvű programokat készíteni. A fejezet három, különböző architektúrához készült fordító használatát mutatja be. A fejezet első harmada a Microchip PIC18 mikrokontroller-

családhoz készült fordító bemutatásával foglalkozik. A fejezet középső része a Microchip PIC10/12/16 mikrokontroller-családhoz készült Hi-Tech C fordítót ismerteti. Végül a Microchip PIC32 mikrokontroller-családhoz készült fordító bemutatásával zárul a fejezet. minden egyes fordító ismertetése a fordító könyvtárszerkezetének bemutatásával kezdődik, majd az első projekt létrehozásának lépéseivel folytatódik. A fejezet ismerteti az egyes fordítók által használt változótípusokat, a konfigurációs bitek beállításának menterét, a regiszterek használatát és a megszakításrutin készítését.

A könyv végén lévő **Melléklet** a PIC mikrovezérlőkkel kapcsolatos kiegészítő információkat tartalmazza. Mivel a PIC mikrovezérlők fejlesztésekor a perifériák nem változtak jelentősen, ezért csak a működésüket kellett leírni, a használathoz szükséges konkrét információkat már az adatlapokról lehet begyűjteni.

A véges méretek miatt dönteni kellett arról, hogy milyen részek milyen részletekkel kerüljenek bele a könyvbe. Mivel a könyvhöz DVD-mellékletet csatoltunk, ezért számos információt azon is el lehetett helyezni.

Jelenleg számos PIC család létezik egy időben, amelyek eltérő adatszélességgel (8, 16, 32 bit) és utasításszélességgel (12, 14, 16, 24, 32 bit) rendelkeznek. Sajnos a Microchip jelölésrendszere és az említett jellemzők között nem egyértelmű a kapcsolat. Ezért a könyvben az egyes architektúrák jelölésére egységesen az

ADATSZÉLESSÉG / UTASÍTÁSSZÉLESSÉG

jelölési rendszert fogjuk használni azokon a helyeken, ahol valamelyik családra jellemző információkat közlünk.

Ennek megfelelően a következő táblázat összefoglalja a különböző architektúrákhoz kapcsolódó egyértelmű jelöléseket, amelyeket a könyvben alkalmazunk. Ezekkel fogjuk jelezni, hogy az egy-egy szövegrészben szereplő magyarázat melyik családhoz kapcsolódik, segítve ezzel a könyvbeli gyorsabb tájékozódást.

- 8/12 PIC10F/PIC12F 5 /PIC16F 5 - 8 bites adat-/12 bites utasításszélességű csoport – Baseline Flash Mikrokontroller
- 8/14 PIC12F/PIC16F – 8 bites adat-/14 bites utasításszélességű csoport – Mid range architecture – Középszintű termékvonal
- e8/14 PIC16F 1XXX 8 bites adat-/14 bites utasításszélességű csoport – Továbbfejlesztett középszintű termékvonal – Enhanced PIC16
- 8/16 PIC18F - 8 bites adat/16 bites utasításszélességű csoport – High performance microcontrollers – Nagyteljesítményű mikrovezérlök
- 16/24 PIC24F/PIC24H/dsPIC30/dsPIC33 – 16 bites adat/24 bites utasításszélességű mikrovezérlök
- 32/32 PIC32 – 32 bites adat/32 bites utasításszélességű mikrovezérlök

I. RÉSZ

PIC MIKROVEZÉRLŐK ALKALMAZÁSTECHNIKÁJA

KÓNYA LÁSZLÓ

1. ISMÉTLÉS

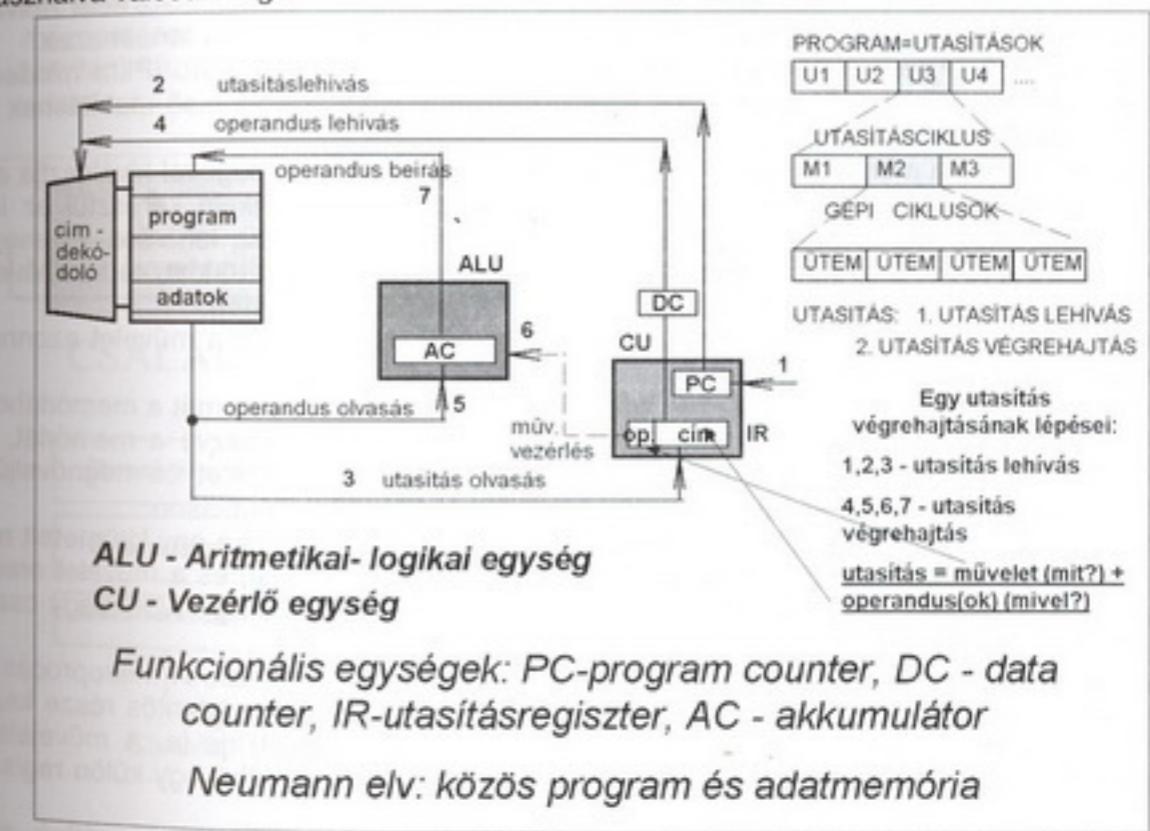
A következőkben röviden összefoglaljuk a mikroprocesszorokkal kapcsolatos legfontosabb ismereteket.

1.1. A MIKROPROCESSZOR MŰKÖDÉSE

A mikroprocesszor működésekor minden utasítás és minden adat bitcsoportokkal van kódolva. Azt az eljárást, amikor egy bitcsoporthoz valamelyen információegységet rendelünk, kódolásnak nevezzük.

Ha az utasításokat egymás után akarjuk végrehajtani, akkor egy memóriából kell sorban egymás után ezeket kiolvasni, és az ALU segítségével feldolgozni. Ennek végrehajtását a vezérlőegységnek nevezett elem (*CU = Control Unit*) végzi. A működés az 1.1. ábra alapján érhető meg.

Az utasítások és az adatok a memóriában vannak. A vezérlés maga néhány regisztert használva valósul meg:



1.1. ábra
A mikroprocesszor működése

Programszámláló (PC). A programszámláló minden utasításnak a címét tartalmazza, amit a mikroprocesszornak majd a tárból le kell hívnia.

Utasításregiszter (IR). Amikor a mikroprocesszor egy utasítást hív le a tárolóból, az utasításnak megfelelő bináris kód az utasításregiszterbe kerül. Az utasításdekódoló egység ezt feldolgozva állítja elő az utasítás tényleges végrehajtásához szükséges logikai vezérlőjeleket.

Az adattároló címmutató regisztere (DC). Ez a regiszter tárolja annak az adatnak a

címét, amelyet a CPU kiolvas vagy beír a memóriába. A címbuszra a címek két forrásból kerülhetnek: a programszámlálóból és a tároló címregiszterből.

Akkumulátor (AC). Az aritmetikai-logikai egységben lévő akkumulátorregiszter a legtöbb műveletben részt vevő, kitüntetett szerepű regiszter. Az utasítások nagy része kapcsolódik ehhez a regiszterhez.

Általános segédregiszterek. Ezeket a CPU műveleteihez szükséges adatok átmennetet tárolására használják. Minél nagyobb ezek száma, annál kevesebbet kell a CPU-nak a külső memóriához fordulni, azaz a mikroprocesszor teljesítményét növeli.

Státusregiszter. A státusregiszter bitjei tartalmazzák a CPU által végrehajtott műveletekre vonatkozó információkat. Flageknek (kiejtése: fleg) vagy magyarul jelzöbriteknek is nevezzük, és minden egyik valamelyen speciális mikroprocesszor-állapotot jelez. A legfontosabbak az átvitelbit (ha egy összeadás eredménye 9 bites, akkor ezt az átvitel- [CY=CarrY] bit értéke jelzi), az eredmény negatív voltát jelző bit stb.

A regiszterek közül a legfontosabb a PC, amit szoktunk utasításmutatónak is hívni: ennek tartalma mondja meg, hogy melyik a memóriából következőnek kiolvasandó utasítás címe. A kiolvasott utasítás a feldolgozáshoz az IR nevű utasításregiszterbe kerül. Az adat kiolvasásához az adatcímét a PC-hez hasonló módon a DC adatmutató regiszter tartalmazza.

Először a PC-be be kell tölteni az elsőnek végrehajtandó utasítás címét. (Pl. a minden alaphelyzetbe hozó RESET folyamat ezt a regisztert nullázza – így az első utasításnak a nulla című tárolóhelyen kell lennie.)

Ez a cím kijutva a memóriát címző vonalakra, a címdekódoló segítségével kiválasztja az utasítást tartalmazó regisztert, és a tartalmát a memória adatvezetékein keresztül az IR utasításregiszterbe írja. Ezzel megtörtént a memóriában lévő utasítás lehívása, és megkezdődhet a végrehajtása. Mint tudjuk, egy utasítás bináris formában tartalmazza a műveleti kódot, és az operandust (illetve a tényleges operandus memóriabeli címét).

Ha az utasításban ténylegesen egy adat van operandusként, akkor a művelet azonnal végrehajtható az ALU segítségével.

Ha az utasítás adatrésze a tényleges adat címét tartalmazza, akkor ismét a memóriához kell fordulni: a DC regiszterbe be kell tölteni ezt a címet, és megcímézve a memóriát, a tényleges adatot olvassuk ki, és az ALU-ba juttatva elvégezzük a műveletet, és megnöveljük a PC értékét egyelőre, hogy a következőként végrehajtandó utasításra mutasson.

Mivel egy művelet általában két operandust igényel, ezért az ALU-ban egy kitüntetett regisztert (AC) használunk, ami minden műveletben tartalmazza az egyik operandust, és a művelet eredménye is ott képződik. A kitüntetett regiszter neve akkumulátor (AC vagy ACC vagy csak egyszerűen A) vagy a PIC-eknél working regiszter (W).

Az egyoperandusos műveletek legtöbbje az akkumulátorra vonatkozik. A mikroprocesszor által végrehajtható utasítások halmazának, az utasításkészletnek jelentős része kapcsolódik az ALU-hoz, tulajdonképpen annak funkcionális működését írja le. A műveletek eredményére utaló ún. **státusbitek** (túlcordulás, az eredmény nulla stb.) egy külön regiszterbe, a **státusregiszterbe** kerülnek.

Ha az ALU és CU egységeket a kiegészítő áramköreikkal egy tokba integráljuk, akkor beszélünk mikroprocesszorról.

1.2. MIKROSZÁMÍTÓGÉPEK, MIKROKONTROLEREK

Mikroszámítógépnek vagy mikrogépnek az integrált áramkörökkel felépített programozható rendszert nevezünk. A mikroprocesszor típusától függetlenül a mikroprocesszor a következő főbb részekkel kell rendelkeznie:

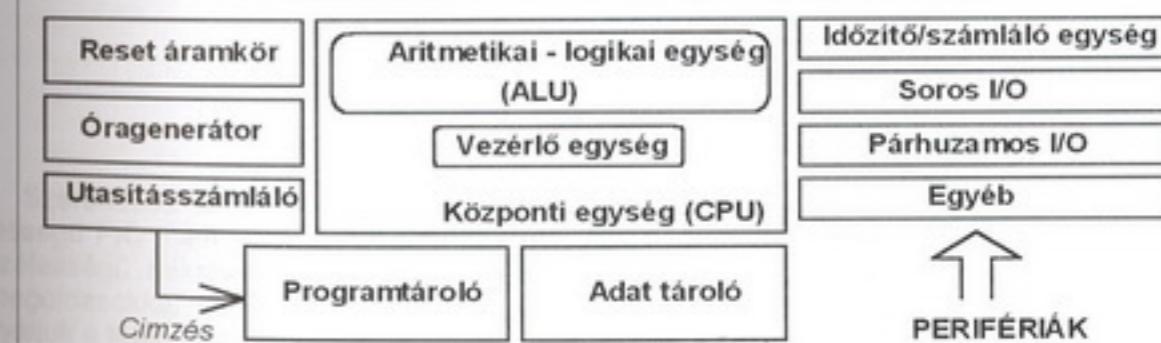
1. fejezet: Ismétlés

- A központi egység** (ez a mikroprocesszor). A mikroprocesszor a mikrogép központi egysége. Az eredeti angol elnevezése alapján szokták CPU-nak is mondani. Jellemzői alapvetően meghatározzák a teljes mikrogép működését. Funkciót a következőképpen foglalhatjuk össze:
 - A mikrogép minden eleme számára biztosítja az időzítő- és vezérlőjeleket.
 - Elvégzi az adatok és utasítások tárból/tárba való mozgatását.
 - Dekódolja és végrehajtja az utasításokat.
 - Irányítja a be/kimeneti egységekkel kapcsolatos adatforgalmat.
- A programtár és adattár** (ez a memória, ami lehet közös – pl. PC-nél – vagy különálló), az egyikben a működtető program, a másikban a program működése során létrejövő vagy a külvilág felől érkező adatok vannak tárolva.
- A be- és kiviteli egység** (Input/Output, röviden I/O vagy Be/Ki egység – ezek neve: periféria), amely a külvilággal tartja a kapcsolatot.
- A egységek közötti információ- és adatáramlást biztosító vonalak.** Ez a kommunikáció az ún. síneken vagy más néven **busz** vonalakon keresztül valósul meg. Mi is az a sín? A sín azonos funkciójú vezetékek csoportja. Erre a vezetékkötégre minden egység párhuzamosan kapcsolódik azért, hogy egymáshoz információt tudjon továbbítani. Például a mikroprocesszor, a memória, valamint a be- és kimeneti áramkörök, az egymás (és a külvilág) közötti adatcsere érdekében ugyanarra a nyolc vezetékre kapcsolódnak: ez az adatsín.

Mikrokontrollerek felépítése

Mikrokontroller vagy mikrovezérlő: mikroszámítógép egy tokban

CSALÁDOKNÁL KÖZÖS



A mikrokontroller belső felépítése

A PERIFÉRIÁK HASZNÁLATA CSÖKKENTI A CPU TERHELÉSÉT (a perifériakezelés hardveres megoldása miatt)

1.2. ábra
A mikrokontrollerek felépítése

A mikroszámítógépek fejlődésére a méretek csökkentése a jellemző. Kezdetben az egyes funkcionális egységeket, a CPU-t, memóriát, I/O-t külön nyomtatott áramköri kártyákon, kártyamodulként alakították ki. Később az integrálási technika fejlődése lehetővé tette az összes egység egy kártyára történő elhelyezését. Ilyen például a PC alaplapja. Ezeket hívjuk egykártyás mikrogépeknek vagy SBC-nek (*SBC – Single Board Computer*).

Az integrálási technika további fejlődése lehetővé tette, hogy egy mikroszámítógép minden részegységét egy lapkára integrálják: ezek az **egytokos mikroszámítógépek**, más néven **mikrokontrollerek**, magyarul **mikrovezérlők**.

Azért, hogy különféle feladatoknak meg tudjon felelni egy mikrokontroller, ezeket a **család** alapján különböző kialakításban gyártják. (Család: CPU azonos, az adat- és programmemória mérete, a perifériák száma, fajtája különbözik.)

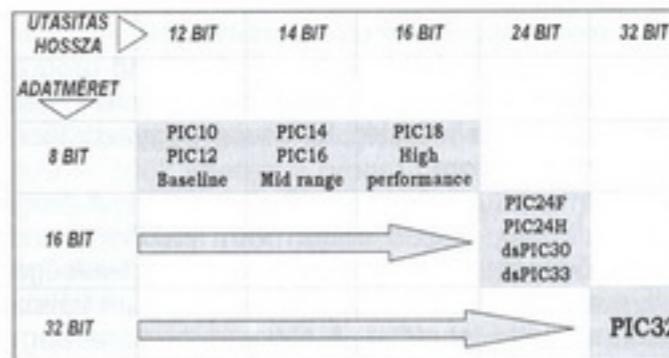
2. A PIC MIKROVEZÉRLŐK FELÉPÍTÉSE, FEJLÖDÉSE

A következőkben összefoglaljuk a PIC mikrovezérlők felépítését, ami az ismétlésben már bemutatott részekből áll:

- CPU
- Programmemória
- Adatmemória
- Perifériák

A PIC mikrovezérlőket két módon osztályozhatjuk:

- Az egyik szempont a mikrovezérlő utasításainak a bitekben mért szélessége: ez először 12 bit szélességű volt, majd 14, 16, 24, illetve 32 bit szélességűre változott. Az utasítás-szélesség pedig meghatározza az utasítások bonyolultságát, a közvetlenül kezelhető adat- és programmemória nagyságát.
- A másik osztályozási lehetőség az adatok szélessége: ez 8, 16 és 32 bit lehet. Ez – mivel az egy utasítással kezelhető adatbitek számát határozza meg –, alapvetően meghatározza a számítási teljesítményt.

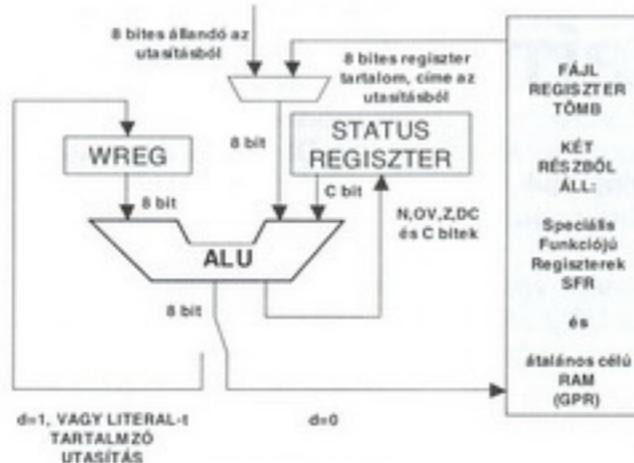


2.1. ábra
PIC mikrovezérlők

Szerencsés helyzetben vagyunk, mert a folyamatos fejlesztések a 8 és 16 bit adatszélességű PIC mikrovezérlőknél a felépítést alapvetően nem változtatták meg. A 32 bit adatszélességű mikrovezérlő processzormagja már nem Microchip-fejlesztés, ezért ott más megoldásokkal találkozhatunk. Ezért a következőkben ennek a figyelembevételével tárgyaljuk a felépítést.

2.1. ARITMETIKAI-LOGIKAI EGYSÉG (ALU)

Az utasítások általában a legtöbb művelethez két operandust használnak, amit érdemes úgy megoldani, hogy az egyik operandust az ALU-ban található speciális, kitüntetett regiszter tartalmazza, aminek a neve akkumulátor (A) vagy a Microchip szóhasználatában working regiszter (W). A másik operandus az utasításban szereplő című adatmemória-rekesz tartalma. Ezzel és az akkumulátor tartalmával végezzük el a műveletet. Célszerűen az eredmény is az egyik regiszterben keletkezik, amit a továbbiakban felhasználhatunk. Az ALU és a működés vezérlését végző, önállóan meg nem jelenő vezérlőegység együttesen alkotja a CPU egységet. Az ALU működése közben számos bitműveletet végez, ezért az utasítás-készletet ennek kezelésére is fel kell készíteni.



2.2. ábra
8 bites ALU

2.2. PROGRAMMEMÓRIA

A programmemória egyenként megcímzhető, utasításokat tartalmazó tároló regiszterek halmaza. Mérete határozza meg az adott típusba elhelyezhető program nagyságát. Az éppen végrehajtandó utasítás címe az utasításszámlálóban van. Az utasításszámláló bitszáma határozza meg a memória maximális méretét. Ez csupán egy lehetőség, általában fizikailag kisebb memóriaterület áll rendelkezésre.

A programmemória tartalmazza a mikrovezérlő programját. A programmemoriában lévő utasítás szélessége meghatározó, hiszen ebben van kódolva a művelet és az operandus(ok). Ezért a PIC-ek fejlődését, az utasítások növekvő szélességével is jellemzőjük: kezdetben ez 12 bit volt, majd 14, 16, 24 és 32 bit szélességüre változott, ami egyre összetettebb utasítások kialakítását tette lehetővé. A programmemória nagysága („hosszúsága”) is változik, attól függően, hogy mekkora programot tudunk elhelyezni benne.

2.2.1. Programmemória-típusok

A programmemória legfontosabb jellemzője az, hogy a benne lévő tartalom esetleg csak az újraprogramozásával módosítható, ami fejlesztéskor gyakran, üzemszervi működés során ritkán (vagy soha) nem módosul. Ezért az első PIC processzorok minden típusa két formában volt elérhető:

- Kvarcüveg ablakos formában, aminek beprogramozott tartalmát ultraibolya fénnnyel törölhetjük (EPROM) – ez jó a programfejlesztéskor.
- Egyszer programozható kivitelben. Ilyenkor a tok csak egyszer programozható, és tartalma nem változtatható.

Gondolható, hogy a rendszerfejlesztés ilyen ablakos eszközzel nagyon nehézkes volt, a beírt programot a mikrovezérlőt a foglalatába helyezve próbálták ki, és az esetleges hibás működés kiderítéséhez sokszor komoly nyomozómunkát kellett végezni. Igaz – mivel az ultraibolya fénnnyel való törlés elég időigényes volt –, elegendő idő állt rendelkezésre a „Hol hibáztam?” kérdés megválaszolására.

Az ilyen fejlesztés másik problémája az egyszer programozható eszközök programozásánál jelent meg. A programozó élete egy rettegés volt: „Ok. Kipróbáltam, műköött, de minden lehetséges eseményt teszteltem?” Ez különösen akkor vált érdekessé, ha vagy nagy darabszámu termékbe került a tok, vagy esetleg néhány elkészült termék mondjuk Szibériában landolt...

Az első, elektromosan újraprogramozható EEPROM-ot tartalmazó tok a legendás **PIC16C84**-es típus volt. Aki tehette, ezt használta, mert ennek újraprogramozása nagyon egyszerű és gyors volt. Termékbe beépítve már nem okozott nagy gondot az esetleges újraprogramozás. (Persze a szibériai probléma továbbra is megmaradt...).

Igazi változást jelentett, mikor a Microchip bevezette a **PIC16F87x** családot. Ez már flash EEPROM programmemoriával rendelkezett, és megjelent a hibakereső (debug) lehetőség. (A ma már jól ismert flash memóriára jellemző, hogy programozása nem bájtonként, hanem több szomszédos bájtból álló blokkokban történik.)

Az ilyen tokkal történő fejlesztés során, kiegészítve az ICD1 hibakereső-programozó eszközzel, lehetővé vált a program futásának nyomon követése. Megállíthatjuk a program futását adott programszámláló értéknél (ez a töréspont), és a megállás után a regiszterek tartalmát áttölthük az MPLAB IDE környezetbe, és megvizsgálhatjuk tartalmukat, pontosan úgy, mint szimulációnál.

Mivel a család változó lábszámú és változó programmemória-méretű tagokat tartalmazott, különféle perifériákat tartalmazó kiépítésben állt rendelkezésre, ezért lehetővé vált a **PIC16** összes családelemének fejlesztése. Az addig több lábon történő tokprogramozást felváltotta a két jelvezetéket használó megoldás. A **PIC16F87x** eszközökön kifejlesztett, tesztelt programokat könnyű volt az egyszer programozható céleszközbe átvinni.

A **PIC18**-as család bevezetésétől kezdve többségen már csak EEPROM-alapú programmemoriát tartalmazó eszközök vannak; az új típusoknál ez természetes, míg a továbbra is népszerű és sok termékben megjelenő régebbi típusokat (**PIC12Cxx**, **PIC16C5x**, **PIC16Cxx**) is átterveztek flash memóriásra.

Ezek után nézzük meg, hogy milyen programmemória-típusokat biztosít a Microchip a fejlesztések optimalizálása érdekében.

- **Flash (elektromosan újraprogramozható) memória:** a jelenleg legnépszerűbb megoldás. Flash PIC mikrovezérlők esetén a programmemoriát törölhetjük és újraprogramozhatjuk. Ez számos előnyvel jár: a felhasználók a gyártási folyamat végén vagy akár a helyszínen programozhatják be a termékeket. Lehetséges a kód utólagos módosítása, paraméterek átírása. Fontos tudni, hogy egy memóriacella programozási ideje néhány ms. Ezért a memóriacellákat nem egyenként, hanem blokkonként írják, illetve törlik. Adott típusnál a blokkok mérete fontos katalógusadat. Megjelenik az önprogramzási lehetőség (*self programming*). Ez azt jelenti, hogy távolról, egy kommunikációs kapcsolaton keresztül frissítjük a flash programmemóriát. Ennek megvalósításához a tervező egyszerű letöltőprogramot helyez el a memória egy kódvédett területére. Egy biztonságos internetes, rádiós, infravörös fényt használó vagy telefonvonalon keresztül belépünk a programba a tok USART, I²C™ vagy SPI™ soros interfészén keresztül. A letöltő programmal ezután a vonalon érkező programadatokkal újraírhatjuk a programmemóriát. Jelenleg a Microchip két flash memóriatípust használ:

- **Továbbfejlesztett enhanced flash:** 40 éves adatmegőrzési idő, önprogramozhatóság 2 V–5,5 V között, ICSP (*In-Circuit Serial Programming*) 5 V vagy 12 V-on; adat EEPROM 1 millió törlés/írási cikllussal.
- **Szabványos flash:** 10 000 törlés/írási ciklus, 40 éves adatmegőrzési idő, ICSP képesség csak 12 V-on.

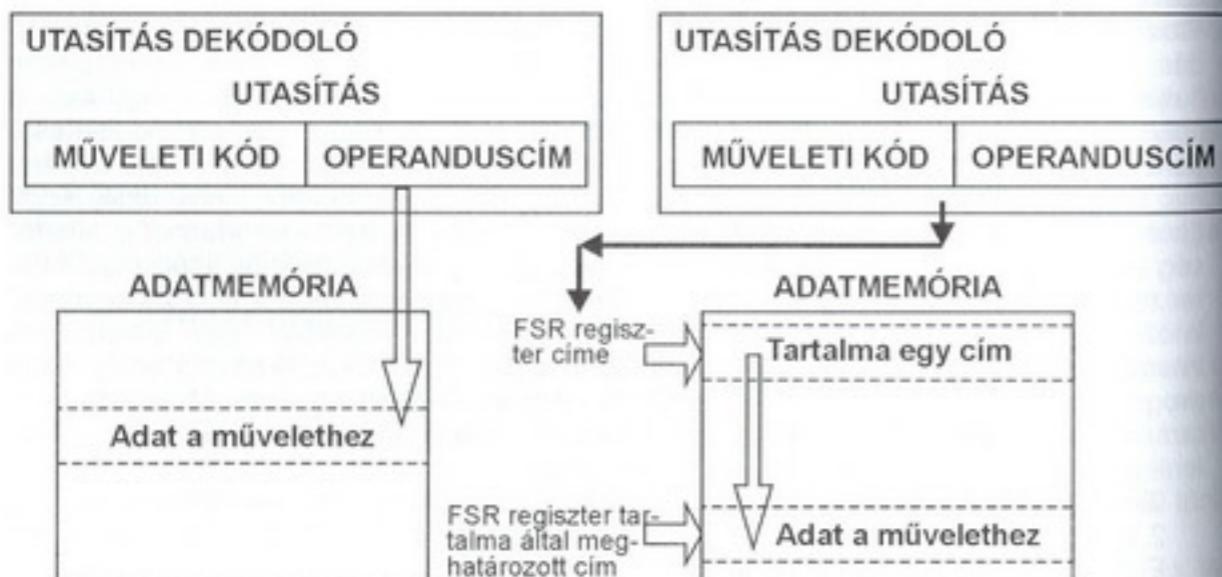
- **Egyszer programozható (ONE TIME PROGRAMMING – OTP) memória:** az OTP memóriájú PIC mikrovezérlőket beírt program nélkül gyártják, és szállítják a felhasználókhöz. Ez azoknak hasznos, akik gyorsan akarják a gyártmányukat piacra dobni, és a gyakran kell a gyártmányt tokcserével frissíteni. Az ilyen típusoknál lehetséges a mikrovezérlők beforrasztott állapotában történő programozása: áramkörben történő programozás. Ez a megoldás nagyon rugalmas, csökkenti a fejlesztés idejét, a gyártás hatékonyságát növeli, és lerövidíti a gyártmány piacon való megjelenését. Lehetséges a gyártott rendszer kalibrálása a gyártás alatt, és egyedi azonosítókódokat is adhatunk a termékeknek a gyártás során. Programozása minden I/O lábat igényel a legtöbb eszköznél.

- Gyorsan gyártható (QUICK-TURN PROGRAMMING - QTP):** van lehetőség a tokok felprogramozására a tokok gyártásának egy adott fázisában. Ez azoknak a megrende-löknek ideális, akik nem saját maguk akarják a viszonylag nagy mennyiségi tokot be-programozni, valamint a beírandon kód már biztosan hibamentes és nem változik.
- Sorszámmal ellátott tokokat eredményező gyors gyártás (SERIALIZED QUICK-TURN PROGRAMMING - SQTPSM):** ennél a megoldásnál a QTP megoldás mellett még minden programozott tok saját egyedi azonosítót, sorszámot is kap.
- Maszkolt (MASKED) ROM:** nagy mennyiségi tok azonos programmal történő használatához a programból gyártási maszkot generálva állítják elő a gyártás során memoriába kerülő programot.

2.3. ADATMEMÓRIA

Az adatmemória az adatszélességgel azonos bitszámú (8, 16 vagy 32 bit) megcímezhető regiszterekből épül fel. A Microchip erre a memóriára a **fájlregiszterek** kifejezést használja. Címzése két módon lehetséges:

- Közvetlen (direkt) címzés:** az utasításban van az a cím, amelynek tartalmával kell elvégezni a műveletet. Szimbolikusan: *MOV REG,W - REG tartalma W-be kerül*.
- Közvetett (indirekt) címzés:** ilyenkor az utasításban egy címhez hasonló bitcsoport szerepel. Az utasításdekomponáló ezt érzékelve egy adott regiszter, az FSR (=File Select Register) tartalmát tekinti címnak, és az azon a címen található regiszter tartalmával végzi el a műveletet. Szimbolikusan: *MOV [REG],W - REG-ben lévő címen található regiszter tartalma kerül W-be*.



Művelet [OPERANDUSCÍM] [A] jelentése: a tartalma Művelet [[FSR]]

2.3. ábra
Közvetlen és közvetett címzés

Az adatmemória két részre oszlik:

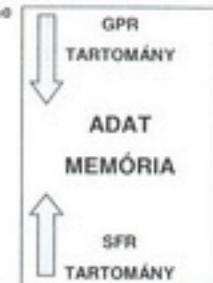
- a felhasználó által általánosan felhasználható adatregiszterekre, angol elvezetése: **General Purpose Registers – GPR = általános célú regiszterek**, illetve
- a mikrovezérlő működéséhez és a perifériák kezeléséhez szükséges regiszterekre. Ennek angol neve: **Special Function Registers – SFR = speciális funkciójú regiszterek**.

2. fejezet: A PIC mikrovezérlők felépítése, fejlődése

8/12 8/14 Mivel fizikailag ez a két adatmemória-tartomány nem ^{CIM=0} különül el, ezért valamilyen módon szét kell osztani. A PIC18-as család megjelenéséig a tervezők úgy döntötték, hogy a 0-s adatmemória-címtől helyezik el az SFR tartományt, majd felette helyezkedik el a GPR tartomány.

8/16 A PIC18-as családban már az adatmemória két végén helyezkednek el ezek az adatmemória-részletek. A GPR terület a 0-s címen kezdődik, míg az SFR memória az adatmemória végén.

Illusztrációul a 12 és 14 bites PIC-eknél alkalmazott SFR kiosztás egy részletét láthatjuk a 2.4. ábrán. Ez a megoldás használható, de a perifériák számának – és azok kezelő regisztereinek – növekedése miatt egyre jobban eltolódik a GPR tartomány kezdete.



Address

| | | | |
|------------------|-----|--------|-----|
| Indirect address | 00h | PORTC | 07h |
| TMR0 | 01h | PORTD | 08h |
| PCL | 02h | PORTE | 09h |
| STATUS | 03h | PCLATH | 0Ah |
| FSR | 04h | INTCON | 0Bh |
| PORTA | 05h | PIR1 | 0Bh |
| PORTB | 06h | | 0Ch |

2.4. ábra
Adatmemória-részlet

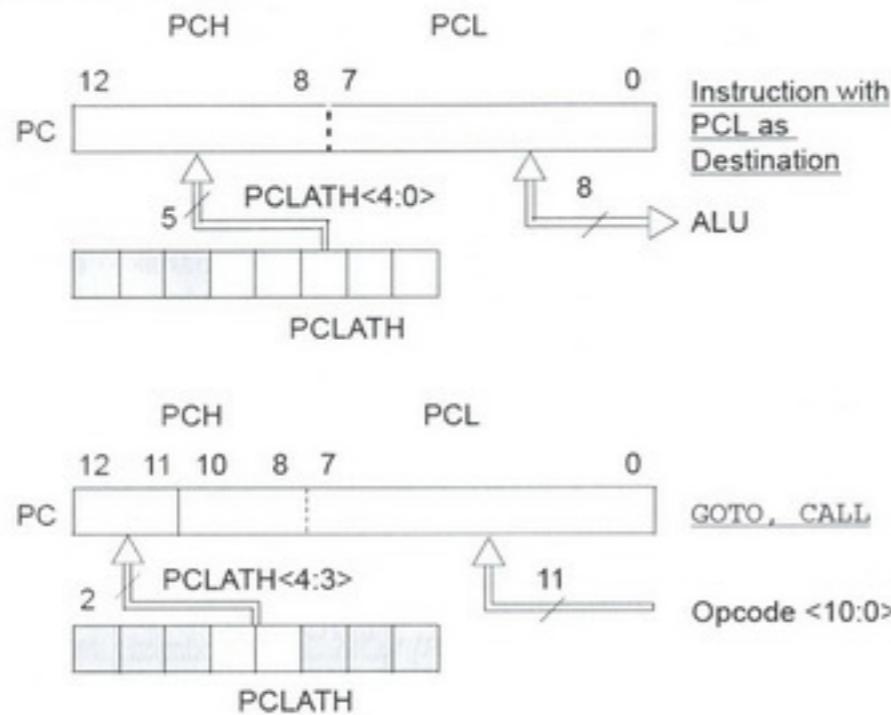
2.4. A PROGRAM- ÉS AZ ADATMEMÓRIA KAPCSOLATA

Sok esetben vannak olyan adataink, amelyek értékei nem változnak, például egy hőmérő kalibrációs táblázata: mit mérünk, és mi a tényleges hőmérséklet, amit ki kell jelezni. Ezek adatként viselkednek, úgy is használjuk ezeket, de változatlanságuk miatt célszerű lenne nem változó, tartalmát megőrző memóriában tárolni. Három megvalósítás is lehetséges:

- 1) Kialakítunk egy **adat EEPROM memóriát**, és az adatokat itt tároljuk. Ehhez szükséges, hogy a kontroller rendelkezzen ilyen memóriával, és megfelelő méretű legyen. A tok felprogramozásakor visszük be ide azt a tartalmat, amit adatként használunk. Ez a megoldás a megadott feltételek mellett használható, de az elhelyezhető adatmennyiség erősen korlátozott, és ilyen esetben nem használjuk ki azt, hogy ez a memória programból írható is.
- 2) Megoldjuk, hogy az **adatokat a programmemóriában tároljuk el**, és gondoskodunk az így elhelyezett adatok olvasásáról.

8/14 Első megoldásként egy olyan utasítást definiáltak, aminek véghajtásakor az utasításban operandusként szereplő állandó adat (konstans vagy más néven literál) a W munkaregiszterbe íródik, és így már adatként használható. Egyúttal az utasítás a szubrutinokat befejező és a szubrutin hívási helye utáni utasításra visszaugró RETURN utasítást is megvalósít. Az utasítás neve: **RETLW (Return with Literal in W)**. Ez a megoldás az adatsorok tárolására való, illetve a segítségével megadhatunk a szubrutin befejezésekor egy hibakódot is, ami jelzi a szubrutin sikereségét. Ezt az utasítást kompatibilitási okokból mind a 8, mind a 16 bites PIC mikrovezérlők utasításkészlete tartalmazza. Az ilyen utasítás használatához biztosítanunk kell, hogy az utasításszámítálóból be tudjuk írni az ezt az utasítást tároló memóriahely címét.

A 2.5. ábra szemlélteti ezt a megoldást, míg az ezt követő programrészlet a programban történő megvalósítást mutatja.



2.5. ábra
PIC16 táblakezelés

```

MOVWF LOW TABLE
ADDWF OFFSET,F
MOVLW HIGH TABLE
BTFS C STATUS,C
ADDLW 1
MOVWF PCLATH
MOVF OFFSET,W
CALL TABLE
.

ORG 0X9FD
TABLE:
MOVWF PCL,F
RETLW 'A'
RETLW 'B'
RETLW 'C'

; W-BE TÖLTJÜK A CÍM ALSÓ 8 BITJÉT
; HOZZÁADJUK AZ ELTOLÁST OFFSET:=OFFSET+W
; W-BE TÖLTJÜK A CÍM FELSŐ 5 BITJÉT
; HA AZ ÖSSZEADÁS EREDMÉNYE
; TÚLCSORDULT, ÁTLEPTÜK A LAPHATÁRT
; HA IGEN FELSŐ CÍMET 1-GYEL NÖVELNI KELL
; A FELSŐ CÍM PCLAT-BA TÖLTÉSE
; SZÁMÍTOTT ELTOLÁS W-BE ÍRÁSA
; TÁBLAELEM KIOLVASÁS MEGHÍVÁSA

```

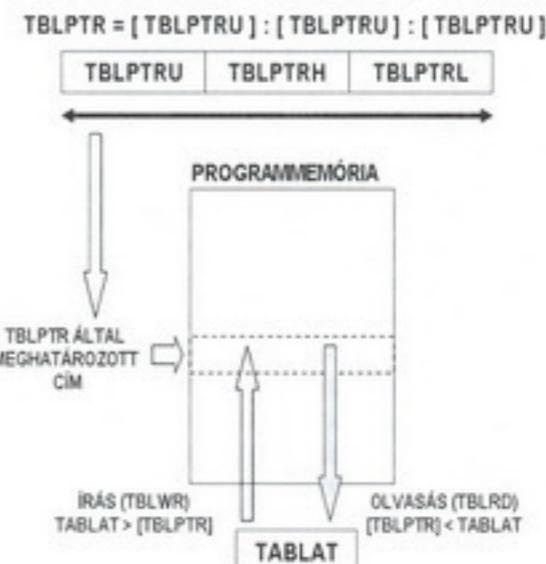
```

; SZÁMÍTOTT ELTOLÁS PCL-BE TÖLTÉSE
; RETURN ASCII CHAR A
; RETURN ASCII CHAR B
; RETURN ASCII CHAR C

```

8/16 A másik, PIC18-as családnál megjelenő megoldás: a programmemória egy bájtját megcímző táblamutató bevezetése. A programmemória egybájtos memóriahelyét a TBLPTR-nek nevezett 22 bit szélességű regiszter tartalma cími. Innen a megcímzett memóriabájt olvasásakor az adat a TABLAT elnevezésű regiszterbe kerül. Íráskor a TABLAT regiszter tartalma kerül a TBLPTR által megcímezett memóriahelyre (2.6. ábra). Fizikailag a TBLPTR regiszter három nyolcbites regiszterből épül fel:

$$\text{TBLPTRU} = \text{TBLPTR}[21:16] \quad \text{TBLPTRH} = \text{TBLPTR}[15:8] \quad \text{TBLPTRL} = \text{TBLPTR}[7:0]$$



2.6. ábra
PIC18 táblakezelés

Az adatmozgatás két utasítást igényel. Olvasáskor először kiolvassuk az értéket a TABLAT regiszterbe, utána az eredményt valahová tároljuk. Például:

| | | |
|-------|--------------|-----------------------------|
| TBLRD | * | ; PROGMEM(TBLPTR) -> TABLAT |
| MOVFF | TABLAT, REGI | ; TABLAT TARTALMA REGI-BE |

A hatékonyság érdekében vannak műveletek, ahol a táblamutató változtatása is automatikus, az utasításban összekapcsolódik a memóriatartalom-kezelő és a mutató mozgató művelet:

| | |
|-----------------|---|
| TBLRD*, TBLWT* | TBLPTR tartalma nem változik |
| TBLRD*, TBLWT*+ | TBLPTR tartalma a művelet után eggyel nő |
| TBLRD*, TBLWT*- | TBLPTR tartalma a művelet után eggyel csökken |
| TBLRD*, TBLWT+* | TBLPTR tartalma a művelet előtt eggyel nő. |

3) A programmemória azon része, amiben adatok vannak, az adatmemória egy adott címtartományában adatként „látszik”. Ezt részletesebben a 16 bites mikrovezérlőknél mutatjuk be.

2.5. MEGSZAKÍTÁS

Ha egy számítógépes rendszerben valamilyen esemény létrejöttét kívánjuk érzékelni, ezt szokásos módon kétféleképpen tehetjük meg. Az első módszernél a külső esemény létrejöttét egy bemenet változásának figyelésével érzékelhetjük.

Például ilyen megoldás alkalmazható, amikor egy billentyűzetről akarunk beolvasni. Bár melyik billentyű megnyomásakor a billentyűzet kimenetén lévő „adat érvényes” jel szintet vált. Ha ezt egy bemeneti port egyik bitjére kötjük, akkor az állapotának a programból való figyelése lehetővé teszi a billentyű megnyomásának az érzékelését, majd a kód beolvasását.

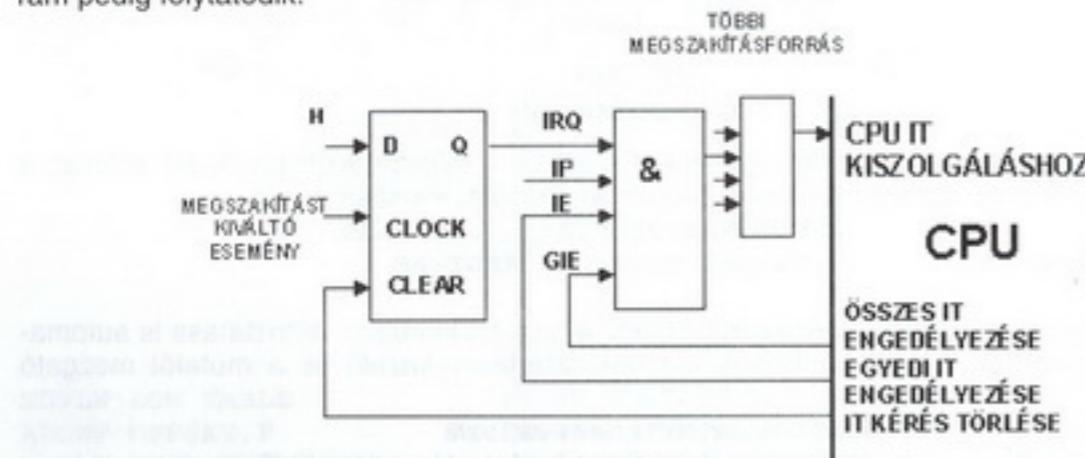
Ezt a módszert általában elterjedt kifejezéssel **pollingnak** (lekérdezésnek), azaz programozott átvitelnek hívják. Alkalmazása azonban lelassítja a rendszer tényleges működési sebességét, hiszen a mikroprocesszor az idejének nagy részét azzal tölti, hogy ciklikusan megvizsgálja a kijelölt bemeneti bit állapotát.

Sokkal szerencsésebb, ha az esemény maga jelzi a processzor számára állapotának megváltozását. Ez a megoldás a **megszakítás** vagy ismert angol kifejezéssel az *interrupt* (szokták IT-nek rövidíteni).

A megszakítás az eredetileg futó program utasításainak végrehajtását leállítja, és a processzor egy ún. megszakítási alprogramot (ISR – *Interrupt Service Routine*) hajt végre, ami az esemény kezelését elvégzi, majd ennek befejeztével a processzor visszatér a megszakított program végrehajtására. Lényegében a programunk egy szubrutint hajt végre, majd visszatér folytatni az eredeti programot.

Az előbbi példánál maradva, a billentyű megnyomását jelző „adat érvényes” jel megszakítást okoz, a megszakítási alprogram elvégzi a lenyomott billentyűhöz tartozó kód beolvasását, majd utána folytatódik a megszakított program.

A processzor oldaláról a megszakítási lehetőség kialakítása azt kívánja meg, hogy legyenek olyan bemenetei, amelyek állapotainak megváltozásakor képes a processzor az éppen futó program utasításainak végrehajtását felfüggeszteni, a megszakított program programszámlálójának az értékét elmenteni, és helyébe a megszakítási alprogram kezdőcímét betölteni, és ilyen módon az alprogramot elindítani. A végrehajtás befejeztével (amit általában az utolsónak elhelyezett, speciális utasítás jelez) a programszámlálóba a megszakított program programszámlálójának elmentett értéke töltödik vissza, a megszakított program pedig folytatódik.



2.7. ábra
A megszakítás elve

A megszakítás olyan speciális szubrutinhívás, amelynél a hívás bekövetkezésének időpontját nem tudjuk.

Mivel több megszakítást alakítanak ki, a megszakítás kiszolgálásának első lépése a megszakítást kiváltó azonosítása. Ezért megszakításkor, egy ahhoz tartozó IR (*Interrupt Request*) bit – addig 0 állapotú – bit fog 1-re váltani. Ezek a megszakításokhoz tartozó IR bitek jelzik az adott megszakítás létrejöttét.

Ha a processzor több megszakítási vonallal rendelkezik, ezek mindegyikéhez egy-egy eseményt rendelhetünk hozzá. Olyan rendszerekben, ahol több esemény okozhat megszakítást, megtörténhet, hogy egyszerre egy időben két megszakítás is fellép. Ilyen esetben a megszakítások kiszolgálásának fontossági sorrendje – **prioritása** – dönti el a kiszolgálási sorrendet. Ezt az IP bit jelöli.

A program futása nem minden esetben szakítható meg káros következmények nélkül. Ezért a legtöbb rendszer biztosítja, hogy a megszakítások programból tilthatók, illetve engedélyezhetők legyenek, erre szolgálnak az IE (*Interrupt Enable*) bitek. Több megszakítás esetén a megszakításokat egyenként kell engedélyezni, illetve tiltani, ezért bevezették a GIE bitet, amivel egyedül az összes megszakítás engedélyezhető, illetve tiltatható.

Ha több megszakításforrás van, akkor a PC-be be kell tölteni az adott megszakításhoz tartozó kiszolgálórutin (ISR – *Interrupt Service Routine*) kezdőcímét. Az így működő kialakítás a vektoros megszakítás. A **megszakításvektor** a megszakítás kiszolgálásakor betöltött rutin kezdőcíme.

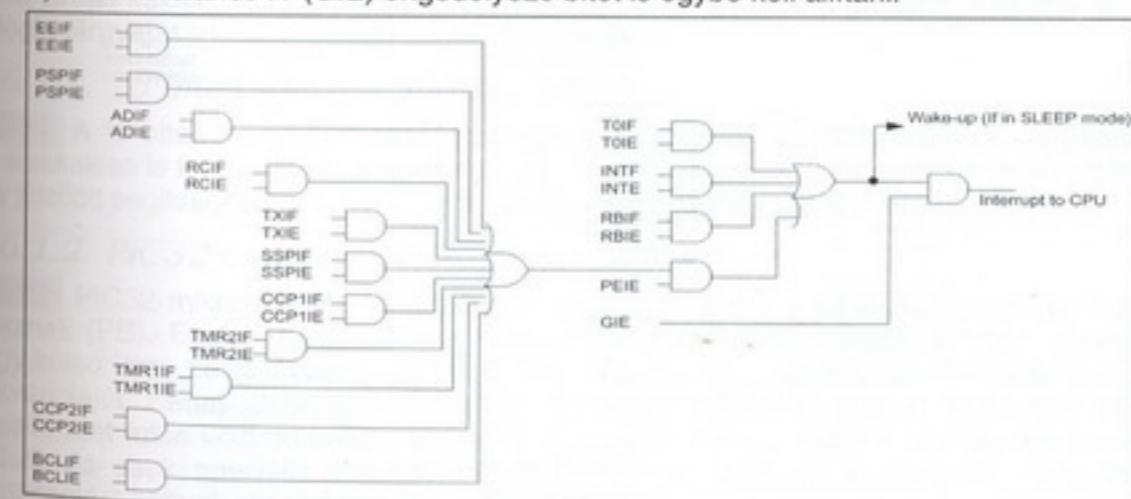
8/12 A legelső, **PIC16C5xx** család nem is rendelkezett megszakítással, minden eseményfigyelés csak lekérdezéssel (pollinggal) volt lehetséges. A következő típusokban jelent a megszakítás, minden maximum két visszatérési címet tároló veremmel, ami azt jelentette, hogy vagy két egymásba skatulyázott szubrutinhívás, vagy egy megszakításban egy szubrutinhívás volt lehetséges. A megszakítás egyszintű, azaz egyszerre csak egy megszakítás kiszolgálása történhet, az egy időben bekövetkező megszakítás esetén a prioritást az ISR-ben való lekérdezési sorrend határozza meg. Több megszakításforrás lehetséges, de a megszakításoknak csak egy vektorcíme van (04h).

8/14 **PIC16-os** termékvonalonál a verem mélysége nyolcra növekedett.

A megszakítás létrejöttekor – mivel egy megszakítási cím van – még nem tudjuk, melyik forrás okozta a megszakítást. Ezért a megszakítási alprogram elején meg kell vizsgálni az egyes megszakításhoz tartozó IR bitek állapotát. Amelyik H állapotú, az a forrás okozta a megszakítást. Több forrás esetén a bitek lekérdezési sorrendje a prioritást is megadja, hiszen több IT egyidejű bekövetkezése esetén a lekérdezés sorrendje alapján az első IT-t okozó forrást szolgáljuk ki. A megszakítás kiszolgálása végén töröljük a D tárolót, amely az IT kérést tárolta a kiszolgálás alatt.

Több megszakítás a processzort a szundi (*sleep*) módból ébreszti, a megszakítás felismerése 3 utasításciklus, illetve 4 ciklus külső megszakításoknál. A megszakítás kiszolgálásakor fontos megörizni a STATUS, a W regiszter, valamint a PCLATH regiszter tartalmát. Az egyes megszakításforrások engedélyezése-tiltása az INTCON, PIE1 és PIE2 regiszterek segítségével lehetséges. A megszakítások globális engedélyezésére-tiltására a GIE bit szolgál. A megszakítás kiszolgálásának kezdetekor a hardver törli a GIE bitet, és a visszatérési címet a verembe helyezi. Több perifériát tartalmazó típusoknál még az ezek megszakításait külön engedélyező PIE bit állítása is szükséges (háromszintű engedélyezés: egyedi – PIE – GIE).

A 2.8. ábra a **PIC16F87x** megszakításrendszerét mutatja. Az IF végződés a megszakításjelző bitet, az IE végződések a megszakítás engedélyezését jelölik. Látható az ábrán, hogy perifériák megszakításainak engedélyezéséhez az egyedi (xxIE), a közös periféria (PEIE) és az általános IT (GIE) engedélyező bitet is egybe kell állítani.



2.8. ábra
A PIC16F87x megszakításrendsze

8/16 A **PIC18Fxxx** család számos megszakítási lehetőséggel rendelkezik. Már két, eltérő prioritású megszakítási vektor van, ezért a minden megszakításhoz az eddigi, az adott megszakítást egyedileg engedélyező IE (*interrupt enable*) és a megszakításhoz tartozó esemény bekövetkezését jelző IF (*interrupt flag*) bitek mellé egy újabb bit társult. Ez a megszakítás prioritását jelző IP (*interrupt priority*) bit.

A megszakítások prioritásos kezelése csupán lehetőség, ha nem alkalmazzuk, akkor bármelyik megszakításkor a **0008h** cím töltödik be az utasításszámlálóba.

A prioritásos megszakításrendszer alkalmazásakor minden megszakításhoz a két prioritási szint valamelyikét rendeljük hozzá. Az alacsonyabb prioritású megszakítás bekövetkezékor a **0018h** cím íródik be az utasításmutatóba, míg a magasabb prioritású megszakítások bekövetkezésekor a **0008h** cím töltödik. Úgy is fogalmazhatunk, hogy az IP bit értéke határozza meg azt, hogy melyik cím töltödik be az utasításszámlálóba.

Hogy működik a prioritásos megszakítási rendszer? Ha egy alacsonyabb szintű megszakítás kiszolgálása közben egy magasabb prioritású megszakítás következik be, akkor az alacsonyabb szintű megszakítás kiszolgálása felfüggesztődik, és a magasabb szinten lévő kerül végrehajtásra. Ennek befejeződése után a felfüggesztett alacsonyabb szintű megszakítás kiszolgálása folytatódik. Azonos szinten lévő megszakítások egymást nem szakítják meg.

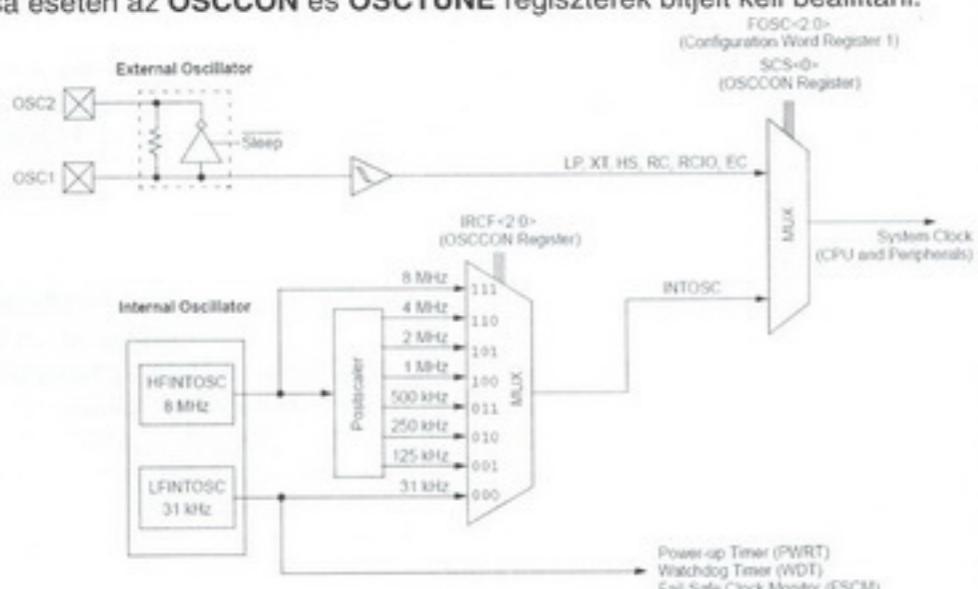
A 16 és 32 bites PIC mikrovezérlők megszakítás-kialakítását azok leírásánál ismertetjük.

2.6. RENDSZERELEMÉK

Ebben a részben a mikrovezérlők működéséhez szükséges rendszerelemeket foglaljuk össze.

2.6.1. Oszcillátor, órajel

Az órajelgenerátor általános vázlata a 2.9. ábrán látható. A külső oszcillátor-csatlakozás mellett egy belső oszcillátorblokk is megtalálható. Adott típusnál az adatlap tanulmányozása alapján tudhatjuk meg, hogy a konkrét esetben milyen megoldásokat valósítottak meg. Az órajel forrásának és típusának a beállítása a konfigurációs bitekkel lehetséges. Belső órajel választása esetén az OSCCON és OSCTUNE regiszterek bitjeit kell beállítani.



2.9. ábra
Órajelforrások

A működtető órajel előállítására számos lehetőség van, ezek a következők:

I. Külső kristály vagy kerámiarezonátor használata:

- 1) LP – Low Power Crystal – kis frekvenciájú (max 200 KHz) kristály vagy kerámiarezonátor
- 2) XT 0,4 – 4 MHz-es kristály vagy kerámiarezonátor

2. fejezet: A PIC mikrovezérlők felépítése, fejlődése

3) HS 4 – 20 MHz-es kristály vagy kerámiarezonátor

4) HSPLL 4 – 10 MHz kristály, aminek frekvenciáját egy belső PLL áramkör még négyesresére növeli.

II. Külső RC tag használata

- 5) RC – Külső R-C tag használata, javasolt értékek: $3k < R < 100k$, $C > 20 \text{ pF}$
- 6) RCIO – Külső R-C tag használata, az OSC2 kivezetést I/O lábnak lehet használni.

III. Belső órajel-generátor használata

- 7) INTIO1 – Belső oszcillátor használata, az OSC2 kivezetésen megjelenik a generált frekvencia $\frac{1}{4}$ -e, az OSC1 kivezetést I/O lábnak lehet használni.
- 8) INTIO2 – Belső oszcillátor használata, az OSC1 és az OSC2 kivezetést I/O lábnak lehet használni.

IV. Külső órajelforrás használata

- 9) EC – Külső órajel használata, az OSC2 kivezetésen megjelenik a frekvencia $\frac{1}{4}$ -e.
- 10) ECIO – Külső órajel használata, az OSC2 kivezetést I/O lábnak lehet használni.

Az órajel kezelése nagy mértékben kihat egy tok fogyasztására. Ezért a nanowatt technológiával tárnyalásánál ezt részletesen fogjuk tárgyalni.

2.6.1.1. 16 bites PIC oszcillátor, órajel

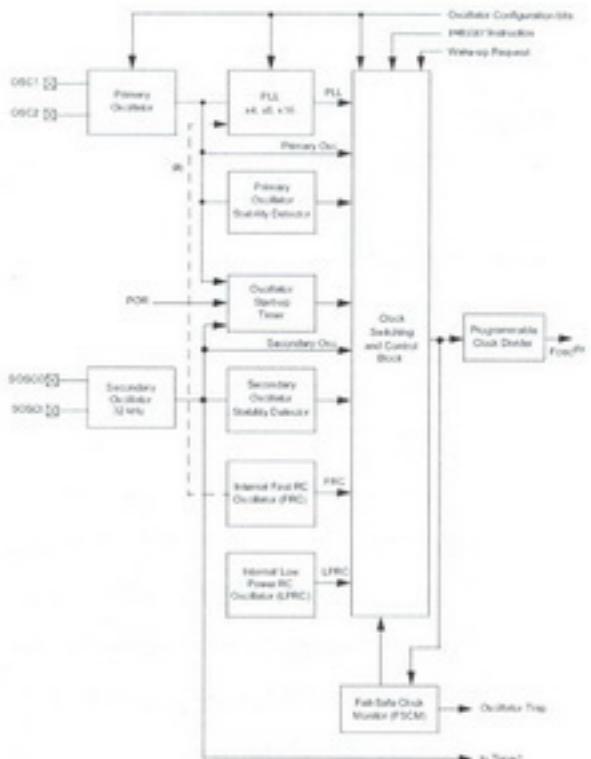
16/24 A 16 bites mikrovezérlőknél az órajel-átkapcsolások miatt az órajelek stabilitásának az érzékelése is fontos, tehát megjelentek az ellenőrző érzékelők, amelyek az órajel-stabilitást egy jelzőbit segítségével jelzik. Az óragenerátor megoldást a 2.10. ábrán mutatjuk be.

2.6.1.2. PIC32 oszcillátor, órajel

32/32 PIC32 mikrovezérlőknél két belső órajel van: egyik a CPU-nak, másik a Periféria Busznak (PB). Ezeket az aktuálisan kiválasztott órajelforrásból származtatjuk. Összesen négy külső vagy belső órajelforrás lehet. Némelyiknek PLL áramkörrel lehet sokszorozni a frekvenciáját, illetve az elő- és utóosztóval számos frekvenciaértéket állíthatunk be. Az órajelforrás futás közben programból változtathatjuk. Az oszcillátor vezérlése hardveresen védeott, és csak speciális utasítássorozattal lehet a védelmet kikapcsolni, hogy órajel-átkapcsolást tudjunk megvalósítani.

Három fő órajele van a PIC32 eszközöknek:

- A CPU és néhány periféria által használt rendszerórájel, System Clock (SYSCLK),
- A legtöbb periféria által használt perifériabusz-órájel, Peripheral Bus Clock (PBCLK),
- USB órajel, USB Clock (USBCLK).



2.10. ábra
16 bites mikrovezérlők óragenerátorához kapcsolódó blokkvázlat

Ezek az órajelek a következő forrásokból származhatnak:

- Az OSCI és OSCO lábakra kapcsolt elsődleges oszcillátor, *Primary Oscillator (POSC)*,
- Az SOSCI és SOSCO lábakra kapcsolt másodlagos oszcillátor, *Secondary Oscillator (SOSC)*,
- Belső gyors oszcillátor, *Internal Fast RC Oscillator (FRC)*,
- Belső, kis fogyasztású oszcillátor, *Internal Low-Power RC Oscillator (LPRC)*.

Mindegyik oszcillátor egyedileg konfigurálható (PLL, elő-, illetve utóosztó).

| Oszcillátor mód | Órajel-forrás |
|---|---------------|
| Fast RC Oscillator with Postscaler (FRCDIV) | belso |
| Fast RC Oscillator divided by 16 (FRCDIV16) | belso |
| Low-Power RC Oscillator (LPRC) | belso |
| Secondary (Timer1/RTCC) Oscillator (SOSC) | másodlagos |
| Primary Oscillator (HS) with PLL Module (HSPLL) | elsődleges |
| Primary Oscillator (XT) with PLL Module (XTPLL) | elsődleges |
| Primary Oscillator (EC) with PLL Module (ECPLL) | elsődleges |
| Primary Oscillator (HS) | elsődleges |
| Primary Oscillator (XT) | elsődleges |
| Primary Oscillator (EC) | elsődleges |
| Fast RC Oscillator with PLL Module (FRCPLL) | belso |
| Fast RC Oscillator (FRC) | belso |

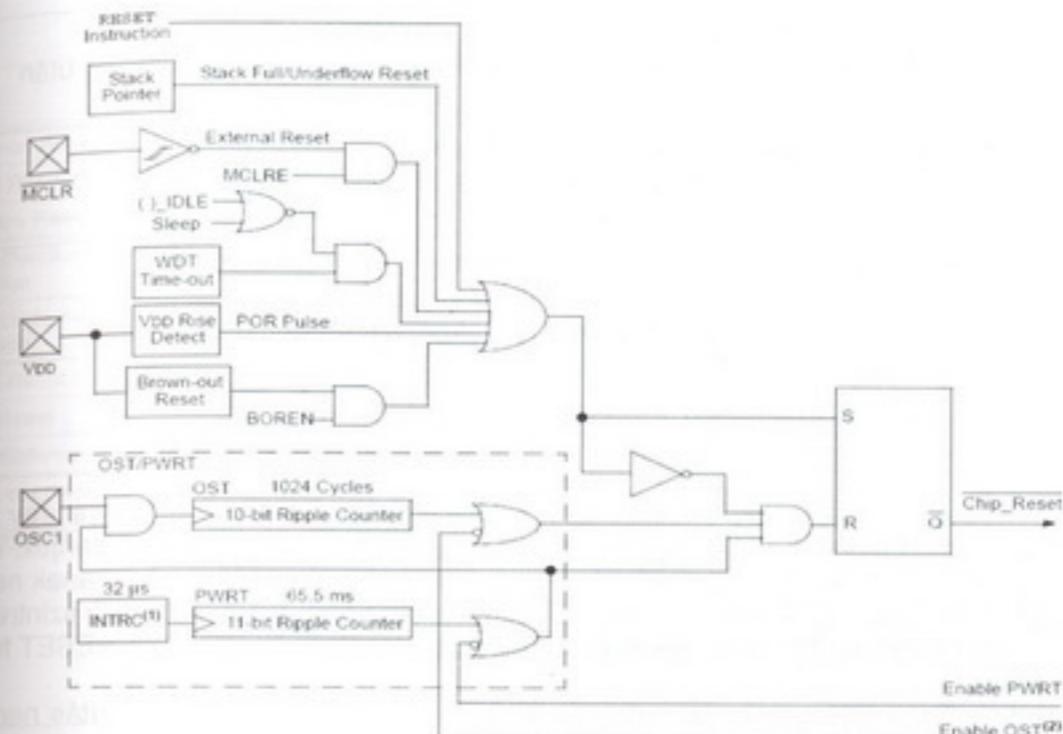
2.11. ábra
PIC32 oszcillátor módok

2.6.2. Reset

Áramkori felépítését a 2.12. ábrán foglaltuk össze. A tápfeszültség bekapcsolásakor a mikrovezérlő belső áramköreit, regisztereit a megfelelő működés miatt jól meghatározott alaphelyzetbe kell állítani. Ez a RESET folyamat. Ehhez a belső órajel stabil működése szükséges! Bekapcsoláskor elindul egy független belső RC oszcilláttorról működő 10 bites számláló (POWER-UP TIMER – PWRT). Ennek szerepe az esetleges lassú tápfeszültség növekedésének a kompenzációja. Amikor ez túlcordul, elindul egy, az oszcillátor esetleges lassú berezgése miatti hibás működés kivédését célzó, második késletetést biztosító számláló (OSC START-UP TIMER – OST). A mikrovezérlő belső regisztereit alapállapotba hozó RESET folyamat bekövetkezését több esemény kiválthatja, amelyek száma a PIC mikrovezérlök fejlesztésével egyre nő.

Fontos tudni, hogy RESET-kor nem csupán az utasításszámlálóba kerül egy érték, hanem számos regiszter kezdőértéke beállítódik, amit a mikrokontrollerek adatlapjain nézhünk meg.

Mivel RESET-et több esemény is kiválthat, ezért a RESET-ből indulás után általában meg kell határozni, hogy mi okozta a RESET állapotot.



2.12. ábra
PIC18 RESET áramkör felépítése

8/14 A PIC18-as családig ez a STATUS regiszter T0 és PD bitjeinek, illetve a PCON regiszter POR és BOR bitjeinek vizsgálatával volt lehetséges.

T0 bit nulla állapota jelzi, hogy WDT túlcordulás történt, míg PD bit SLEEP utasítás után kerül 0 állapotba. Alaphelyzetben a britek értéke 1.

8/16 A PIC18-as családban a RESET modult is átterveztek; a jelzőbitel az RCON regiszterbe kerültek. Ennek az volt az oka, hogy a minél kisebb fogyasztást megcélzó Nanowatt Technológia megjelent. Ezeknél a RESET-et kiváltó események:

- **Bekapcsolási (Power-on) Reset (POR):** Tokra kapcsolt feszültség hatására, amikor az elér egy bizonyos feszültségszintet, RESET impulzus keletkezik. Ez biztosítja, hogy a tokban lévő áramkörök akkor kezdjenek működni, amikor a tápfeszültségük már megfelelő nagyságú. Ezt a működést a legegyszerűbben úgy érhetjük el, hogy a tok MCLR ki-vezetését 1–50 kΩ-os ellenálláson keresztül a tápfeszültségre (VDD) kötjük. Egy kis értékű kapacitás használatával a RESET folyamat elnyújtható. A POR eseményt a POR bit (RCON<1>) jelzi. Ez a bit 0 lesz, ha POR történt, és más, RESET-et kiváltó esemény hatására állapota nem változik. Ha szükséges, programból állítható ismét 1-be.

| RCON REGISZTER BITJEI | |
|-----------------------|--|
| BIT 7 | IPEN – Megszakításprioritást engedélyező bit 1 – prioritás engedélyezve 0 – prioritás használata tiltva (PIC16CXXX-cal való kompatibilitás) |
| BIT 6-5 | Nem használt britek |
| BIT 4 | /RI Reset utasítást jelző bit 1 – Nem volt végrehajtva RESET utasítás 0 – RESET utasítás okozta a tok RESET-et, programból kell 1-be állítani Brown-out Reset után |

| | |
|--------------|---|
| BIT 3 | /TO Watchdog Time OUT jelző bit 1 – táp bekapcsoláskor, CLRWDT vagy SLEEP utasítás végrehajtása után 0 – WDT túlcsordulás történt |
| BIT 2 | /PD Tápkikapcsolást (Power Down) jelző bit 1 – tápbekapcsolás vagy a CLRWDT utasítás állítja 1-be 0 – SLEEP utasítás törli ezt a bitet |
| BIT 1 | /POR – Tápbekapcsolási Reset állapot bit 1 – Nem volt Tápbekapcsolási Reset (programból kell 1-be állítani) 0 - Volt Tápbekapcsolási Reset (ezután programból kell 1-be állítani) |
| BIT 0 | /BOR 1 – Nem volt Brown OUT (programból kell 1-be állítani) 0 - Volt Brown OUT (ezután programból kell 1-be állítani) |

- MCLR Reset normál működéskor:** A tok MCLR jelű lába teszi lehetővé egy külső RESET végrehajtását, ami akkor jön létre, ha ezt a lábat alacsony szinten tartjuk. A lábhoz kapcsolódó belső szűrő áramkör biztosítja, hogy a külső zajok ne okozzanak nem kívánt RESET-et. Az MCLR láb más RESET eseménykor nem megy alacsony szintre. Sok típusnál az MCLR láb bemenetnek is konfigurálható ilyenkor, annak külső RESET funkciója elvész.
- Watchdog Timer túlcsordulása okozta RESET:** Ilyenkor a CLRWDT utasítás nem törli WDT-t.
- Reset, előtte Sleep állapot volt.**
- Programozható Brown Out Reset (BOR):** Ha a tápfeszültség rövid időre lecsökken, az RESET-et okoz. Mi van, ha a RESET folyamat alatt ismét lecsökken a feszültség? Ez általában hibás RESET-et okoz. A valóságban, ipari környezetben gyakran előfordulhat ilyen jelenség.
A megoldás erre a problémára a **BOR** áramkör kialakítása, amely biztosítja, hogy a RESET-kor minden az utolsó tápfeszültség-csökkenéstől induljon újra a RESET folyamat. A BOR áramkör aktivizálódási szintje is megadható.
- MCLR Reset fogyasztáskezelési módokban:** Erről a Nanowatt Technológiáról ismertető részben írnunk.
- RESET utasítás hatására (RI bit):** A PIC18-as családtól kezdődően megjelent a RESET utasítás, aminek a hatása egyező a hardver RESET-tel.
- Verem tele (Stack Full) Reset-Verem alulcsordult (Stack Underflow) Reset:** A veremhiba is RESET-et okozhat, ha a megfelelő konfigurációs bitet bekapcsoljuk.

Az RCON regiszter alsó öt bitje jelzi azt, hogy RESET esemény következett be. A legtöbb esetben a biteket maga az esemény törli, és a programnak kell 1-be állítania az esemény kiértékelése után. A bitek együttes vizsgálata alapján dönthetjük el, hogy mi okozta a RESET-et. A 2.13. ábrán ábrázoltuk az RCON bitjeit, illetve hogy állapotuktól hogyan következtethetünk a RESET eseményre.

| RCON REGISZTER | R/W-0 | R/W-0 | U-0 | R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-0 |
|-------------------|-------|-------|-----|-------|-------|-------|-------|---------------------|
| | IPEN | LWRT | - | RI | TO | PD | POR | <u>BOR</u> bit 0 |

RESET ÁLLAPOTOK - BITEK ÖSSZEFÜGGÉSE

| Condition | Program Counter | RCON Register | RI | TO | PD | POR | BOR | STKFUL | STKUNF |
|---|-----------------------|---------------|----|----|----|-----|-----|--------|--------|
| Power-on Reset | 0000h | 00-1 1100 | 1 | 1 | 1 | 0 | u | u | u |
| MCLR Reset during normal operation | 0000h | 00-u uuuu | u | u | u | u | u | u | u |
| Software Reset during normal operation | 0000h | 0u-0 uuuu | 0 | u | u | u | u | u | u |
| Stack Overflow Reset during normal operation | 0000h | 0u-u uull | u | u | u | u | u | u | 1 |
| Stack Underflow Reset during normal operation | 0000h | 0u-u uull | u | u | u | u | u | 1 | u |
| MCLR Reset during SLEEP | 0000h | 00-u 10uu | u | 1 | 0 | u | u | u | u |
| WDT Reset | 0000h | 0u-u 01uu | 1 | 0 | 1 | u | u | u | u |
| WDT Wake-up | PC + 2 | uu-u 00uu | u | 0 | 0 | u | u | u | u |
| Brown-out Reset | 0000h | 0u-1 1iu0 | 1 | 1 | 1 | 0 | u | u | u |
| Interrupt Wake-up from SLEEP | PC + 2 ⁽¹⁾ | uu-u 00uu | u | 1 | 0 | u | u | u | u |

u - NEM VÁLTOZIK - NINCS BIT, OLVASVA 0

2.13. ábra
PIC18 RESET állapotok

| Flag Bit | Set by: | Cleared by: |
|------------------|---|-------------------------|
| TRAPR (RCON<15>) | Trap conflict event | POR |
| IOPWR (RCON<14>) | Illegal opcode or uninitialized W register access | POR |
| EXTR (RCON<7>) | MCLR Reset | POR |
| SWR (RCON<6>) | RESET instruction | POR |
| WDTO (RCON<4>) | WDT time-out | PWRSAV instruction, POR |
| SLEEP (RCON<3>) | PWRSAV #SLEEP instruction | POR |
| IDLE (RCON<2>) | PWRSAV #IDLE instruction | POR |
| BOR (RCON<1>) | POR, BOR | |
| POR (RCON<0>) | POR | |

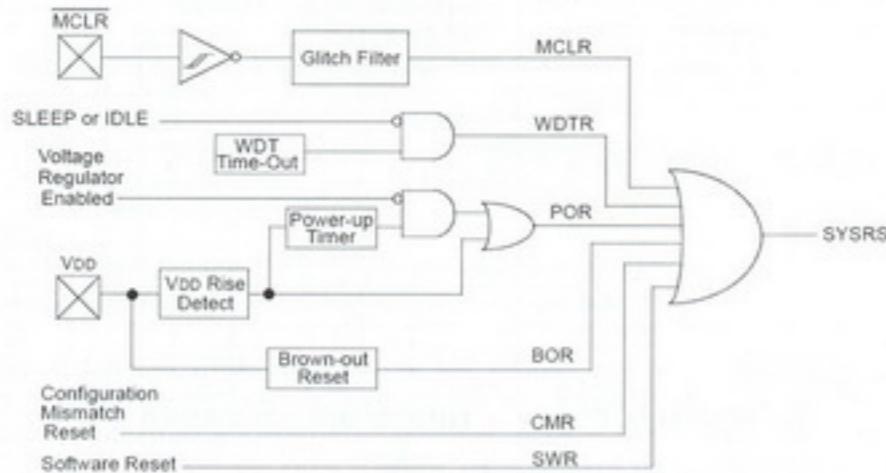
2.14. ábra
RESET a 16 bites PIC mikrovezérlőknél

16/24 16 bites mikrovezérlőknél az RCON regiszter már 16 bites. Az alsó bitek funkcióiban megegyeznek a PIC18 családnál leírtakkal, de mivel továbbra is két energiatakarékos mód van: SLEEP (CPU, perifériák leállnak) és az IDLE (CPU leáll, perifériák tovább működnek) ezért van szükség szétválasztásra. A SLEEP utasítás helyett megjelent a PWRSAV utasítás egy bites paraméterrel: PWRSAV #0 – SLEEP, PWRSAV #1 – IDLE.

A rendszerfelépítésből adódóan a veremhiba helyett a megszakítási konfliktusok, illetve hibás műveleti kód vagy nem feltöltött W regiszter forrásoperandusként történő használata is okozhat RESET-et.

32/32 32 bites mikrovezérlőknél egyszerűsödött a RESET kezelése. A 32 bites RCON regiszter alsó 5 bitje megegyezik a 16 bites RCON regiszter bitjeivel. A programból nem adható ki RESET utasítás – ilyen az átvett 32 bites utasításkészletet tartalmazó proceszszorban nem volt, hanem a kézikönyv ennek a megoldására egy utasításszekvenciát

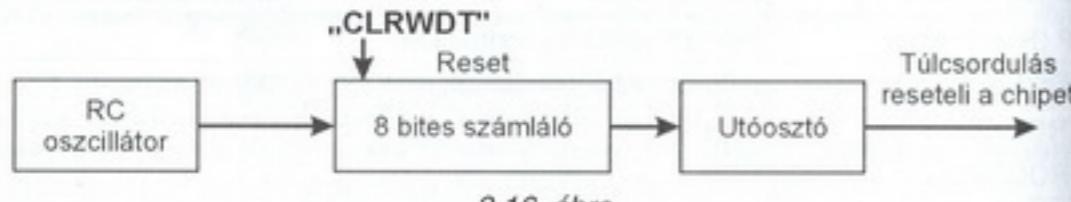
javasol. Szoftverhiba okozta RESET csak egy van: a konfiguráció-nem-egyezés RESET. A konfiguráció helyes megőrzése érdekében minden konfigurációs bitnek a negáltját tárolják le. Ezeket a letárolt biteket összehasonlják, amikor a konfigurációt úratöltenek, ami például SLEEP utasításkor is megtörténik. Ha valahol nincs egyezés, az RESET-et okoz.



2.15. ábra
PIC32 RESET-kialakítás

2.6.3. Watchdog

A watchdog (WDT – Watchdog Timer) lényegében egy belső, szabadonfutó RC oszcillátor-órájellel léptetett 8 bites számláló, amelyhez utóosztó kapcsolódik. A programban elhelyezett, periodikusan végrehajtott CLRWDT utasítással töröljük a számlálót és az utóosztót. Ha a program „elkószál”, azaz valamilyen elektromos zaj miatt rossz címre lépve hibásan működik, akkor a törlés elmaradása miatti túlcsordulás nullázza és újraindítja a kontrollert. A mikrovezérlők SLEEP állapotában, ha engedélyezzük, a WDT tovább fut, és a túlcsordulása felébreszti a mikrovezérlőt a SLEEP-ből.



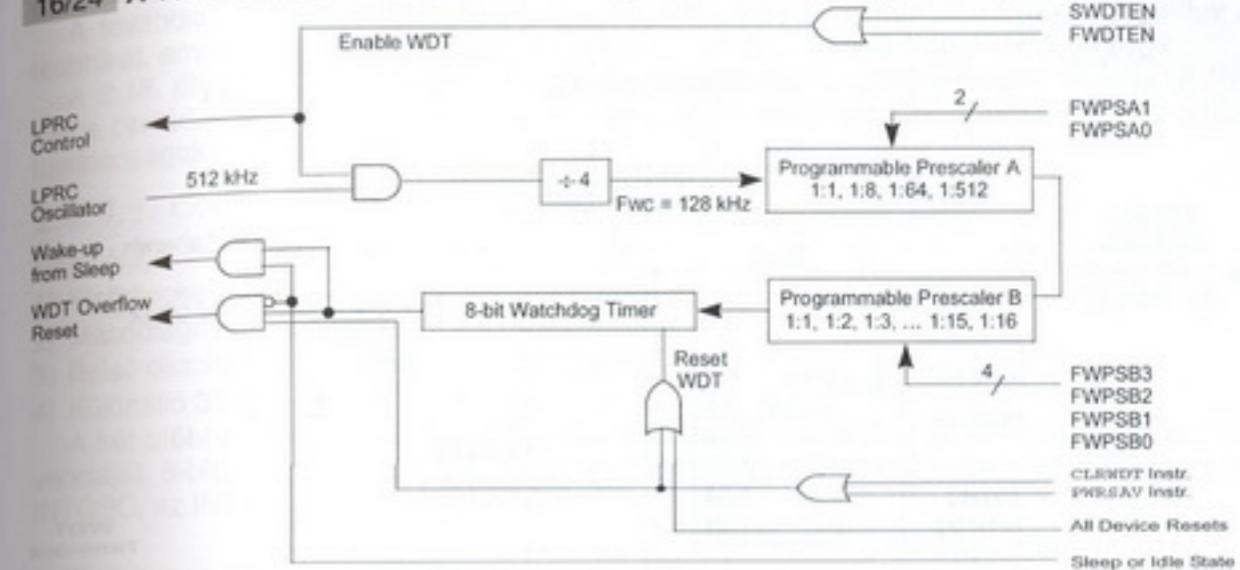
2.16. ábra
Watchdog blokkvázlata

FONTOS! Mivel a WDT 8 bites számlálója nem írható, ezért az adatlapok a számláló túlcsordulási periódusidejét adják meg utóosztó nélkül. Ez az érték hőmérséklet- és tápfeszültséggfüggő, ezért a hozzá tartozó diagramokat is mellékelik. Névleges értéke 18 ms.

8/14 A PIC18-as családig a WDT konfigurációs bittel engedélyezhető, és az utóosztó osztozik a TMR0 számláló utóosztójával. Az utóosztó WDT-hez, vagy TMR0-hoz rendelhető, de ez működés közben átkapcsolható. Bekapcsolt állapotát csak újabb tokprogramozás-sal (a konfigurációs bit módosításával) lehetséges megváltoztatni.

8/16 A PIC18-as családnál módosították a WDT-t. Önálló utóosztót kapott, ami nemcsak 8 bites, hanem akár 16 bites is lehet. Normál működéskor a WDT túlcordulás-RESET-et generál. Ha az eszköz SLEEP-ben van, a WDT túlcordulás felébreszti az eszközt, és a program fut tovább. A WDTEN konfigurációs bittel engedélyezhetjük a WDT működését. Ha ezt a bitet nem kapcsoljuk be, akkor programból állíthatjuk a SWDTEN bitet, ami engedélyezi/tiltja a WDT áramkört.

16/24 A 16 bites PIC mikrovezérlőknél folytatódott a továbbfejlesztés (2.17. ábra).



2.17. ábra
WDT 16 bites PIC-eknél

WDT Periódusidő = 2 ms × Prescale A × Prescale B

Megjegyzés:

Prescale B max = 16384 ms

WDT Periódusidő min = 2 ms.

32/32 PIC32 WDT, ha engedélyezett, a belső Low-Power RC (LPRC) oszcillátor az órajele. A WDT 25 bites utóosztójával számos időtartamot be tudunk állítani. A WDT használható a mikrovezérlő SLEEP vagy IDLE módjából történő ébresztésre.

2.6.4. Microchip nanowatt technológia

A tok teljesítményfelvételének csökkentése érdekében három fogyasztáskezelési lehetőség van, attól függően, hogy a CPU, illetve a perifériák működését fenn kívánjuk-e tartani. Ezek a következők:

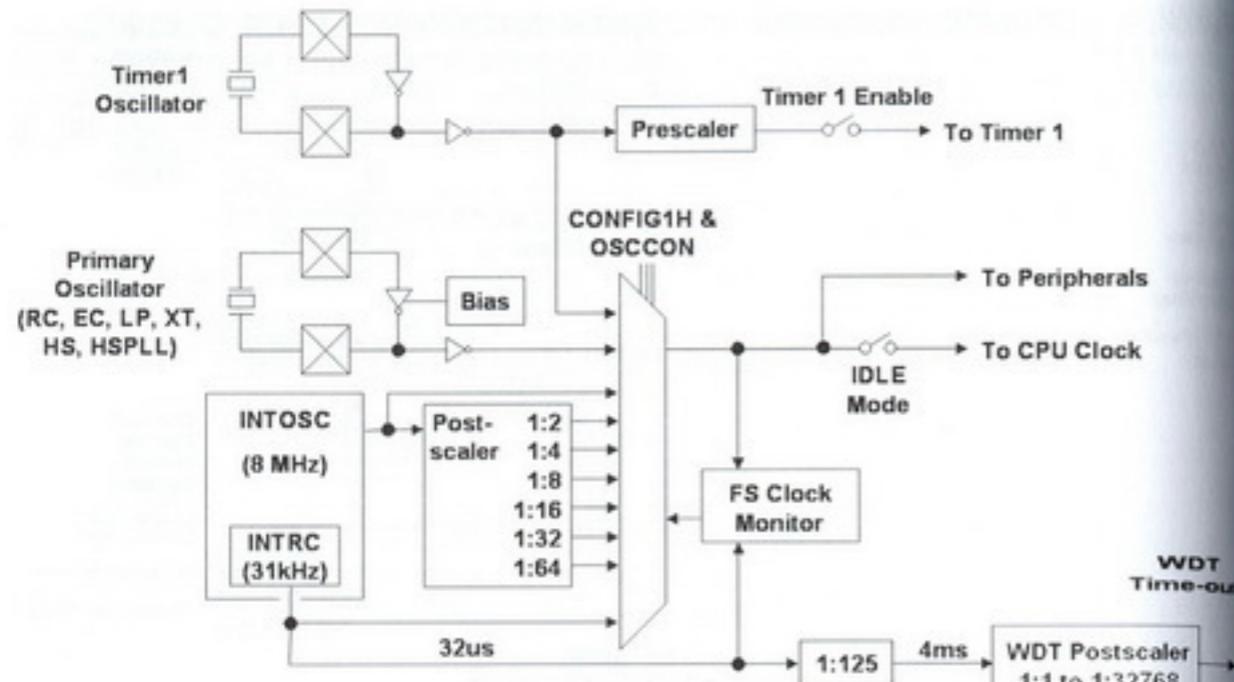
- **RUN** mód – ez az üzemszerű működés.
 - **IDLE** mód – a CPU működése megszűnik, de a perifériák tovább működnek.
 - **SLEEP** mód – ilyenkor a tokban lévő CPU és a perifériák is kikapcsolnak, azaz az óra-jellel történő működésük megszűnik, ilyenkor a legkisebb a fogyasztás.

Általában négy óraielforrás közül lehet választani (2.18. ábra), ezek:

- Elsőleges (Primary Oscillator),
 - Másodlagos (secondary) (Timer1 Oscillator)
 - Belső INTOSC (Internal Oscillator Block - IOB)
 - Belső RC oszcillátor (INTRC)

Lehetséges egy tokon belül az elsődleges és másodlagos órajel felváltva történő használata, aminek kettős előnye van:

- Az egyik órajelforrás meghibásodása esetén át lehet kapcsolni a másik órajelre.
 - Mivel ezek sebessége jelentősen különböző lehet, ezért a fogyasztás csökkentése érdekében lehetséges a kisebb sebességre átkapcsolni, ha az aktuális programvégrehajtás nem igényel nagy számítási teljesítményt.



2.18. ábra
PIC 18 órajel előállítása

A tok fogyasztásának a kezelését a SLEEP utasítás végrehajtásával kezdeményezhetjük, és az OSCCON regiszter bitjeinek beállításával választhatjuk ki a kívánt üzemmódot.

| Az OSCCON REGISZTER BITJEI | | | | | | | |
|----------------------------|--|-------|-------|------|------|------|------|
| IDLEN | IRCF2 | IRCF1 | IRCF0 | OSTS | IOFS | SCS1 | SCS0 |
| IDLEN | CPU órajeltiltás/-engedélyezés (IDLE mód beállítása (csak PIC 18-nál)) | | | | | | |
| IRCF<2:0> | INOSC utóosztója osztásviszonyának állítása | | | | | | |
| OSTS | 1, ha az elsődleges oszcillátor adja az órajet | | | | | | |
| IOFS | 1, ha INTOSC kimenete már stabil | | | | | | |
| SCS<1:0> | Órajelforrás kiválasztása a teljesítménykezeléshez | | | | | | |

Mivel ez a technika megvalósítja a tok fogyasztásának a teljes szabályozását, telepes működés esetén megnő a működési időtartam, mert az áramkör fogyasztása nagymértékben csökken változó üzemmódok használatával. A technológia kismértékben különbözik a PIC16-os és PIC18-as családonknál, és nem minden családtípus rendelkezik ezzel a képességgel. A 16 és 32 bites mikrovezérlőknél ez a technológia minden családtípusban működik.

A technológia több tulajdonság együttes alkalmazását jelenti, ezek:

- 1) Órajel előállítása (**Clock system**)
- 2) Kétsebességű indítási mód alkalmazása (**Two-speed start-up**)
- 3) Hibás órajel figyelése (**Fail Safe Clock Monitor – FSCM**)
- 4) Watchdog időzítés (**WDT**) használata
- 5) Teljesítménykezelési módok (**Power Managed Modes**) használata: órajelforrások és ezek jelzőbitjei, a módba való belépés, illetve kilépés.

A Microchip ezt úgy valósította meg, hogy részben a meglévő kontrollereit áttervezte, részben az új eszközöket már ezen elvek alapján tervezte: a meglévő részegységek fo-

2. fejezet: A PIC mikrovezérlők felépítése, fejlődése

gyasztását csökkentette, új belső órajelforrást, új teljesítménykezelési módokat alakított ki. A továbbiakban elsődlegesen a PIC16 és PIC18-as családon illusztrálva tárgyaljuk a technikát, ami a 16 és 32 bites vezérlőknél csak a rendszersajátosságok miatt változik.

A 2.18. ábra a PIC18-as család órajelhez kapcsolódó rész blokkvázlatát mutatja. A PIC 16-os család ettől kis mértékben eltér: ott nincs IDLE mód. Vizsgáljuk most meg az egyes tulajdonságokat!

2.6.4.1. Órajel előállítása (*Clock System*)

Négy órajelforrás áll rendelkezésre:

- 1) Elsődleges (primary);
- 2) Másodlagos (secondary);
- 3) Belső oszcillátorblokk (8-16 MHz, osztással);
- 4) Különálló 31 kHz-es belső oszcillátor.

A két utóbbi esetén az OSCTUNE regiszter tartalmával állíthatjuk az órajel pontos frekvenciáját 6-bit, állítási lehetőség max. $\pm 12,5\%$, 0,4% / lépésekkel. Hatással van az INTOSC, az INTRC, a WDT frekvenciára.

1. Elsődleges (primary) órajel

A FOSC3:FOSC0 (`_CONFIG1H<3:0>`) bitek határozzák meg az elsődleges órajelforrást: 10 mód lehetséges: külső órajel : EC, ECIO, kristályoszcillátor – LP, XT, HS, HSPLL (ez utóbbi csak PIC18-as családnál), külső RC oszcillátor – RC, RCIO, belső oszcillátorblokk (IOB) – INTIO1, INTIO2. Az elsődleges oszcillátor az, amelyik RESET-kor vagy megszakítás után elindul.

- A kristály alkalmazó módok mindenkorban használják az oszcillátor késleltető számlálóját, ami 1024 oszcillátorciklus után engedi el a RESET jelet. A HSPLL mód még további késleltetést igényel, amíg a PLL stabilizálódik.
- A külső órajel mindenkorban változata (*External Clock [EC]*) órajelet vár az OSC1 lábon, míg az OSC2 lábon vagy az órajel negyede (*Fosc/4*) jelenik meg, vagy RA6 portkivezetésként használható.
- A belső RC oszcillátor használata előnyös. Nem igényel külső csatlakozást. Az OSC1 lábon RA7 portként használható, míg OSC2 kivezetésen vagy Fosc/4 frekvencia jelenik meg, vagy RA6 portkivezetésként használható. A frekvencia zárt hurok (FLL) a belső RC oszcillátor 31,25 kHz-es jelét fogadja, megszorozza 256-tal, ez lesz 8 MHz. Mikor az FLL működni kezd, kimeneti frekvenciája kb. 20%-kal nagyobb, mint a névleges érték. Kb. 1 ms időtartam szükséges, amíg stabilizálódik. Akkor válik stabillá, mikor az OSCCON regiszter IOFS bitje 1-re vált. Ezt akkor kell figyelembe venni, ha a futtatott program első pillanattól kezdődően időkritikus, mert akkor ezt a bitet induláskor figyelni kell.

2. Másodlagos (secondary) órajel

TMR1 időzítőhöz kapcsolódó oszcillátor, frekvenciája változik a tápfeszültség és a hőmérséklet függvényében. Árama: 20-30 μ A. Újra tervezett Timer1 oszcillátoron a fogyasztás csak 3 μ A, az amplitúdot szabályozzák, és nem függ a tápfeszültségtől, hőmérséklettől, megbízható működésű. T1OSCEN (`T1CON<3>`) 1-be állításával engedélyezhető, frekvenciája célszerűen 32 768 kHz (ennek 2^{15} -nel való osztása 1 Hz), a felhasználói programban kell figyelni, hogy a másodlagos órajel működöképes-e.

3. Belső oszcillátorblokk (IOB – Internal Oscillator Block) – INTOSC

8 MHz-re kalibrálva a pontosság 1-2% nagyságú 25 fokon. F típusokat 5 V-on, LF típusokat 3 V-on kalibrálják. Egy utóosztót vezérlő multiplexer választja ki a működtető 8, 4, 2, 1 MHz, 500, 250, 125 kHz vagy 31 kHz frekvenciájú órajelet.

A stabilizálódáshoz 1–4 ms időtartam szükséges induláskor, de közben már kódot hajt végre. IOFS jelzőbit (OSCCON<2>) 1-be állása jelzi, hogy az órajel már stabil. Ezt esetleg figyelni kell.

4. Belső RC oszcillátor – INTRC

31 kHz névleges frekvenciájú, 28–34 kHz között változhat, nem érzékeny a hőmérséklet- és tápfeszültség-változásra. Feléledési ideje nincs, azonnal működöképes. Ha ezt választjuk rendszerórajelnek, INTOSC kimenete, illetve utóosztója is tiltott lesz – kisebb a fogyasztás. Hol használjuk az INTRC-t?

- INTRC-t a WDT-t használja. Névleges értékek: PIC16 - $T_{WDT} = 1\text{ms}$ PIC18 - $T_{WDT} = 4\text{ms}$
- Kétsebességű indítási mód alkalmazása: Reset és éledés SLEEP módból.
- Órajel-meghibásodás figyelése.

Órajel-átkapcsolás

Az IRFC<2:0> bitek módosítása azonnal kiválasztja az új INTOSC frekvenciát. El kell olvasni az adatlapokat, mert a helyes működéshez szükséges a frekvencia/tápfeszültség megfelelő értéke. A változtatáskor a javasolt algoritmus: OSCCON beolvasása egy segédregiszterbe – segédregiszter módosítása – segéd visszairása OSCCON regiszterbe.

```
MOVF OSCCON, W      ; OSCCON WREG-BE MÁSOLÁSA
ANDLW B'10001111'   ; IRFC BITEK TÖRLÉSE
IORLW B'01010000'   ; ÚJ IRFC BITEK (PL. 2MHZ)
MOVWF OSCCON        ; EREDMÉNY VISSZAMÁSOLÁSA OSCCON REGISZTERBE
```

2.6.4.2. Kétsebességű indítási mód

Lehetővé teszi a programvégrehajtást, mielőtt az elsődleges órajel (kristály, esetleg PLL) aktív lesz. Ezt csak a belső oszcillátorblokk tudja biztosítani, és automatikusan átkapcsol, ha az elsődleges órajelforrás már működik. Jól használható, mikor a tok RESET-ből indul, vagy SLEEP módból ébred.

Az elsődleges órajel indulása késhet az indulási késleltetés miatt. Ennek oka:

- kristály/rezonátor indulási ideje,
- OST (Oscillator Start Timer) 1024 ciklusos késleltetése,
- PLL késleltetése (2 ms HS/PLL módban).

Ilyenkor az órajelet csak az IOB szolgáltathatja. Az IOB működési frekvenciát a programból határozhatjuk meg. Az is lehet, hogy a kód részlet végrehajtása után olyan gyorsan lépünk ismét SLEEP módba, hogy az elsődleges oszcillátor nem is kezd el működni.

Ha az elsődleges órajelforrás már működöképes, akkor ennek hatására átkapcsol az elsődleges órajelre, a tok OSTS (OSCCON<3>) bitje 1-be áll, IOB lekapcsolhat, de INTRC működhet tovább: kell a WDT működtetéséhez vagy a hibás órajel figyeléséhez (FSCM).

A kétsebességű indítási módhoz 1-be kell állítani az IESO (Internal External Switch Over) bitet (CONFIG2<1>)(PIC16), illetve (CONFIG1H<7>) (PIC18) esetén. Csak kristályalapú oszcillátorban működik, HS/PLL mód 2 ms-mal megnöveli a feléledési időt.

Milyen kristályfeléledési idők vannak? LP-nél 100 ms, XT-nél ms, HS-nél 10 μs nagyságrendű. HS/PLL-nél 2 ms. A CPU-nak 5–10 μs szükséges, míg elkezdi a program futtatását.

Néhány számszerű példa: Ébredjen a tok SLEEP módból. Az OSCCON regisztert SLEEP előtt már módosítottuk: 31 kHz LP módnál (ugyanaz, mint a használt kristály órajele); 4 MHz XT módban (ugyanaz, mint a használt kristály órajele); 8 MHz HS és HS/PLL módonban.

| | Kétsebességű indítás nélkül | Kétsebességű indítással |
|--|---|---|
| 32,7 kHz LP módban (30,5 μs , 1 TCY = 122 μs) | Kristályoszcillátor kb. 750 ms múlva indul, OST ehhez hozzáad 31,3 ms-ot ($1024 * 30,5 \mu\text{s} = 31,3 \text{ ms}$), ez összesen 781 ms, mielőtt utasításokat kezd végrehajtani. | Késleltetés csak a CPU feléledés 7 μs . 781 – 7 = 774 ms alatt 6000 utasítás hajtható végre, míg az órajel-átkapcsolás megtörténik. |
| 4 MHz XT mode (250 ns, 1 TCY = 1 μs) | Kristályoszcillátor kb. 1 ms múlva indul, OST ehhez hozzáad 256 μs -ot ($1024 * 0,25 \mu\text{s} = 256 \mu\text{s}$), ez összesen 1,26 ms, mielőtt utasításokat kezd végrehajtani. | Késleltetés csak a CPU feléledés 7 μs . 1,26 ms – 7 μs = 1,25 ms alatt 1250 utasítás hajtható végre, míg az órajel-átkapcsolás megtörténik. |
| 20 MHz HS mode (TOSC = 50 ns, 1 TCY = 200 ns) | Kristályoszcillátor kb. 10 us múlva indul, OST ehhez hozzáad 51,2 μs -ot ($1024 * 0,05 \mu\text{s} = 51,2 \mu\text{s}$), ez összesen 61,2 μs , mielőtt utasításokat kezd végrehajtani | Késleltetés csak a CPU feléledés 7 μs . 61,2 – 7 = 54,4 μs alatt 109 utasítás hajtható végre, míg az órajel-átkapcsolás megtörténik. |
| 40 MHz HS/PLL mode (25 ns , 1 TCY = 100 ns) | Kristályoszcillátor kb. 10 μs múlva indul, OST ehhez hozzáad 102 μs -ot ($1024 * 0,1 \mu\text{s} = 102 \mu\text{s}$). PLL éledés: 2 ms, ez összesen 2,1 ms, mielőtt utasításokat kezd végrehajtani. | Késleltetés csak a CPU feléledés 7 μs . 2,1 ms alatt 4200 utasítás hajtható végre, míg az órajel-átkapcsolás megtörténik. |

**2.6.4.3. Az órajel-meghibásodás figyelése
(Fail Safe Clock Monitor – FSCM)**

Célja: a külső órajelforrás megszűnésének a figyelése. (Csak IOB az egyetlen belső órajel-forrás.)

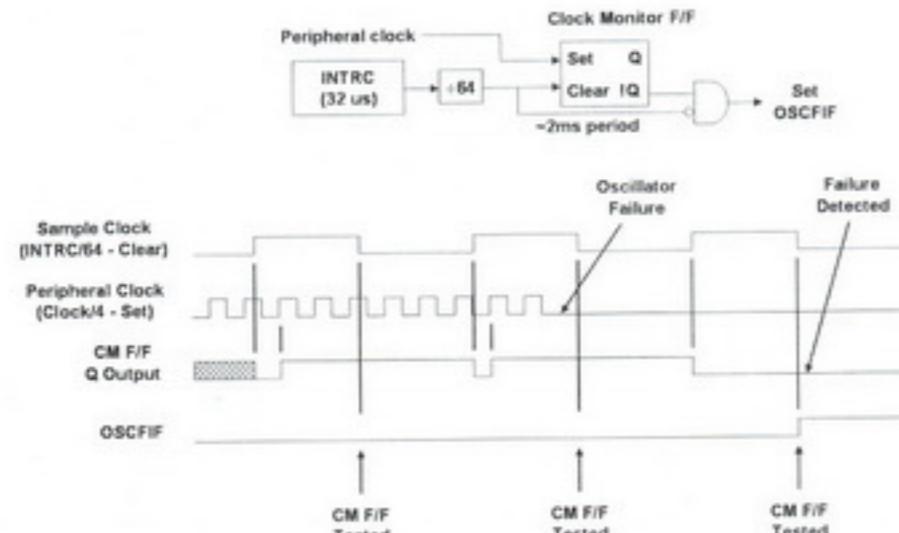
Engedélyezése: FSCMEN (Fail-Safe Clock Monitor Enable) bit (CONFIG2<0>) (PIC16-nál), illetve (CONFIG1H<6>) (PIC18-nál) 1-be állításával.

Működik mind az elsődleges, mind a másodlagos órajelforrás hibájánál: töri a WDT-t (nem muszáj engedélyezni), és az órajelforrás IOB lesz.

Ha elsődleges oszcillátor állt le, akkor OSTS bit törlődik. Másodlagos oszcillátorhiba esetén T1RUN (T1CON<6>) bit törlődik. OSCCON nem frissül, OSCFIF (PIR2<7>) bit (Oscillator Fail Interrupt Flag) 1-be áll. Ha a megszakítás engedélyezett, akkor az ébredés úgy történik, hogy az elsődleges órajelforrásra nem kapcsol át a tok.

Hiba esetén, ha belső RC a forrás:

- Ha IRFC bitek = 000, az órajel INTRC 31 kHz lesz.
- Ha IRFC bitek ≠ 000, az órajel INTOSC lesz, nagyobb sebességen. Az órajelhibát megelőzően az IRFC bitek beállíthatók, pl. induláskor.
- **RESET után vagy SLEEP-ből való ébredéskor:**
- **Hibamentes működésnél** IOB biztosítja az órajelet, amíg az elsődleges órajel rendelkezésre nem áll. (Hasonló a kétsebességű induláshoz.) Ha az elsődleges órajel aktív, automatikus órajelváltás választja ki az elsődleges órajelet. Hibás órajelfigyelés aktivizálódik, és OSTS bit 1-be áll, INTOSC tiltva lesz, INTRC (31 kHz) működik a hibás órajel detektálása és a WDT használata. Ez megakadályozza a hibás riasztást induláskor.
- **Hibás induláskor** elsődleges oszcillátor nem lesz aktív, órajelhibát nem lehet detektálni. Ezért programból kell ezt megoldani: OSTS ellenőrzése megfelelő késleltetés után, FSCM aktív lesz, ha valamelyik másodlagos módot választjuk, mielőtt az elsődleges aktív nem lesz.



2.19. ábra
Órajel-megszűnés detektálása

2.6.4.4. Watchdog időzítő (Watchdog Timer – WDT) áramkörök

Három típusuk van: általános, PIC16 WDT, PIC18 WDT. A belső RC oszcillátor INTRC (32 μ s periódusidő) biztosítja az órajelét. WDT-t programból engedélyezhetjük/tilthatjuk, a SWDTEN (WDTCON<0>) konfigurációs bit aktivizálásával.

8/14 PIC16 WDT:

- Timer0 és WDT az utóosztón osztozik. Az utóosztók kaszkádolhatók: osztás így: 1:1 - 1:8,4M (1ms tól 268 ms-ig [=4,5 perc]) között.
- PSA = 0 az utóosztót TMR0-hoz rendeljük, WDT csak a WDTCON utóosztót használja, osztás 1:32-től 1:63 536-ig (1 ms - 2,1 s). Alaphelyzet: 1:512 (16,4 ms).
- PSA = 1 az utóosztót WDT-hez rendeljük, WDT minden utóosztót használja. OPTION_REG vezérli az OPTION utóosztót, az osztás 1:1 - 1:128.

8/16 PIC18 WDT:

WDT utóosztó több osztást támogat: WDT periódusidő 4,0 ms ($125 * 32 \mu$ s), osztás: 1:1 - 1:32 768 (4 ms és 131 ms/2,2 perc) között. Az osztást fixen a CONFIG2H regiszterben adjuk meg.

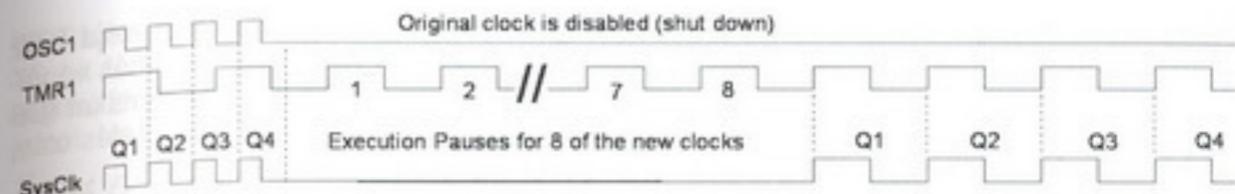
2.6.4.5. Teljesítménykezelési módok (Power Managed Modes)

Több lehetőség van az áramfogyasztás csökkentésére: órajelforrás és frekvencia megválasztása, CPU órajelének tiltása (PIC18-nál).

A legtöbb elsőleges, valamint a másodlagos órajel frekvenciája fix, nem változtatható, egyedül IOB használata teszi lehetővé programból az órajel-frekvencia változtatását. A másodlagos órajelforrás frekvenciája szintén fix.

Az órajelforrás kiválasztása SCS<1:0> bitek (OSCCON<1:0>) beállításával történik: PRI = 00 SEC = 01 INTOSC = 1X. Ezután PIC18 családnál egy SLEEP utasítás kerül végrehajtásra, míg PIC16 család esetén azonnal megtörténik az átkapcsolás (célszerű árnyékrezisztort használni).

Példa órajel átkapcsolására: Kapcsolunk XT (régi)-ról Timer1 (új)-ra. A végrehajtás a Q4 órajelciklus végénél kezdődik. Az új órajel 8 ciklusát megvárunk, és utána kezdődik az új órajellel a működés.



2.20. ábra
Órajel-átkapcsolás

Órajel-státusbitek: minden órajelforrás 1-be állít egy státusbitet, jelezve, hogy adja a rendszer órajelét. (Mindig csak egy státusbit lesz aktív.)

- Elsőleges órajelnél: OSTS bit (OSCCON<3>)
- Másodlagos órajelnél: T1RUN bit (T1CON<6>)
- IOB esetén: IOFS bit (OSCCON<2>)

Ha egyetlen státusbit sem 1, ez azt jelzi, hogy IOB adja az órajelet, de még nem stabil, vagy INTRC adja az órajelet (31 kHz). Annak eldöntésére, hogy vajon IOB az órajel forrása, megfelelő késleltetés után ellenőrizni kell IOFS bitet vagy IRFCF biteket.

- IOFS minden = 0, ha IRFCF<2:0> = 000 (31 kHz)
- IOFS bit 1 lesz az indulási késleltetés után, ha IRFCF<2:0> ≠ 000

Teljesítménykezelési módok áttekintése

Amint a 2.21. ábra táblázatában látható, PIC18 esetén hét, PIC16 esetén – mivel nincs IDLE mód – csupán négy különböző lehetőség van a teljesítmény kezelésére. Az áttekintésben először a közös módokról írnunk, majd utána az IDLE mód tárgyalása következik.

| Mode | CPU | Periph | IDLEN | SCS <1:0> |
|----------|---------|---------|-------|-----------|
| PRI_RUN | PRI CLK | PRI CLK | X | XX |
| SEC_RUN | T1OSC | T1OSC | 0 | 01 |
| RC_RUN | INTOSC | INTOSC | 0 | 1X |
| SLEEP | OFF | OFF | 0 | 00 |
| PRI_IDLE | OFF | PRI CLK | 1 | 00 |
| SEC_IDLE | OFF | T1OSC | 1 | 01 |
| RC_IDLE | OFF | INTRC | 1 | 1X |

2.21. ábra
Teljesítménykezelési módok

PRI_RUN mód: Ez a normál működési mód, ide tér vissza más teljesítménykezelési módból. A CPU-nak és a perifériáknak az elsőleges órajelforrás (PRI_CLK) adja az órajelet. T1RUN, IOFS bitek törölve vannak, OSTS bit: 1.

SLEEP mód: Ez a hagyományos SLEEP. IDLEN = 0 (PIC18-nál ezt kell beállítani), SCS<1:0> = 00. SLEEP utasítást hajtjuk végre utána. A CPU-nak és rendszerórajelet használó perifériáknak nincs órajele. Elsőleges oszcillátor tiltva van. Ez az egyetlen teljesítménykezelési mód, amikor egyetlen rendszerórajelekkel működik.

SEC_RUN Mode: Ez helyettesíti az órajel-átkapcsolást a PIC18 családnál. A perifériákat át kell állítani T1OSC frekvenciára. IDLEN = 0 (csak PIC18-nál), SCS<1:0> = 01. SLEEP utasítás végrehajtása (csak PIC18-nál). CPU-t, és a perifériákat T1OSC órajele működteti. Elsőleges oszcillátor tiltva. IOFS, OSTS bitek törlődnek, T1RUN bit 1-be áll.

RC_RUN Mode: A perifériákat át kell állítani az új frekvenciára. IRCF<2:0> választja ki az órajel-frekvenciát. IDLEN = 0 (csak PIC18-nál), SCS<1:0> = 1X. SLEEP utasítás végrehajtása (csak PIC18-nál). Elsödleges oszcillátor tiltva van, CPU-t és a perifériákat IOB órajele működteti. OSTS, T1RUN bitek törlődnek, IOFS = lesz 1–4 ms késleltetés után, ha Freq ≠ 31 kHz.

8/16 A következők csak a PIC18-as típusra vonatkoznak.

CPU-működés: A programból kikapcsolhatjuk a CPU órajelét. A program végrehajtása leáll, a perifériák működése folytatódik. A CPU SLEEP-hez hasonló állapotba kerül. WDT túlcordulásra, megszakításokra vagy RESET-re, a CPU újraindul. A CPU leállításához IDLEN bitet (OSCCON<7>) 1-be kell állítani, és utána kell a SLEEP utasítást végrehajtani.

PRI_IDLE mód: IDLEN = 1, SCS<1:0> = 00. SLEEP utasítás végrehajtása. CPU nem kap órajelet. Azok a perifériák, amelyek rendszerórajellel működnek, megkapják az elsödleges órajelet, az elsödleges órajel tovább működik.

SEC_IDLE mód: A perifériákat át kell állítani az új frekvenciára. IDLEN = 1, SCS<1:0> = 01. SLEEP utasítás végrehajtása. CPU nem kap órajelet. Azok a perifériák, amelyek rendszerórajellel működnek, megkapják az T1OSC órajelet. Az elsödleges órajel leáll. OSTS, IOFS bitek törlődnek, T1RUN bit értéke 1 lesz.

RC_IDLE Mode: IDLEN = 1, SCS<1:0> = 1X Perifériákat át kell állítani új frekvenciára. SLEEP utasítás végrehajtása. CPU nem kap órajelet. Azok a perifériák, amelyek rendszerórajellel működnek, megkapják az IOB órajelet. Az elsödleges órajel leáll. OSTS, T1RUN bitek törlődnek IOFS 1-be áll késleltetés után.

PIC 18 esetén (bármelyik) _RUN mód átkapcsolása (ugyanarra) az _IDLE módra.

```
; ILYENKOR UGYANAZT AZ ÓRAJELET HASZNÁLJÁK
    BSF    OSCCON, IDLEN ; IDLE MÓD VÁLASZTÁSA
    SLEEP          ; BELÉPÉS (*)_IDLE MÓDBA
; VÉGREHAJTÁS LEÁLL, ITT VÁRAKOZIK ÉBREDÉSRE, AMIT MEGSZAKÍTÁS,
; RESET VAGY WDT TÚLCORDULÁS OKOZHAT
; CPU NEM MŰKÖDIK, PERIFÉRIÁK UGYANAZT AZ ÓRAJELFORRÁST HASZNÁLJÁK.
```

Kilépés teljesítménykezelési módokból:

Mitől ébred SLEEP-ből a processzor (függetlenül a nanowatt technológiáról)?

- Megszakítás
- RESET
- WDT túlcordulás.

Megszakítás esetén:

A program működését, a GIE/GIEH megszakítást engedélyező bitek vezérik. Ha =1, akkor ugrás a megszakítást kiszolgáló rutinra, ha = 0, akkor a következő utasítás végrehajtása. Elsödleges órajel elindul, utána automatikus órajel-átkapcsolás akkor, ha az elsödleges órajel már működik.

Ha a megszakítás SLEEP módból kilépést okoz: Az órajel nincs kiválasztva, a program végrehajtás folytatódik, amikor az elsödleges oszcillátor már működik. Ha a kétsebességű indítás vagy az órajel-meghibásodás figyelése módok valamelyike be van kapcsolva, akkor a működés azonnal indul.

RESET esetén:

Az elsödleges órajel elindul. A végrehajtás addig várakozik, míg az elsödleges oszcillátor nem lesz stabil. Ha a kétsebességű indítás vagy az órajel-meghibásodás figyelése módok valamelyike be van kapcsolva, akkor a működés azonnal indul. OSTS = 1 lesz, amikor az elsödleges oszcillátor biztosítani tudja a rendszer órajelét.

2. fejezet: A PIC mikrovezérlők felépítése, fejlődése

WDT túlcordulás esetén:

TO bit törlése: PIC16-nál – STATUS<4>, PIC18-nál RCON<3>. Ha a CPU kódot hajtott végre (nem SLEEP vagy IDLE állapot), akkor RESET, CPU órajelet kap, és futtatja a programot. Ha SLEEP vagy IDLE módból ébred, akkor onnan folytatja a program végrehajtását.

2.6.4.6. Teljesítménykezelés a 16/24 mikrovezérlőknél

16/24 Alapvetően minden fontos megoldás hasonló, mint a PIC18-as mikrovezérlőknél. Üjdonság az is, hogy minden periféria parancsregiszterében van egy bit, ami azt határozza meg, hogy a periféria kapjon órajelet vagy nem. Az egyszerűbb kezelhetőség érdekében az IDLEN bit állításának szerepét két utasítás vette át:

```
PWRSAV #SLEEP_MODE      ; AZ ESZKÖZ SLEEP MÓDBA KERÜL
; A #SLEEP LITERAL ÉRTÉKE: 0
PWRSAV #IDLE_MODE       ; AZ ESZKÖZ IDLE MÓDBA KERÜL
; A #IDLE LITERAL ÉRTÉKE: 1
```

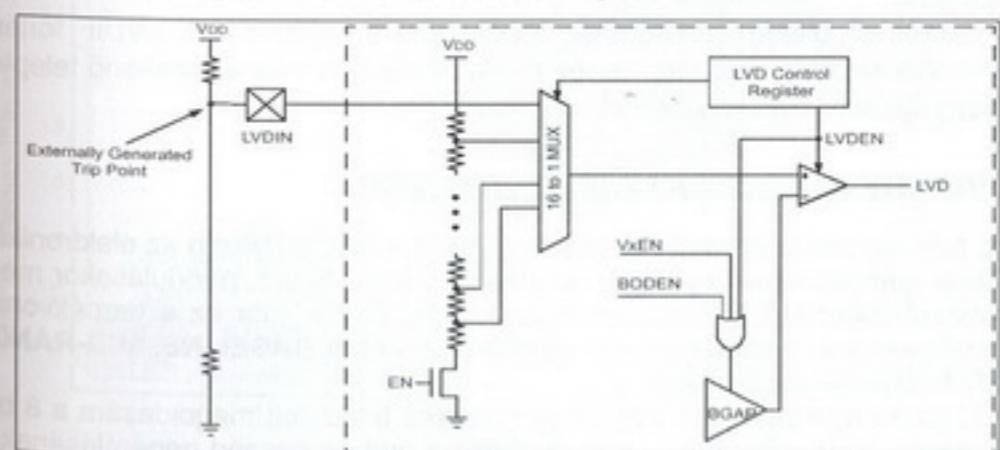
SLEEP módban minden belső órajel megszűnik, kivéve a WDT órajelét. Ha SLEEP előtt a WDT-t nem kapcsoljuk ki, akkor annak 32 kHz-es LPRC órajele tovább működteti a WDT-t. IDLE módban a CPU órajele kikapcsol, de a perifériák órajele aktív marad. Az órajel-generátorok azonban még egy további lehetőséget biztosítanak a fogyasztás szabályozásához, amit DOZE módnak hívunk. Mikor átkapcsolunk erre a módra, a perifériák folytatják a működésüket a rendszer órajele által meghatározott sebességgel, de a CPU-t kisebb sebességre kapcsolhatjuk. Az, hogy mennyivel lesz kisebb ez az órajel-frekvencia, az eredeti rendszerórajel leosztásával programozható 1, 2, 4 és 128 között. A DOZE és az IDLE mód, kiegészítve az egyedileg tiltható periféria-órajellel még hatékonyabb fogyasztásszabályozást eredményez. Ha a periféria működése nem szükséges, akkor az órajelét egyedileg kikapcsolhatjuk IDLE és DOZE módban is.

A PIC33F és PIC24H családonkál a DMA vezérlő is használható a fogyasztás csökkenésére IDLE mód esetén. Ugyanis ilyenkor aktív marad a csatlakozó perifériákkal együtt, és a perifériákkal való adatforgalom ilyenkor is folytatódhat, a CPU felébresztése nélkül. Blokkos átvitelkor, csak annak a végén kell a CPU-t felébreszteni, hogy elvégezze a blokk fel-dolgozását.

2.6.5. Alacsony tápfeszültség figyelése (LVD)

Ezt az áramkört (2.22. ábra) – bár perifériának is tekinthető – funkciója miatt célszerűbb a mikrovezérlöt kiegészítő rendszeráramkönek tekinteni.

Egyre több mikrovezérlöt tartalmazó rendszer telepes működésű.



2.22. ábra
LVD áramkör

A rendszernek figyelnie kell a telep állapotát, és ha a feszültsége egy minimális érték alá süllyed, akkor ezt a felhasználó felé jeleznie kell. A PIC18XXX családnál erre a célról egy külön periféria szolgál: az **alacsony feszültséget figyelő (Low Voltage Detect = LVD)** áramkör.

Az áramkör a tápfeszültség értékét figyeli: ha az elér egy előre beprogramozott küszöb-értéket, akkor a modul aktiválódik, megszakítást vált ki. Az általánosabb felhasználhatóság érdekében a modul feszültségsfigyelő pontja kívülről is (RA5 ponton) elérhető. Ha erre az LVDIN pontra egy külső feszültséget kötünk, akkor a periféria azt figyeli, és jelzést ad.

A gyakorlatban természetesen a küszöbértéket egy olyan minimális feszültségre célszerű állítani, aminél még a rendszer működik, hogy figyelmeztető jelzést tudjon adni.

Áramkörileg az egység egy komparátor áramkörből és egy digitális potenciometert tartalmazó feszültségesztóból áll. Az osztó táplálásához szükséges referenciafeszültséget a modul maga állítja elő.

A küszöbértéket 16 lépcsőben lehet változtatni 2,00 V és 4,77 V között.

A modul tápellátása külön bittel kapcsolható, amivel lehetséges a modult csak időnként ellenőrzésre használni. Ezt energiatakarékosnak miatt célszerű így kezelni.

LVD inicializálása elég összetett művelet, a következő lépésekkel áll:

- 1) Ha a külső LVD lábat használjuk (LVDIN), biztosítjuk, hogy minden, erre a lábra kapcsolódó (multiplexelt) periféria tiltva van, és a láb bemeneteket konfigurálva.
- 2) Írjuk az LVDL bitek értékeit az LVDCON (8 bit), illetve az RCON (16 bit) regiszterbe, kiválasztva a kívánt küszöbfeszültséget.
- 3) LVDIE = 0 legyen, ami tiltja az LVD megszakítást.
- 4) Engedélyezzük az LVD modult: LVDEN = 1.
- 5) Várunk meg, míg a belső referenciafeszültség stabilizálódik, az IRVST (8 bit), illetve BGST (16 bit) figyelésével (várunk, míg 1 nem lesz).
- 6) Feltétlenül töröljük LVDIF bitet a megszakítás engedélyezése előtt. Ha LVDIE = 1, akkor a tápfeszültségnak a küszöbfeszültség felett kell lennie.
- 7) Engedélyezzük az LVD megszakítást: LVDIE és GIE BITEK 1-be állítása (8 bit), illetve a prioritási szint beállítása LVDIP<2:0> bitekkel, majd LVDIE = 1.

FONTOS! Ha V_{DD} akár egy pillanatra is a küszöb alá esik, attól kezdve LVDIF = 1 lesz!

Két tevékenység végezhető az LVD megszakítás kiszolgálásakor:

- 1) LVDIE bit törlése, hogy ne legyen újabb megszakítás, és a kívánt tevékenység (pl. leállás) elvégzése;
- 2) vagy a küszöbfeszültség csökkentése LVDL bitek újraállításával, LVDIF törlése, hogy tovább működhessen az eszköz, és így nyomon követhessük a csökkenő telepfeszültséget (esetleg ilyenkor figyelmeztetéseket kiírva).

2.7. A PIC MIKROVEZÉRLÖK FEJLŐDÉSE

A Mikrochip folyamatosan fejleszti PIC mikrovezérlőit, ezzel is jelezve az elektronikai ipar és a verseny által igényelt tevékenységet, az innovációt. A Mikrochip indulásakor megjelent **8 bites mikrovezérlöket** is folyamatosan fejlesztették, és ma már ez a termékvonal három típusára terébelyesedett: a szokásos elnevezések szerint: a **BASELINE**, **MID-RANGE** és az **ENHANCED 8-BIT** mikrovezérlökre.

A fejlődés során nyilvánvalóvá vált, hogy bizonyos feladatok megoldására a 8 bites számítási teljesítmény nem elegendő, gondolunk itt az emberi beszéd generálásának és felismerésének, a villamos motorok szabályozásának feladatára, illetve a digitális jelfeldolgozás által támasztott igényekre.

2. fejezet: A PIC mikrovezérlök felépítése, fejlődése

Ezért a Mikrochip kifejlesztette a 16 bites adatokkal dolgozó családot, ami a 16 bites mikrovezérlővel egybeintegrált digitális jelfeldolgozó egységet is tartalmazhat. Ezek a típusok: a PIC24F és a nagyobb teljesítményű PIC24H család, amelyek digitális jelfeldolgozó egységet nem tartalmaznak, illetve a dsPIC30 és a nagyobb teljesítményű dsPIC33F család.

A fejlődés következő szintje a 32 bites adatokkal dolgozó PIC32-es termékvonal kialakítása volt. Ennek kifejlesztésekor processzormagként már nem egy saját fejlesztést használtak, hanem a MIPS cég M4K típusjelű, 32 bites mikroprocesszorát, és ezt vették körül a 16 bites mikrovezérlök perifériákészletevel.

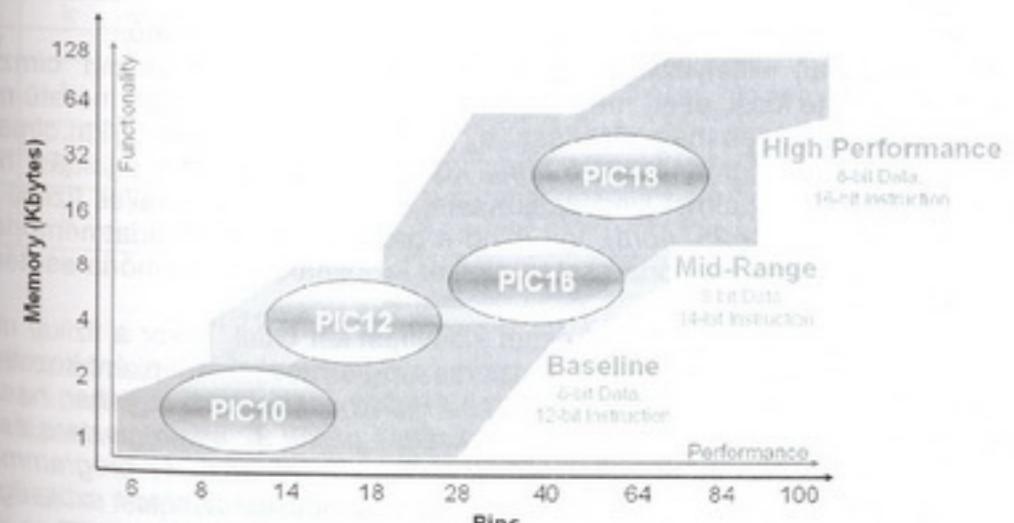
A következőkben röviden összefoglaljuk a Microchip mikrovezérlő termékvonalainak legfontosabb jellemzőit, amiből nyomon követhető a folyamatos fejlődés.

Sajnos a Microchip jelölésrendszer nem elég átgondolt: 8 bites mikrovezérlöknél az elnevezések: PIC10, PIC12, PIC16, PIC18.

A legnagyobb problémát a PIC16-os elnevezés jelentette, mivel emiatt a 16 bites mikrovezérlök PIC24, PIC30, PIC33 néven szerepelnek, míg a 32 bites mikrovezérlök neve PIC32. Vagyis az elnevezések nem köthetők valamilyen felépítési paraméterehez. A fejlődést a 2.23. ábra illusztrálja.

2.8. 8 BITES MIKROVEZÉRLÖ-CSALÁDKOK (8/12, 8/14, E8/14, 8/16)

Nagyon sok, mikrovezérlővel megoldható feladat nem igényel extra számítási teljesítményt. Ilyenkor a lábak száma, a program és az adatmemória mérete, valamit az eszközben lévő perifériák határozzák meg a választást. Ezért a Microchip három termékvonalat is kialakított a minél rugalmasabb tervezések érdekében. Érdekes és tanulságos, hogy a Microchip a régebbi típusarchitektúrákat áttervezéssel megtartotta, biztosítva ezzel a régebbi típusokkal való programkompatibilitást. A következőkben röviden áttekintjük a 8 bites családok legfontosabb jellemzőit.



2.24. ábra
8 bites PIC mikrovezérlök

2.8.1. PIC10F/PIC12F/PIC16F – 8/12-es család (Baseline Flash Microcontrollers)

8/12 Ide a következő típusjelzésű csoportok tartoznak:

- PIC10F csoport:** 6-pin PIC MCU/PIC10F2XX család. A tokok kivezetéseinek száma mindössze 6. 2,0–5,5 V feszültségtartományban működnek, és 4/8 MHz-es belső oszcillátort tartalmaznak a tokok.
- PIC12F csoport:** 8-pin PIC MCU/PIC12F5XX család. A tokok kivezetéseinek száma 8. 2,0–5,5 V feszültségtartományban működnek, és 4/8 MHz-es belső oszcillátort tartalmaznak a tokok.
- PIC16F csoport:** ≥14-pin PIC MCU/PIC16F5XX család. A tokok kivezetéseinek száma 14 vagy több. 2,0–5,5 V feszültségtartományban működnek, és max. 20 MHz-es külső illetve belső oszcillátort tartalmaznak a tokok.

Az utasításhossz 12 bit. Ebben a 12 bites mezőben kell elhelyezni a műveleti kódot, valamint az operandust is. Emlékeztetőül: az operandus alapvetően háromféle lehet:

- állandó (konstans,** a Microchip szóhasználatában: literál) érték,
- adatmemória-cím** (a Microchip szóhasználatában: fájlregiszter) egy regiszterének a címe, valamint
- programmemória-cím**, amivel megadhatjuk, hogy hol folytassuk az utasítások végrehajtását. Ezek a program utasításainak egymás utáni végrehajtását megváltoztató ugrási utasítások, mint a CALL és a GOTO.

A literál egy 8 bites adat, tehát így a műveleti kód hossza csak 4 bit lehet. Vagyis ha le foglalunk számára egy 4 bites csoportot, akkor a többi utasítás már nem kezdődhett ezzel a 4 bittel, a maradék tizenöt 4 bites kombinációt használhatjuk a többi utasítás kódolására.

Az utasításkészlet tervezésénél az adatmemória címzésére 5 bitet, míg a programmemória címéhez az utasításban csak 9 bitet használtak.

Ez utóbbi még erősebben korlátozta a különböző utasításcsoportok számát, hiszen ezenkívül minden 4 bites csoportot van, aminek felső 3 bitje különböző.

A tervezés során azonban további korlátokba ütköztek, ami miatt a CALL utasítás operandusa csak 4 bites lehetett. Ennek a következménye, hogy mivel a címnek 9 bitesnek kell lennie, az utasításdekóder minden 0-nak tekinti a CALL utasításban szereplő 9. bit értékét. Vagyis a szubrutinok belépési címeinek a programmemória-lap első 256 címén kell elhelyezkedniük!

Az utasításkészlet minden 33 utasításból áll, és a 2.25. ábrán látható.

Mivel az utasításban elhelyezkedő program-, illetve az adatmemoriát címző címoperandus mező mérete kicsi, ezért meg kellett oldani azt, hogy nagyobb méretű memória-ákat alkalmazhassunk. Ennek megvalósítása: az utasításban szereplő logikai címek kibővítése a tényleges fizikai memóriát elérő címekké. A legegyszerűbb megoldás, hogy az utasításban szereplő logikai címet kiegészítjük bitekkel, amivel már a teljes fizikai memória megcímzhetővé válik (2.25. ábra). Így mind a program, mind az adatmemória szegmensekből épül fel, programmemória esetén ezeket lapoknak, adatmemória esetén pedig bankoknak nevezzük.

Például ha a programmemória logikai címét kibővítjük két bittel, akkor a fizikai memória már 11 bittel címzhető, ami négy darab 512 szavas programmemória-lapként kezelhető.

A 8/12 bites felépítést a kis lábszámú PIC10 és PIC12 család több tagjában használják. Ugyanis a legnagyobb hátrányát – azt, hogy bankváltás nélkül 32 adatregisztere és lapváltás nélkül 512 memóriaszava lehet – úgy védték ki, hogy a tok adat- és programmemória mérete maximum ezeket a határokat éri el, vagyis programíráskor nincs szükség bank- illetve lapváltásra.

| Mne-monik | Ope-randum | Leírás | Cik-lus | 12 bites műv. kód | Státus-bitek |
|-----------|------------|--------------------------------------|---------|-------------------|--------------|
| ADDWF | f,d | w és f összeadása | 1 | 0001 11df ffff | C,DC,Z |
| ANDWF | f,d | w és f ÉS kapcsolata | 1 | 0001 01df ffff | Z |
| CLRF | f | f törlése | 1 | 0000 011f ffff | Z |
| CLRW | - | w törlése | 1 | 0000 0100 0000 | Z |
| COMF | f,d | f komplementálása | 1 | 0010 01df ffff | Z |
| DECW | f,d | f csökkentése 1-gyel | 1 | 0000 11df ffff | Z |
| DECFSZ | f,d | f csökkentése 1-gyel skip ha nulla | 1,2 | 0010 11df ffff | nem |
| INCF | f,d | f növelése 1-gyel | 1 | 0010 10df ffff | Z |
| INCFSZ | f,d | f növelése 1-gyel skip ha nulla | 1,2 | 0011 11df ffff | nem |
| IORWF | f,d | w és f megengedő VAGY kapcsolata | 1 | 0001 00df ffff | Z |
| MOVF | f,d | f mozgatása | 1 | 0010 00df ffff | Z |
| MOVWF | f | w mozgatása f-be | 1 | 0000 001f ffff | nem |
| NOP | - | nincs művelet | 1 | 0000 0000 0000 | nem |
| RLF | f,d | f forgatás balra carry-n keresztül | 1 | 0011 01df ffff | C |
| RRF | f,d | f forgatás jobbra carry-n keresztül | 1 | 0011 00df ffff | C |
| SUBWF | f,d | w kivonása f-ból | 1 | 0000 10df ffff | C,DC,Z |
| SWAPF | f,d | f felső és alsó 4 bitjének a cseréje | 1 | 0011 10df ffff | nem |
| XORWF | f,d | w és f kizáró VAGY kapcsolata | 1 | 0001 10df ffff | Z |

BITKEZELŐ MŰVELETEK

| | | | | | |
|-------|-----|---------------------------|-----|----------------|-----|
| BCF | f,b | bit törlése | 1 | 0100 bbbf ffff | nem |
| BSF | f,b | bit 1-be állítása (set) | 1 | 0101 bbbf ffff | nem |
| BTFSZ | f,b | bit vizsgálata skip, ha 0 | 1,2 | 0110 bbbf ffff | nem |
| BTFSZ | f,b | bit vizsgálata skip, ha 1 | 1,2 | 0111 bbbf ffff | nem |

ÁLLANDÓT (LITERÁLT) HASZNÁLÓ MŰVELETEK

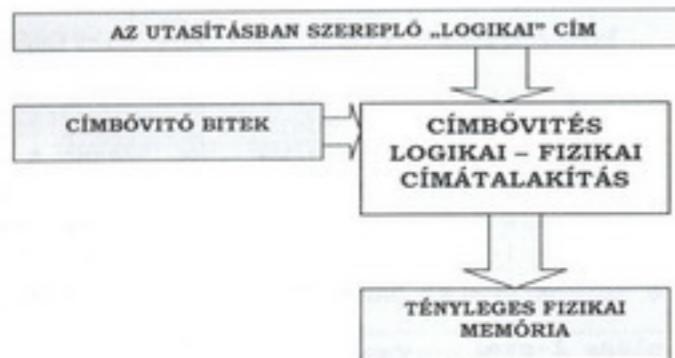
| | | | | | |
|--------|---|--------------------------------------|---|----------------|---------|
| ANDLW | k | állandó és w ÉS kapcsolata | 1 | 1110 kkkk kkkk | Z |
| CALL | k | szubrutinhívás (ugrás a szubrutinra) | 2 | 1001 kkkk kkkk | nem |
| CLRWDT | | WDT törlése | 1 | 0000 0000 0100 | /T0,/PD |
| GOTO | k | ugrás adott címre | 2 | 1010 kkkk kkkk | nem |
| IORLW | k | állandó és W megengedő VAGY kapcsol. | 1 | 1101 kkkk kkkk | Z |
| MOVlw | k | állandó W-be mozgatása | 1 | 1100 kkkk kkkk | nem |
| OPTION | - | OPTION regiszter törlése | 1 | 0000 0000 0010 | nem |
| RETLW | k | visszatérés, és k literál W-be írása | 2 | 1000 kkkk kkkk | nem |
| SLEEP | - | szundi üzemmód kiváltása | 1 | 0000 0000 0011 | /T0,/PD |
| TRIS | f | TRIS regiszter feltöltése | 1 | 0000 0000 0fff | nem |
| XORLW | k | állandó és w kizáró VAGY kapcsolata | 1 | 1111 kkkk kkkk | Z |

2.25. ábra

A 12 bites PIC mikrovezérlők utasításkészlete

Az ugrást tartalmazó utasítások két utasításciklust igényelnek. Rövidítések:
skip a következő utasítás átlépése f fájlregiszter d eredmény helyét jelző bit
/T0,/PD üzemmód bitek b bit sorszáma (0...7) k állandó (literal)

8/14 A 8/14 felépítés utasításkészlete ezzel csaknem azonos. Változás: megszünt az OPTION és TRIS utasítás. Új utasítások: ADDLW - állandó W-hez adása, SUBLW - W kiválasztása állandóból, RETFIE - visszatérés megszakításból, RETURN - visszatérés szubrutinból.



2.26. ábra
Címképzés

A 12 bites mikrovezérlők 6, 8, 14, 20 és 40 lábú kivezetéssel kaphatók, és a legfontosabb közös tulajdonságaiak:

- újraprogramozható flash programmemória,
- a CPU 33 utasítást képes végrehajtani,
- a kimenetek 25 mA-es áramot képesek nyelni, illetve kiadni,
- alvó (sleep) állapotban a tok fogyasztása mindössze 100 nA,
- egy darab 8 bites számlálója/időzítője van (TMR0),
- van egy watchdog időzítője,
- áramkörben programozható (*In Circuit Serial Programming – ICSP* képesség).
- nincs megszakítás**, tehát a perifériakezelés csak folyamatos figyeléssel (pollinggal) lehetséges.

| | | | | | | | |
|--|-------|--------|------|-----|--------|--------|-----|
| | 256 W | Page 0 | 000h | 00h | INDF | INDF | 20h |
| | 512 W | | 1FFh | 01h | TMR0 | TMR0 | 21h |
| | | Page 1 | 200h | 02h | PCL | PCL | 22h |
| | | | 3FFh | 03h | STATUS | STATUS | 23h |
| | | Page 2 | 400h | 04h | FSR | FSR | 24h |
| | 1 KW | | 5FFh | 05h | OSCCAL | OSCCAL | 25h |
| | | Page 3 | 600h | 06h | PORTB | PORTB | 26h |
| | | | 7FFh | 07h | GPR | GPR | 27h |
| | 2 KW | Page 4 | 800h | 08h | Common | Common | 28h |
| | | | 9FFh | 09h | GPR | GPR | 29h |
| | | Page 5 | A00h | 0Ah | | | |
| | | | BFFh | 0Bh | | | |
| | | Page 6 | C00h | 0Ch | Banked | Banked | 30h |
| | | | DFFh | 0Dh | GPR | GPR | 31h |
| | | Page 7 | E00h | 0Eh | | | |
| | | | FFFh | 0Fh | | | |
| | | | | | Bank 0 | Bank 1 | |

2.27. ábra
Baseline program- és adatmemória

Természetesen a közös tulajdonságok mellett még más jellemzők is meghatározzák a család tagjait, típusról függően:

- belső 4/8 MHz-es oszcillátor vagy külső oszcillátor,
- analóg komparátor,
- egy vagy több 8 bites A/D periféria,
- adat EEPROM

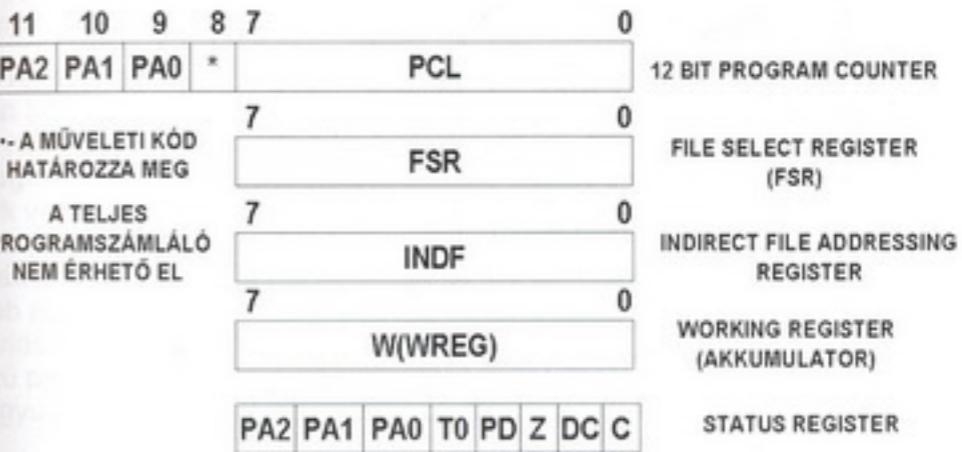
A program- és adatmemória felépítése a 2.27. ábrán látható.

2. fejezet: A PIC mikrovezérlők felépítése, fejlődése

Adatmemória: az utasításban szereplő operanduscím 5 bites, vagyis maximum 32 adatbájtot tudunk közvetlenül címezni. A címbővítéshez az FSR regiszter felső bitjeit használjuk fel. Az adatmemóriabankok első 16 bájtja azonos, vagyis a bankváltástól függetlenül ugyanazt a regiszter jelenti. Az első nyolc SFR regiszter, míg a másik nyolc általános célra használható. A következő 16 regiszter tartalma már a banktól függ. Vagyis összesen 8 bájt minden bankban azonos, és 8×16 bájt bankenként változó regiszterünk van, összesen 136 bájt.

Közvetlen (direkt) címzésnél a tényleges fizikai adatmemória-cím úgy alakul ki, hogy az utasításban szereplő 5 bites címhez hozzáillesztjük az FSR regiszter <7:5> bitjeit. Az így kialakuló cím már 8 bites, azaz 256 bájtot tud címezni, ami 8×32 bájtos bankokba van szervezve.

Közvetett (indirekt) címzésnél az FSR regiszter tartalma határozza meg azt a címet, amivel az utasításban szereplő műveletet el akarjuk végezni. Indirekt címzés esetén annak a regiszternek a címét, amelynek tartalmával a műveletet akarjuk végrehajtani, nem az utasítás tartalmazza, hanem egy regiszter, amelynek a neve FSR (*File Select Register*). Azért, hogy az utasításkészlet egyötötlégi maradjon, bevezettek egy áregiszter-jelölést, aminek a neve INDF (*INDirect File register*). Ha egy utasításban erre hivatkozunk, akkor valójában az FSR regiszterben lévő cím által meghatározott regiszter tartalmával végezzük a műveletet. Ez azt is jelenti, hogy ennek a regiszternek nem érdekes, hogy mi a címe, a művelet során ezt nem használjuk fel, csak a közvetett címzést jelöli, ezért ténylegesen nem is létezik! Ennek az áregiszternek a címe: 0.



2.28. ábra
Baseline programozói modell

Programmemória: a programmemória címe az utasításban csak 9 bites lehet. Ezzel 512*12 bites memória címezhető meg. A lapozást a STATUS regiszter legfelső <PA2:PA0> bitjeivel valósíthatjuk meg, Ezzel a három bittel összesen 4 kiszó fizikai memória használata lehetséges.

Verem: a szubrutinok 12 bites visszatérési címei vannak benne. Csupán kétszintű. Ez azt jelenti, hogy egy meghívott szubrutinban csak még egy szubrutinhívást helyezhetünk el.

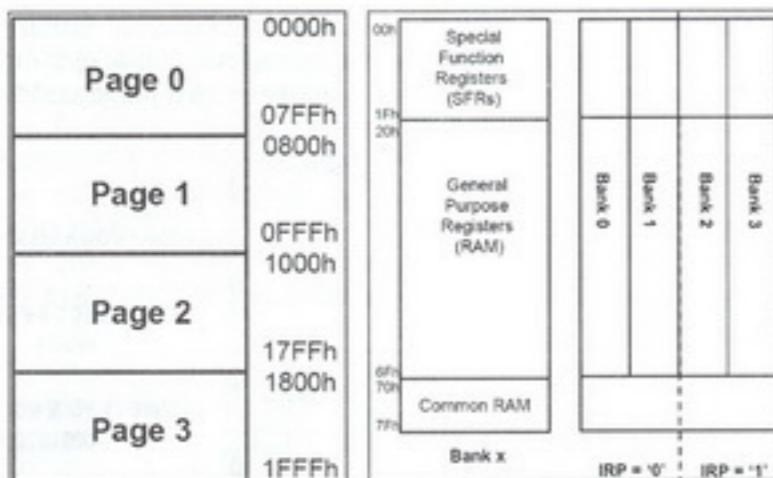
2.8.2. PIC12F/PIC16F – 8/14-es család (Mid Range Microcontrollers)

8/14 A PIC16-os mikrovezérlők 14 bites utasításhossza már több lehetőséget biztosít. A legfontosabb: az utasításban szereplő címek hossza 2 bittel megnövekedhetett. Az utasításkészlet mindenbőven két utasítással bővült, ami biztosította a lefelé kompatibilitást. Az OPTION és a TRIS utasítás megszünt, így az új utasítások: ADDLW, SUBLW, RETFIE, RETURN.

Adatmemória: az utasításban szereplő operanduscím 7 bites, vagyis maximum 128 adatbájtot tudunk közvetlenül címezni. A címbővítéshez a STATUS regiszter <RP1:RP0> illetve <IRP> bitjeit használjuk. Az adatmemóriabankok első 32 bájtja tartalmazza az SFR regisztereit, míg az utolsó 16 bájt azonos, vagyis a tartalma a bankváltástól független. A köztük lévő regiszterek tartalma már a banktól függ. Vagyis összesen maximum $1 \cdot 16 + 4 \cdot 80 = 336$ bájt áll a felhasználók rendelkezésére.

Közvetlen (direkt) címzésnél a tényleges fizikai adatmemória-cím úgy alakul, hogy az utasításban szereplő 7 bites címhez hozzáillesztjük a STATUS regiszter <RP1:RP0> bitjeit. Az így kialakuló cím már 9 bites, azaz 512 bájtot tud címezni, ami $4 \cdot 128$ bájtos bankba van szervezve.

Közvetett (indirekt) címzésnél az FSR regiszter tartalmához legfelső bitként még az IRP bitet kell illeszteni, hogy megkapjuk a teljes fizikai adatmemóriát elérő 9 címbet. Az így megcímzett regiszter tartalmát címnek tekintjük, és az ezen címen lévő tartalommal elvégzik az utasításban kijelölt műveletet.



2.29. ábra
Mid-range program- és adatmemória



2.30. ábra
Mid-range programozói modell

Programmemória: a programmemória címe az utasításban 11 bites lehet. Ezért 2048*14 bites (= 2 kiszó, kWWord) memória címezhető meg. Mivel az utasításszámláló 13 bites, ezért két bittel végezhetjük a memórialapozást. Ezzel összesen $4 \cdot 2$ kWWord = 8192*14 bites lehet a fizikai memória mérete. Az utasításmutató (PC) két részből áll: a műveletekben SFR regiszterként szereplő 8 bites PCL regiszterből, valamint a PC felső 5 bitjét tartalmazó

PCH regiszterből. Ez utóbbi programból nem érhető el. Normál utasítás-végrehajtásnál PC normális 13 bites számlálóként viselkedik. Két esetben PCH értéke PCLATH regiszter tartalmával töltödik fel:

- 1) Ha PCL-lel műveletet végzünk, akkor PCH értéke PCLATH tartalma lesz. Ez a RETLW utasítással együtt, táblázatok kezelésénél előnyös.
- 2) A két ugró utasítás a GOTO és a CALL végrehajtásakor PCLATH<4:3> bitje PCH<12:11> bitje helyére íródik, ilyen módon valósítva meg a lapozást.

Verem: 8 szintű, és a 13 bites utasításszámláló tartalmát tárolja. Ez azt jelenti, hogy az egymásba skatulyázott szubrutinok száma maximum 8 lehet.

Megjelent a megszakítás. A megszakítást kiszolgáló alprogramot az új, RETFIE utasítás-sal kell befejezni. A hozzá kapcsolódó SFR regiszter az INTCON.

Ebben a csoportban néhány PIC12-es termék vonalba sorolt áramkör is megtalálható, természetesen 14 bites utasításhosszal. A tokok lábszáma 8–64 kivezetés között változik.

Rendelkezésre állnak nagyobb lábszám esetén az újra programozható flash, illetve az egyszer programozható (One Time Programming – OTP) kivitelű áramkörök. Flash megoldásnál a működési feszültség 2,0–5,5 V között változhat. Perifériáként többcsatornás A/D, illetve a tokban elhelyezett adat EEPROM is megtalálható. Digitális perifériák: CCP/PWM, USB, SPI, I²C™, USART, LCD is vannak a különböző családelemekben.

2.8.3. PIC12F/PIC16F – e8/14-es család (Enhanced Mid Range Microcontrollers)

e8/14 A 14 bites felépítés nagy sikere és széles körű elterjedése miatt a Microchip – jóllehet, közben kifejlesztette a PIC18 családot – úgy határozott, hogy kissé áttervezi a 14 bites felépítést, megtartva a legfontosabb tulajdonságait, és beépíti a PIC18 család bizonyos sikeres megoldásait. Az utasításkódok fel nem használt kombinációihoz újabb utasításokat rendelt. Mik voltak a tervezési célok?

- Nagyobb programmemória kialakítása
- Perifériák kibővítése
- Nagyobb adatmemória
- A lapváltás/bankváltás szerepének csökkentése
- C nyelvű programok hatékonyabb fordítása a futtatható gépi kódra
- Minél egyszerűbb áttérés

Legegyszerűbb, ha egy táblázatban (2.31. ábra) összehasonlítjuk a három család jellemzőit.

| Eszköz Tulajdonság | PIC16 | PIC16 enhanced | PIC18 |
|------------------------|---|-----------------------------------|-------------------------------------|
| Max. GPR/SFR | 336/110 | 2496/316 | 4096/159+SFR növelhető GPR rovására |
| Max. program | 8K*14 | 32K*14 | 1M*16 |
| FSR-ek | 1 | 2 pr.mem is elérhető | 3 |
| Utasítások száma | 35 | 48 | 75, bővítvé: 83 |
| Verem | 8 | 16 alul-, ill. túlcordulás jelzés | 31 alul-, ill. túlcordulás jelzés |
| Megszakítás | 1 | 1 HW reg. mentés minden | 2 HW reg. mentés, választható |
| Programmemória-olvasás | RETLW utasítás néhány tok EEPROM illesztésén át | RETLW, FSR EEPROM illesztésén át | TABLRD |

2.31. ábra
Összehasonlítás: PIC16 – ePIC16 – PIC18

A program- és adatmemória felépítése a 2.32. ábrán látható.

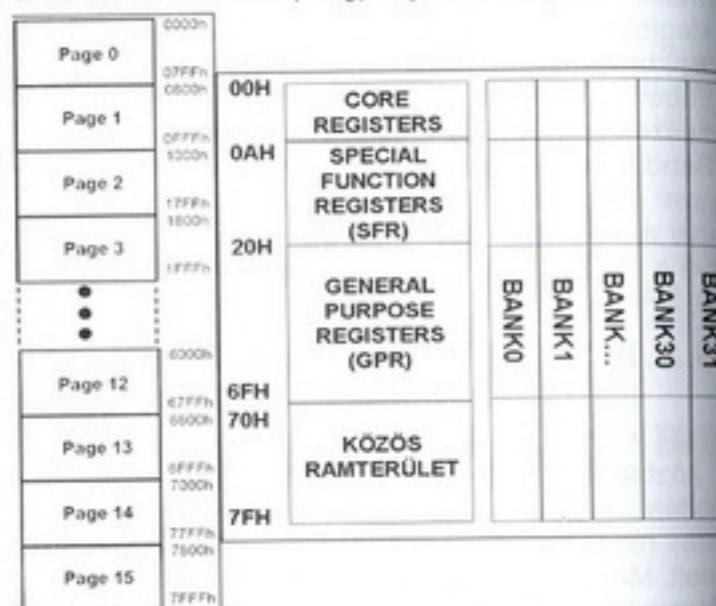
Adatmemória: a STATUS regiszterbeli <RP1:RP0> bitek helyett megjelent az ötbites BSR regiszter, ami 32 darab 128 bájtos adatmemória-bank használatát tette lehetővé.

Közvetlen címzésnél a fizikai cím BSR tartalma hozzáillesztve az az utasításban szereplő 7 bites címhez.

Közvetett címzésnél: az FSR0, FSR1 regiszterek tartalma határozza meg a címet. Mivel ezek a regiszterek a szükséges 12 bit helyett 15 bitesek, ezért a tartalmát programmemória-címként kezelve bármelyik programmemória-hely is elérhető. Egy adatmemória-bank négy területet tartalmaz: 0x00-0x0B címen a 12 darab ún. core (mag) regiszter helyezkedik el.

Ezek minden bankban azonosak. A 0x0C-0x1F címtartomány a perifériák SFR területe, bankról bankra változik. Az utolsó 0X70-0X7F című 16 bájt azonos, vagyis a tartalma a bankváltástól független. A 0x20-0x6F területen lévő bájtok tartalma minden bankban más. A GPR regiszterek száma ezért: $16+31*80 = 2496$ bájt. (Az utolsó, a 31. bank speciális.)

2.32. ábra
Enhanced PIC16 program- és adatmemória



A 2.33. ábrán látható táblázat a rendszerregisztereket mutatja. Már mindegyik ismerős, az ÚJ oszlopban láthatók a frissen bevezetettek. Érdekes, hogy az eddig külön címmel nem rendelkező W regiszter is megjelenik közöttük.

| ÚJ | MENT | CÍM | REG. | FUNKCIÓ |
|----|------|------|--------|----------------------------------|
| | | 0X00 | INDF0 | Indirect Register 0 |
| X | | 0X01 | INDF1 | Indirect Register 1 |
| | | 0X02 | PCL | Program Counter Low |
| X | | 0X03 | STATUS | Status Register |
| X | | 0X04 | FSR0L | File Select Register 0 Low Byte |
| X | X | 0X05 | FSR0H | File Select Register 0 High Byte |
| X | X | 0X06 | FSR1L | File Select Register 1 Low Byte |
| X | X | 0X07 | FSR1H | File Select Register 1 High Byte |
| X | X | 0X08 | BSR | Bank Select Register |
| X | X | 0X09 | WREG | Working Register |
| X | | 0X0A | PCLATH | Program Counter Latch High |
| X | | 0X0B | INTCON | Interrupt Control Register |

2.33. ábra
Core regiszterek

2. fejezet: A PIC mikrovezérlők felépítése, fejlődése

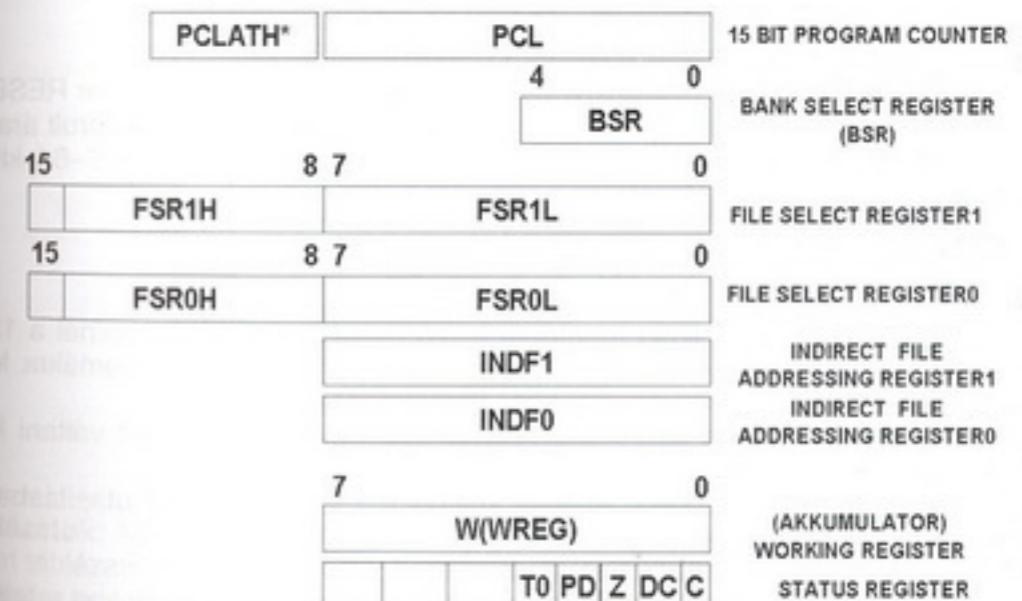
A megszakításnál fontos újdonság, hogy a legfontosabb regiszterek a megszakítás ki- szolgálása előtt minden 31. bankból elérhető árnyékregiszterekbe elmentésre kerülnek, majd a RETFIE utasítás végrehajtásakor visszatölödnek, ilyen módon őrizve meg a tartalmukat (MENT oszlop).

Programmemória: a programmemória címe az utasításban továbbra is 11 bites. Ezzel 2048*14 bites (= 2 kiszó, kWord) memória címezhető meg. Mivel az utasításszámítás 15 bitre növelték, ezért 4 bittel végezhetjük a memórialapozást. Ezzel összesen 16*2 kWord = 32768*14 bit lehet a fizikai memória mérete. Az utasításmutató (PC) két részből áll: a műveletekben SFR regiszterként szereplő 8 bites PCL regiszterből, valamint a PC felső 7 bitjét tartalmazó PCH regiszterből. Ez utóbbit programból nem érhető el. Normál utasítás-végrehajtásnál a PC normális 15 bites számlálóként viselkedik. Két esetben az értéke PCLATH regiszter tartalmával töltödik fel.

| | |
|-------------|------------|
| STATUS | FSR0 Low |
| STATUS_SHAD | FSR0L_SHAD |
| FSR0 High | FSR1 Low |
| FSR0H_SHAD | FSR1L_SHAD |

| | |
|------------|-------------|
| FSR1 High | BSR |
| FSR1H_SHAD | BSR_SHAD |
| WREG | PCLATH |
| WREG_SHAD | PCLATH_SHAD |

2.34. ábra
Árnyékregiszterek



PIC16F1XXX PROGRAMOZÓI MODELL

2.35. ábra
Enhanced PIC16 programozói modell

- Ha PCL-lel műveletet végzünk, akkor PCH értéke PCLATH tartalma lesz. Ez a RETLW utasítással együtt táblázatok kezelésénél előnyös.
- A két ugró utasítás a GOTO és a CALL végrehajtásakor PCLATH<6:3> bitje PCH<14:11> bitje helyére íródik, ilyen módon valósítva meg a lapozást.
Verem: 16 szintű, és a 15 bites utasításszámítás tartalmát tárolja. Ez azt jelenti, hogy az egymásba skatulyázott szubrutinok száma maximum 16 lehet.

Új utasítások: az eredeti utasításkészletben fel nem használt bitcsoportok lehetőségeit adtak 14 új utasítás bevezetésére. Ezek:

- ADDWFC** – Add W + F with Carry – összeadás Carry figyelembevételével;
- SUBWFB** – Subtract F – W with Borrow – kivonás kölcsönvét figyelembevételével;
- LSLF** – Logical Shift Left – tolás balra; MSB Carry-be kerül, LSB-be nulla bit kerül; ugyanaz, mint az aritmetikai balra tolás (MSB – legfelső bit LSB – legalsó bit);
- LSRF** – Logical Shift Right – tolás jobbra; MSB nulla lesz, LSB a Carry-be kerül;
- ASRF** – Arithmetic Shift Right – tolás jobbra, előjel-kiterjesztés MSB-be kerül, LSB Carry-be kerül;
- MOVLP** – Move Literal to PCLATH – 7 bites literál kerül PCLATH-ba, MOVLP+CALL/GOTO: 3 ciklus, két utasítás, MOVL CÍMKE ugyanaz, mint PAGESEL CÍMKE
- MOVLB** – Move Literal to BSR – 5 bites literál kerül BSR regiszterbe, helyette BANKSEL, IRP, RP0, RP1 megszűnt;
- BRA** – Branch Relative (signed) – relatív ugrás BRA N. Ugrás PC+N címre. – $256 \leq N \leq 255$. PC+N eredménye minden 15 bites marad;
- BRW** – Branch PC + W (unsigned) – relatív ugrás PC+W címre, W csak pozitív. PC+W minden 15 bit marad;
- CALLW** – Call PCLATH:W – szubrutinhívás PCLATH:W címre;
- ADDFSR** – Add Literal to FSRn (signed) – előjeles literál FSR-hez adása, literál -32 és +31 között van;
- MOVIW** – Move Indirect to W – [FSRn] >> WREG - Pre/Post Increment, Decrement, Relatív Ofszet módosítókat is lehet használni;
- MOVWI** – Move W to Indirect – WREG >> [FSRn];
- RESET** – Reset Hardware and Software – Hatása megegyezik a hardver RESET-tel.

Ebben a termékvonallal néhány, a kis lábszámú PIC12 termékvonallal sorolt áramkörök megtalálhatók, természetesen 14 bites utasításhosszal. A tokok lábszáma 8–64 kivezetés között változik.

2.8.4. PIC18F – 8/16-os család (High Performance Microcontrollers)

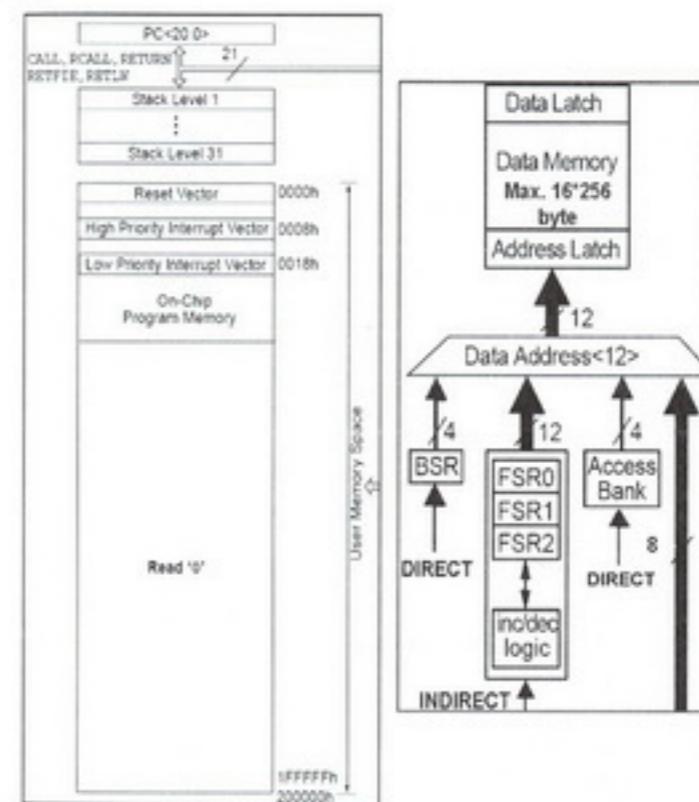
8/16 A PIC18F (F a használt flash memóriát jelöli) termékvonallal tervezésénél a 12 és 14 bit utasításszélességű mikrovezérlők felhasználása során megjelent problémákra kívántak megoldást találni. Mik voltak a 14 bites felépítés problémái?

- Az egyik legnagyobb gondot a regiszterbankok okozzák. Mivel ezeket váltani kell, ez könnyen el lehet téveszteni.
- A programmemória tartalmát nem lehetett kiolvasni, csak a RETLW utasításban tárolt adatokat.
- Az ablakos tokokkal a fejlesztés kellemetlen, nehézkes.
- A 0-dik adatmemória-címen kezdődő SFR regiszterek felett helyezkedtek el a felhasználói adatregiszterek, amelyeknek kezdőcímé típusról típusra változik, új perifériák megjelenése miatt vagy a felhasználói adatmemória kezdőcímét kell megnövelni, vagy más bankba kell a perifériához kapcsolódó regisztereket elhelyezni.
- A verem mélysége mindenkor 8.
- A maximum 2 kiloszavas memóriaméret miatt nagyobb programoknál a memóriát lapozni kellett.
- Az utasításkészlet szegényes, egyszerű utasítások, például nincs szorzás.

A problémák megoldására fejlesztették ki a valóban korszerű PIC18-as mikrovezérlőket.

Jelenleg a PIC18 termékvonallal már három termékcsaládra bomlik:

- a „klasszikus” **PIC18F család**: tápfeszültsége 2,0–5,5 V között lehet, órajele maximum 40 MHz, teljesítménye 10 Mips, flash programmemoriája százezer újraírást tesz lehetővé, és van lapkára integrált adat EEPROM memóriája.
- **PIC18FJ család**: tápfeszültsége 2,0–3,6 V között lehet, órajele maximum 40/48MHz, teljesítménye 12 Mips 3 V-os tápfeszültségnél. A programmemória újraíratósága 1000–10 000 között van, és a programmemoriával emulált EEPROM-ot tartalmaz. Ha több mint 32 kb által programmemoriát igényel az alkalmazás, akkor ez a legolcsóbb megoldás.
- **PIC18FK család**: ez a legújabb család. Tápfeszültsége 1,8–3,6 V/5,5 V lehet, órajele maximum 64 MHz.



2.36. ábra
PIC18 program- és adatmemória

Táblázatok: Mivel az információtárolókat címezhető regiszterekből alakítjuk ki, az ilyen adatokat táblázatoknak tekinthetjük. A táblázat **címezhető elemek rendezett halmaza**. A táblázat elemeire egy mutatóval (*pointer*) hivatkozunk, ez lényegében egy cím, amely meghatározza a táblázat kiválasztott, aktuális elemét. A címzés legtöbbször lineáris, és egyesével változik. A mutatóval kiválasztott (megcímzett) elemet vagy kiolvassuk, vagy tartalmát módosítjuk (írjuk). Mivel ezek az elemműveletek a legtöbbször szomszédos elemek egymásutánjára vonatkoznak, ezért célszerű a mutatót módosító, ún. mutatóműveletek bevezetése:

- **INC** – a mutató tartalmának növelése 1-gyel,
- **DEC** – a mutató tartalmának csökkentése 1-gyel. Az írás/olvasási elemműveletek, illetve a mutatóműveletek egymás utáni sorrendje is kétfajta lehet:
- **PRE** – előbb a mutatóművelet, utána az elemművelet,
- **POST** – előbb az elemművelet, utána a mutatóművelet.

Hol találkoznak a táblázatokkal? A PIC mikrovezérlök felépítése is jobban érhető a fenti alapján, mivel a memóriák minden táblázatnak tekinthetők:

- Utasítások a programmemoriában.** Itt a mutató a programszámláló, a PC. Utasításelhívás után az értéke növekszik. Az ugró utasítás valójában ennek a mutatónak (PC új értéke = ugrási cím) új értékét ad.
- Verem:** a verem is tábla, egy elemét veremmutatóval jelöljük ki: **STKPTR** értékét a rutin hívás vagy megszakítás változtatja.
- Fix adatok tárolása a programmemoriában.** TBLPTR – a táblapointer a mutató, amely egy adatot választ ki a programmemoriában.
- Az adatmemória kezelése** az FSR mutatókkal történik.

Programokban a mutató a paraméterek átadásánál nagyon hatékony. Egy mutató értékének az átadása egy teljes adatstruktúrát azonosít!

A következőben bemutatjuk a család legfontosabb jellemzőit. Az új, 16 bites utasításokkal dolgozó processzormag a szilícium lapkán mindenki 10%-kal több helyet foglal el, mint a széles körben elterjedt 14 bites architektúra processzormagja. Mivel a perifériák alapjában nem változtak, a lapka felülete az új családnál nem növekedett, hanem inkább csökkent, az új, finomabb rajzolatú, csökkentett vonalvastagságú technológiák következtében. Számos új változtatást hajtottak végre az architektúra és a leszűrődött programozási-alkalmazási tapasztalatok figyelembevételével. A család négy memóriatípus tartalmaz:

- A 21 bittel címezhető, 16 bit széles **programmemoriát**, ahol a végrehajtandó utasítások illetve állandó adatok tárolódnak; a programmémória írás/olvasás ciklusainak száma kb. 100 000,
- 12 bittel címezhető, maximum 4096 bájtos fájlregisztertömb **RAM adatmemoriát**, az adatok tárolására,
- 31 visszatérési címet tároló **veremmemoriát**, a megszakítások és a szubrutinhívások kezelésére,
- a maximum 1024 bájtos, bájtonként írható/olvasható **EEPROM adatmemoriát** adatok kikapcsolás utáni tárolására; az EEPROM adatmemória írás/olvasás ciklusainak száma kb. egymillió.

Megjelent egy 8*8 bites hardverszorzó. Ilyen módon az aritmetikai műveletek gyorsasága jelentősen megnövelte, mivel egy ciklus alatt hajt végre egy szorzási utasítást.

Előtérbe került az energiával történő takarékoskodás. Többfajta választható kis fogyasztású üzemmód, valamit a processzor órajelének frekvenciája működés közben átkapcsolható. Ha kell, két oszcillátor használhatunk, egy nagy sebességet az aktív működéshez és egy kis sebességet az energiatakarékos üzemmódban.

Fenntartották az előző típusokkal való látványosságot. Ez könnyű áttérést biztosít az újabb típusra.

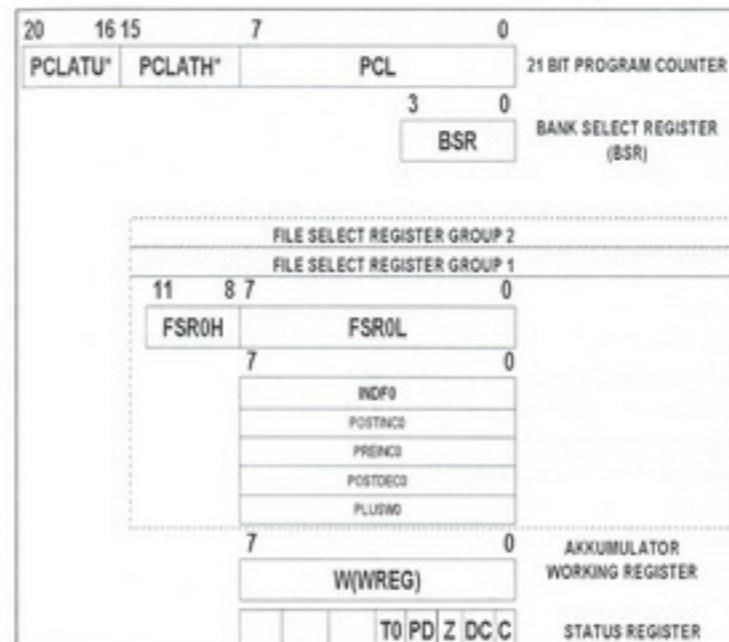
Adatmemória: A 16 bites utasításokban az operandusregiszter címe maximum 8 bit lehet, ami 256 regiszter címezését biztosítja. A maximum 4096 bájtos (4 kb) fizikai adatmemória kezeléséhez 12 bites cím szükséges, ezért ezt 16*256 bájtos bankba kell szervezni. Az adatmemória közvetett címezését három 12 bites FSR regiszterrel is végezhetjük.

Az indirekt címezést biztosító FSR regiszter változása jó példa a családok folyamatos fejlesztésére:

- A 12 bites családnál az FSR a bankkezelést és a bankon belüli címezést biztosította.
- A 14 bites (PIC16XXX) családnál az FSR egy önálló 8 bites regiszterré vált, amivel 256 bájtos RAM-területet lehetett címezni.
- A 16 bites, ma már igen ritkán használt, elavult PIC17XXX családnál már figyelembe vették azt a gyakorlati tapasztalatot, hogy az FSR-rel általában sorban egymás utáni adatmemória-regisztereit címezünk, ezért az FSR kezelése három módon lehetséges: az FSR

2. fejezet: A PIC mikrovezérlök felépítése, fejlődése

címezést használó művelet után az FSR tartalma nem változik, eggyel nő, illetve eggyel csökken. (Az ALUSTA regiszterben vannak a módválasztó bitek.) A címzéshez már két regisztert – FSR0, FSR1 – használhatunk, amivel könnyebben végezhetünk blokkmozgatást (forrás → cél).



2.37. ábra
PIC18 programozói modell

Közvetett címezés: A PIC18FXXX-es családnál az FSR regiszterek száma három, és 12 bites hosszukkal a maximális 4096 bájtos adatterület bankválasztás nélkül címezhető. Indirekt címzés esetén a tényleges műveletet a 12 bites FSRx regiszterben lévő tartalom által meghatározott című regiszterrel végezzük el. A PIC18F családjában ez három irányban bővült:

- Mivel az indirekt címezés felhasználásával könnyen tudunk folytonos adatmemória-területeket címezni, ezért adatblokkok mozgatására jól használható speciális utasításokat alakítottak ki.
- A másik hatékony megoldás az, hogy három, egymástól függetlenül használható FSR regiszter létezik: FSR0, FSR1, FSR2. A hozzájuk tartozó indirekt címregiszter: INDF0, INDF1, INDF2. A sorszámukat (hogy melyik a három közül: 0, 1 vagy 2) a továbbiakban „n”-nel jelöljük.
- Mivel az indirekt címezést ciklikus tevékenységek végrehajtására használják, ezért az FSR regiszterben adott című regiszter elérése után FSR tartalmát növelni vagy csökkenteni kell. Ezért célszerű olyan utasításokat megvalósítani, amik FSR tartalmát automatikusan változtatják (jelölés: [REG] – REG tartalma):

| | | |
|----------|------------------------|--|
| INDFn | [FSRn] = [FSRn] | FSRn tartalma nem változik |
| POSTINCn | [FSRn] = [FSRn+1] | az utasítás végrehajtása után FSRn tartalma 1-vel nő |
| POSTDECn | [FSRn] = [FSRn-1] | az utasítás végrehajtása után FSRn tartalma 1-vel csökken |
| PREINCn | [FSRn] = [FSRn-1] | az utasítás végrehajtása előtt FSRn tartalma 1-vel csökken |
| PLUSWn | [FSRn] = [FSRn+[WREG]] | az utasítás végrehajtása előtt FSRn tartalmához a WREG tartalmát adjuk hozzá |

Pl. ha egy utasításban a **POSTINCn** regisztert szerepeltetjük, akkor miután az indirekt címzéssel kiolvastuk a megcímzett regiszter tartalmát, és az utasításban szereplő műveletet elvégeztük, utána a hozzá tartozó FSRx regiszter tartalmát 1-gyel megnöveljük (inkrementáljuk).

LFSR 2,0X3AB

MOVWF POSTINC,2

;FSR2 TARTALMA 3ABH LESZ ([FSR2]=3ABH)

;[W]=[3AB] ÉS [FSR2]=3ACH LESZ

Közvetlen címzés: A regisztercímeket tartalmazó utasításokban egy 8 bites rész tartalmazza annak a regiszternek a címét, amivel az adott műveletet el kell végezni. Mivel a regisztertartomány fizikailag 12 bittel címezhető, meg kell találni azt a megoldást, hogyan képezzük le az utasításban szereplő 8 bites „logikai” címet a létező 12 bites „fizikai” címre. Ezért kétfajta adatmemória használatát tették lehetővé:

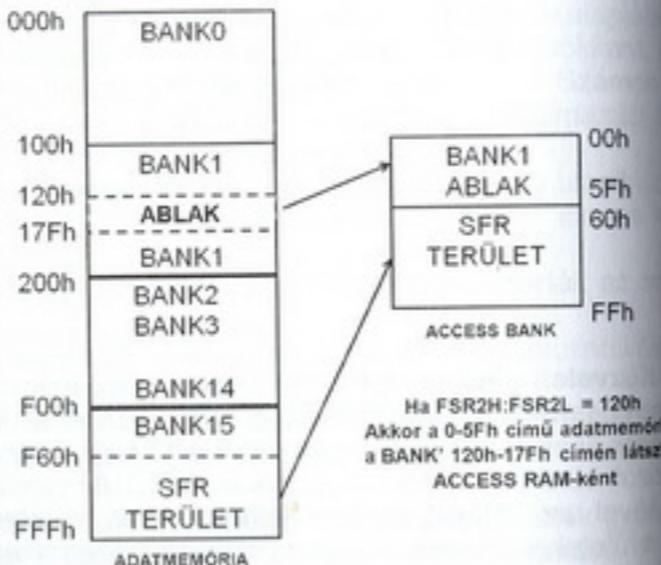
- Az egyik megoldásként kialakítottak egy 4 bites bankválasztó regisztert (**Bank Select Register – BSR**). A 12 bites tényleges fizikai cím alsó 8 bitjét az utasításban lévő címbeiktet, a maradék felső 4 bitjét a BSR tartalma adja. Így 16 darab, egyenként 256 bájtos memóriabank van kialakítva. **A bankváltások elmaradása, illetve hibás használata sok rejtélyes hiba forrása lehet!**

- Egyszerűbb esetekben a programozónak elegendő lehet kevesebb regiszter is. Ilyenkor a bankváltásos regiszterfájl-kezelésre nincs szükség, és csupán a 0. bank első 128 bájtját használjuk. Ennek a modellnek a neve: **ACCESS RAM**. Ez az adatmemória első 128 bájtja (00H-7FH) az általános felhasználói regisztertömb, illetve a 4 kbájtos memória utolsó 128 bájtja (címe: F80H-FFFH), amely az SFR regisztereket tartalmazza. Így a tartomány ($2 \times 128 = 256$) 8 bittel címezhető.

Azért, hogy egy utasítás végrehajtásakor a címkezelő egység tudja, hogy az adott regiszter melyik memoriamodellek szerint (BSR alapján vagy ACCESS RAM-ként használva) címződik, fel kellett használni az utasítás 16 bites kódjában egy bitet. Ennek a neve „a” (= access bit). Ha az utasításban ennek az értéke a = 0, akkor az ACCESS RAM, ha a = 1, akkor a BSR határozza meg azt a bankot, amiben az adott 8 bites bankon belüli cím regiszterét használjuk.

Az **SFR regiszterek** legtöbbje az adatmemória felső címtartományában <FFFH:F80H> található. Ez az elhelyezés azt is biztosítja, hogy a meglévők „alá”, csökkenő címekben újabb SFR regiszterek definiálhatók, esetleg megjelenő újabb perifériák (pl. CAN busz) és rendszerelemek támogatására. Mivel a meglévők helyzete nem változik, ezért a régi és az újabb fejlesztésű mikrovezérlők között a kompatibilitás SFR regiszterek vonatkozásában fennmarad.

Ennek a megoldásnak van egy nagy hibája: mivel ezeket az SFR regisztereket a rendszerek és az alap-perifériákhoz rendelték, az egyéb perifériák SFR regisztereit már nem kezelhetők ACCESS RAM-ként, csak a BSR segítségével és indirekt címzéssel. Ezért az újabb 18-as típusok fejlesztésénél kissé módosították az eredeti ACCESS RAM koncepciót: bevezettek nyolc új utasítást, és megnövelték az ACCESS SFR-területet, míg a felhasználói ACCESS RAM területet lecsökkentették, de ez a terület az adatmemória bármelyik részére áthelyezhetővé vált, az ACCESS tulajdonság fenntartásával. Ez a gyakorlatban azt jelenti, hogy csupán 96 bájt felhasználói ACCESS RAM-unk van, de ennek kezdőcímét az FSR2 regiszter tartalma határozza meg. Ha történetesen FSR2 értéke 0, akkor az eredeti helyén marad.



2.38. ábra
Módosított ACCESS RAM

2. fejezet: A PIC mikrovezérlők felépítése, fejlődése

Az ACCESS RAM-ként kezelt regiszterek száma viszont 160-ra nőtt, és az adatmemória [F60h-FFFH] tartományát foglalja el.

Alapértelmezésben a kibővített utasításkészlettel együtt az ablakozás (FSR2-vel való indexelt címzés használata) le van tiltva, az egyik konfigurációs regiszterben lévő XINST bit egybeírásával lehet engedélyezni.

Programmemória: címzésére 21 bites PC regiszter szolgál, amelynek részei: PCU<20:16>PCH<15:8>PCL<7:0>. A programmemória-lapozás megszünt. A PCL regiszterrel végzett műveleteknél a PCU, PCH regiszterek tartalmát a PCLATU és PCLATH regiszterekből kapjuk. Bár a belső memória mérete (21 bites cím!) igen nagy lehet, egyes típusoknál mégis kialakítottak egy, a külső memóriát elérő lehetőséget.

A lapozás megszüntetése miatt szakítani kellett a „minden utasítás egyszavas” elvvel. (Például a 21 bites programmemória-cím nem fér bele egy 16 bites utasításba.) A CALL, GOTO utasítások a teljes memóriát elérik (mivel ezek két szóból [= 32 bit] állnak). Ilyen kétszavas utasítás még két 12 bites című fájlregiszter tartalmának mozgatása (MOVFF), illetve az FSRx-et 12 bites címliterállal feltöltő utasítás (LFSR).

Mivel a programmemoriában elhelyezhetünk bájtos felépítésű adattáblákat is, de az utasítások minden 16 bitesek (2 bájt), ezért a memóriakezelésnél erre minden tekintettel kell lennünk.

A programmemória egy folytonos blokknak tekinthető. Három jellegzetes pontja van: a 0000000h címén lévő RESET cím és a két megszakítási cím: 000008h (magas prioritású), 000018h (alacsony prioritású).

Az utasítások a programmemoriában minden 16 bites formában helyezkednek el. Az utasítás alsó 8 bitje (LSB) a páros címen (a legkisebb címbit = 0), míg a felső 8 bitje (MSB) a páratlan címen helyezkedik el.

Ez azt is jelenti, hogy az utasításszámláló (PC = Program Counter) minden kettesével növekszik, vagyis PC legkisebb helyi értéke minden nulla. Ebből az is következik, hogy a címet tartalmazó utasításokban csak 20 bites szócímek szerepelnek, mivel így 1 bitet (a cím legkisebb bitjét) nem kell szerepeltetni.

Verem: 31 darab 21 bites regiszterből áll, az 5 bites veremmutató és a verem alul-, illetve túlcsordulását jelző bitek egy írható/olvasható verem-státusregiszterben találhatók. A PUSH és POP utasításokkal a verem tetején lévő tartalom és az utasításmutató (PC) cseréje kezelhető. Vagyis programból is lehetséges a verem kezelése: a verem tartalmának és mutatójának írása/olvasása.

A verem a szubrutinhíváskor (CALL) vagy a megszakításkor elmentett 21 bites memória-címét tárolja: összesen 31 ilyen címet tud tárolni. Lényegében a verem egy 31 rekeszből álló, 21 bit széles RAM, amelyben az aktuális rekesz címét egy 5 bites veremmutató (STKPTR) tartalma határozza meg. A veremmutató kezdeti értéke RESET után 00000b. Ehhez az értékhez nem tartozik veremrekesz! Az első szubrutinhívás vagy megszakításkérés során STKPTR értéke 1-gyel nő, és a 00001b című verem-memória helyre kerül a visszatérési cím.

Azt mondjuk, hogy a verem tetejére „belenyomtuk” (angolul PUSH) a visszatérési címet. A rutin vagy megszakítás végén álló RETURN típusú utasítás az STKPTR által meghatározott veremcíméről annak tartalmát az utasításszámlálóba tölti vissza. Kiemeli a veremből (angolul: POP). Az ábrán felsoroltak az utasításokat, amelyek a veremmutatót módosítják. Ebben a PC és TOS regiszterek vesznek részt. A POP utasítással csupán a veremmutatót módosítjuk.

A verem tetején lévő rekesz, amire a veremmutató mutat a TOS (= Top Of Stack – a verem teteje). Ide helyeztünk el utoljára tartalmat, közvetlenül írható és olvasható három SFR regiszter segítségével, ezek: TOSU, TOSH, TOSL. Mivel ez hozzáférhető programból is, ezért a verem tetején lévő cím megváltoztatható. Például egy másik visszatérési címet írva a verem tetejére (a TOS regiszterbe), a RETURN hatására ez fog betölteni a PC-be, és erről a címről fog folytatódni a program végrehajtása.

A veremkezeléssel kapcsolatosan két alapvető hiba fordulhat elő:

- overflow** = a verem túlcsordulása. A verem megtelik, azaz a veremmutató eléri az 11111b értéket, és újabb címet akarunk a verembe helyezni (PUSH). Ilyenkor az ezt jelző **STKFUL** bit értéke 1 lesz, az új cím felülírja a 31. helyen lévő értéket, és a **STKPTR** értéke változatlanul 11111b marad. A következő PUSH hatása ugyanez lesz.
- underflow** = a verem alulcsordulása. Akkor következik, ha az üres veremből (**STRKPTR=00000b**) akarunk kiolvasni a PC-be. Ilyenkor a **STKPTR** értéke változatlanul nulla marad, a PC-be a nulla cím (RESET) töltödik, és az eseményt jelző **STKUNF** bit értéke 1 lesz.

Ez a hibajelzés lehetővé teszi, hogy a hibás programműködésből adódó veremcsordulásokat programból tudjuk kezelni, ha szükséges!

Mivel a veremmutató (STKPTR), illetve a verem teteje TOS regiszterek tartalma elérhető és programból módosítható (PUSH és POP műveletek), ezért a veremmanipulációk „trükkös programok” írására adnak lehetőséget.

Megszakítás: a kezelésében is újdonságot találunk: választható módon egy- vagy kétszintű megszakításkezelés használatára van lehetőség. Kétszintű megszakítás esetén a magasabb prioritású megszakításvektor a 08H címen, míg az alacsonyabb prioritású megszakításvektor a 18H címen kezdődik. A megszakításkezelésnél a megszakítás bekövetkezését jelző bit (IR) és a megszakítást engedélyező IE bit mellett a prioritást meghatározó bit (IP) is megjelent.

A megszakítások prioritásos kezelése csupán lehetőség, ha nem alkalmazzuk, akkor bármelyik megszakításkor a 0008h cím töltödik be az utasításszámlálóba.

A prioritásos megszakításrendszer alkalmazásakor minden megszakításhoz a két prioritási szint valamelyikét rendeljük hozzá. Az alacsonyabb prioritású megszakítás bekövetkeztekor a **0018h** cím íródik be az utasításmutatóba, míg a magasabb prioritású megszakítások bekövetkezésekor a **0008h** cím töltödik. Úgy is fogalmazhatunk, hogy az IP bit értéke határozza meg azt, hogy melyik cím töltödik be az utasításszámlálóba. Hogy működik a prioritásos megszakítási rendszer?

Ha egy alacsonyabb szintű megszakítás kiszolgálása közben egy magasabb prioritású megszakítás következik be, akkor az alacsonyabb szintű megszakítás kiszolgálása felfüggesztődik, és a magasabb szinten lévő kerül végrehajtásra. Ennek befejeződése után a felfüggesztett alacsonyabb szintű megszakítás kiszolgálása folytatódik. Azonos szinten lévő megszakítások nem szakítják meg egymást.

A megszakítás kiszolgálásának kezdetén végrehajtódik a **gyors regisztermentés**. A WREG, a STATUS és a BSR regiszterek tartalma elmentésre kerül a WREGS, a STATUS és a BSRS árnyékregiszter-hármasba.

Vigyázat! A prioritásos megszakításrendszerben egy alacsonyabb szintű megszakítást egy magasabb szintű megszakíthat; ilyenkor a mentett értékek felülíródnak! Hasonlóan vigyázni kell, mert a gyors regisztermentés szubrutinhíváshoz is hozzákapcsolható, és egy esetleg bekövetkező megszakítás ezeket a szubrutin elején mentett értékeket felülírja.

A megszakítási rutin végén elhelyezett RETFIE vagy RETURN utasítás „s” paramétere határozza meg, hogy a gyors regisztermentéssel tárolt regiszterértékek visszatöltődjenek-e a helyükre. S = 0 esetén a gyorsvermet nem használjuk. **Fontos:** A gyors regisztermentés megszakításkor automatikus! A visszatöltésről dönthetünk.

Táblavezélezés: A program- és adatmemória közötti adatcsere egyik módja a táblázatos adatkezelés. A programmemória egybájtós memóriahelyét a TBLPTR-nek nevezett 22 bites regiszter tartalma címezi. Innen a megcímzett memóriabájt olvasáskor a TABLAT elnevezésű regiszterbe kerül. Íráskor a TABLAT regiszter tartalma kerül a TBLPTR által megcímzett memóriahelyre. Fizikailag a TBLPTR regiszter három regiszterből épül fel:

TBLPTRU = TBLPTR[21:16] TBLPTRH = TBLPTR[15:8] TBLPTRL = TBLPTR [7:0]

Az adatmozgatás két utasítást igényel. Olvasáskor elsőnek kiolvassuk az értéket a TABLAT regiszterbe, utána az eredményt valahová tároljuk. Például:

```
TBLRD * ; PROGMEM(TBLPTR) -> TABLAT
MOVFF TABLAT, REGI ; TABLAT TARTALMA REGI-BE
```

2. fejezet: A PIC mikrovezérlők felépítése, fejlődése

Hasonlóan az adatmemória-mutatóhoz, vannak műveletek, ahol a táblamutató változtatása is automatikus:

| | |
|------------------|---|
| TBLRD*, TBLWT* | TBLPTR TARTALMA NEM VÁLTOZIK |
| TBLRD*+, TBLWT*+ | TBLPTR TARTALMA A MŰVELET UTÁN EGGYEL NŐ |
| TBLRD*-, TBLWT*- | TBLPTR TARTALMA A MŰVELET UTÁN EGGYEL CSÖKKEN |
| TBLRD+*, TBLWT+* | TBLPTR TARTALMA A MŰVELET ELŐTT EGGYEL NŐ. |

Összefoglalva, a PIC18F családnál az utasításokat négy csoportba soroljuk:

- Bájtooperandusú utasítások:** három operandusuk van:
 - annak a fájlregiszternek a címe, amire az utasítás vonatkozik
 - az eredmény helye (d), d = 0 wreg
 - a fájlregiszter címének kezelési módja (a)

- Bitoperandusú utasítások:** operandusai:
 - annak a fájlregiszternek a címe, amire az utasítás vonatkozik (f)
 - a fájlregiszter adott bitjének a címe (b)
 - a fájlregiszter címkezelésének módja (a)

- Literálutasítások:**
 - egy konstans érték, amit egy regiszterbe kell tölteni (k)
 - annak a fájlregiszternek a címe, amire az utasítás vonatkozik (f)
 - nincs operandus (–)

- Vezérlő utasítások:**

- programmemória-cím (n)
- a CALL és RETURN utasítások működési módja (s)
- a táblaírás és -olvasás módját határozza meg (m)
- nincs operandus

Kibővített utasításkészlet: A PIC18-as mikrovezérlők 75 utasítása mellett a CPU képes nyolc kibővített utasítás végrehajtására. Ezek további indirekt és indexelt címzést tesznek lehetővé, közöttük szerepel egy speciális indexelt literál ofszet címzési mód, amit számos szabványos PIC18-as utasításnál használhatunk.

Alapértelmezésben a kibővített utasításkészlet használata le van tiltva, az egyik konfigurációs regiszterben lévő XINST bit egybeírásával lehet engedélyezni. A kibővített utasítások egy része literál típusú, amelyek vagy az FSR regisztert használják, vagy indexelt címzést valósítunk meg velük.

Az utasítások közül kettő (az ADDFSR és SUBFSR) az FSR2-es regisztert használja.

ADDFSR x,k FSRx+k → FSRx a k literál hozzáadása FSRx regiszterhez (x = 0, 1, 2)

ADDULNK k FSR2 + k → FSR2, a k literál FSR2-höz adása és RETURN

CALLW [TOS] → PC

[PC+2] → TOS, szubrutinhívás W felhasználásával

[W] → PCL,

[PCLATH] → PCH,

[PCLATU] → PCU

MOVSF Zs,fd [[FSR2]+Zs] → fd

FSR tartalmához hozzáadjuk Zs-t, és az erről a cím-ről kiolvasható érték lesz fd tartalma a forrást és a célt is indexelve kezeljük.

MOVSS Zs,Zd [[FSR2]+Zs] → [[FSR2]+Zd] k literál tárolása FSR2-ben lévő című helyre, FSR2 csökkentése

PUSHL k k → [FSR2], FSR2-1 → FSR2

SUBFSR x,k FSRx-k → FSRx

FSR2-k → FSR2,

[TOS] → PC

FSRx tartalmából egy k konstans kivonása

k literál FSR2-ből való kivonása és RETURN

2.9. 16/24-ES MIKROVEZÉRLŐK, PIC24/PIC30/DSPIC33 CSALÁDOK

16/24 A 8 bites mikrovezérlők piacát meghatározó Microchip cég megjelent a 16 bites processzorcsalád-jával, a dsPIC-kel. Nyilvánvaló, hogy a már meglévő 16 bites piacra betörni csak valami eredeti ötlettel lehetett: a meglévő PIC-el-vekre alapozva nem csupán megduplázták az adatutak számát, hanem olyan felépítést alakítottak ki, amely speciális utasítások segítségével digitális jelfeldolgozásra is képes.

Természetesen nem minden feladat megoldása igényel digitális jelfeldolgozást, ezért a termékcsalád gyorsan kettévált: a jelfeldolgozó egységet nem tartalmazó PIC24-es vonalra valamint a jelfeldolgozó egységgel rendelkező dsPIC30F és a nagyobb teljesítményű dsPIC33F családra. A két termékcsalád csupán a jelfeldolgozó egységben és az azokat kiszolgáló áramköri megoldásokban különbözök, ezért elsőként a PIC24 típus ismertetjük. Mivel a 16 bites termékcsaládról még nincs irodalom, ezért kissé részletesebben fogjuk bemutatni. Bevezetésként célszerű egy táblázatban összehasonlítani a legfejlettebb 8 bites PIC18-as családdal. Fontos megemlíteni, hogy a tervezésükönél folytatták a 8 bites konstrukcióknál megszokott jól bevált megoldásokat.

| Tulajdonság | PIC18 | PIC24, dsPIC30, dsPIC33 |
|--|------------------------|--|
| Adathossz | 8 bit | 16 bit |
| Utasításhossz | 16 bit | 24 bit |
| Adatmemória (DS) mérete | 12 bites cím – 4 kbájt | 16 bites cím 64 kbájt = 32 kiszó |
| Akkumulátor | WREG | W0-W15 regisztertömb |
| Programmemória (PS) mérete | 21 bites cím=1M*16 bit | 23BIT->4M*24 bit |
| Adatcím operandushosszak | bit-bájt-12 bit | bit -4 bit-13 bit-16 bit |
| Táblakezelés | 23 bit | 24 bites TBLPTRH:TBLPTRL |
| Adatmemoriát program-memoriába címezni | nincs | utasításhossz PSV + W összesen 23 bit |
| SFR tartomány | FFF-től lefelé | 0X000-0X07FE – 2 kiszó |
| Megszakítás | 2 szint prioritásos | Igazi vektoros prioritásos |
| PERIFÉRIÁK | | |
| portok | 8 bit | 16 bit + HW-es változásfigyelés |
| Timerek | 8 és 16 bit | 16 és 32 bites |
| COMPARE | 2 | Max 8 |
| CAPTURE | 2 | Max 8 és 4-es puffer |
| AD | 10 bit | 10 és 12 bit 16-os puffer |

2.40. ábra

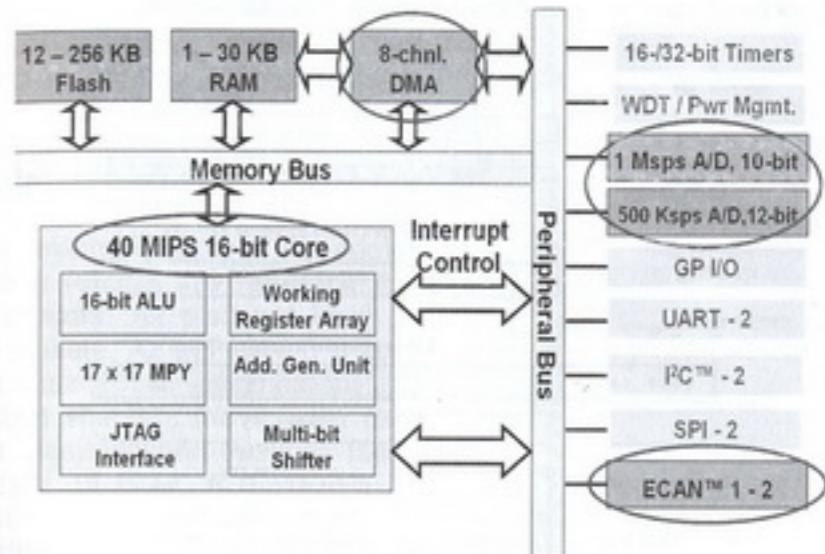
8 és 16 bit adathosszúságú mikrovezérlők összehasonlítása

2. fejezet: A PIC mikrovezérlők felépítése, fejlődése

2.9.1. A PIC24-es termékcsalád

A PIC24 jelölés utáni F és H betük csupán teljesítménybeli és perifériabeli különbségekre utalnak.

- A PIC24H a nagyobb teljesítményű: 40 MIPS-es, tápfeszültsége 3,0–3,6 V között lehet, 12 és 10 bites AD-t, több időzítő, Capture, Compare, perifériát tartalmaz, valamint ECAN és DMA megoldást.
- A PIC24F típus csak 16 MIPS-es, 2,0–3,6 V közötti működtető feszültséggel és 10 bites AD-vel.

2.41. ábra
PIC24F/24H blokkvázlat

A részleteket a Microchip Product Selection Guide anyagában érdemes áttekinteni.

Az utasításkészlet 76 utasítást tartalmaz. A legtöbb utasítás egy ciklus (= két órajel) alatt hajtódik végre, és egy 24 bites programmemória-helyet foglal. Az adatmemória 16 bittel címezhető, vagyis maximum 64 kbájtos lehet. Mivel a 16 bites mikrovezérlők 8 bites adatot is kezelnek, ezt úgy oldották meg, hogy egy utasítás vonatkozhat egy 16 bites adat alsó bájtjára. Ezt az utasítás mnemonikja után írt .B kiterjesztéssel jelzik.

Az eddigi egyetlen WREG akkumulátor helyett 16 elemű W0..W15 16 bites regisztertömböt használunk. A kompatibilitás miatt W0 szerepe megegyezhet a WREG szerepével. Ezek bármelyike használható mint az utasítások adat-, cím- vagy ofszet- (címeltolás) regisztere. Az utolsó (W15) regiszter szoftververem-mutatóként működik, aminek megszakítások és szubrutinhívások (CALL) során van jelentősége.

Programmemória: Az utasítások hossza 24 bit. A programszámláló (PC) 23 bites, így 4M*24 bit memória címzésére van lehetőség. A programmemória elején található a megszakítási vektortábla. A programmemóriában lévő tartalom adatként kezelésére már három megoldás lehetséges:

1. RETLW utasítás

RETLW #LIT10,W5

; RETURN UTASÍTÁS VÉGREHAJTÁSA, W5 TARTALMA

RETLW.B #LIT8,W4

; A 10 BITES KONSTANS LESZ (0..1023)

RETLW.W #LIT8,W4

; RETURN UTASÍTÁS VÉGREHAJTÁSA, W4 ALSÓ FELE

RETLW.D #LIT8,W4

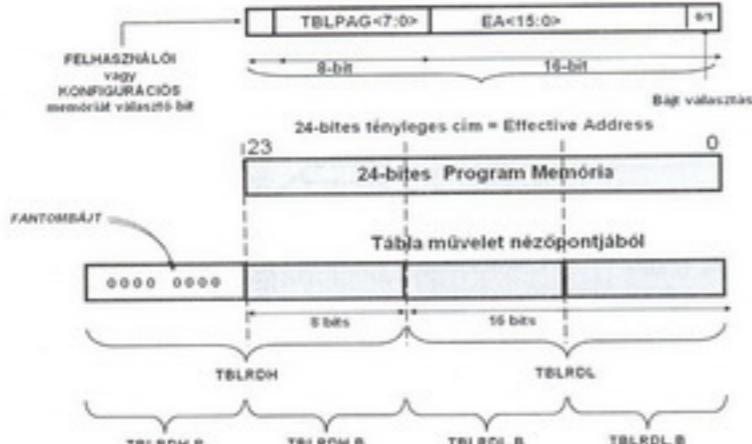
; A 8 BITES ÁLLANDÓT HORDOZZA VISSZATÉRÉSKOR

2. Táblázatkezelés

A táblamutató maximum 24 bites lehet. Ha a legfelső bit 1, akkor a konfigurációs memóriára mutat. Két részből tevődik össze: a TBLPAG regiszter tartalma, kiegészítve valamelyik W0...W15 regiszter tartalmával. Az így képződött 24 bites programmemória-címen lévő 24 bites bitcsoport 24 bitje jelenti az adatot. Ezért két művelet kell az alsó 16 és a felső 8 bites kezelésére.

Ha 8 bites adatot akarunk elérni, akkor a kijelölt W0...W15 regiszter legalsó bitje dönti el, hogy a kijelölés az alsó vagy a felső bájtra vonatkozik (1: felső bájt, 0: alsó bájt).

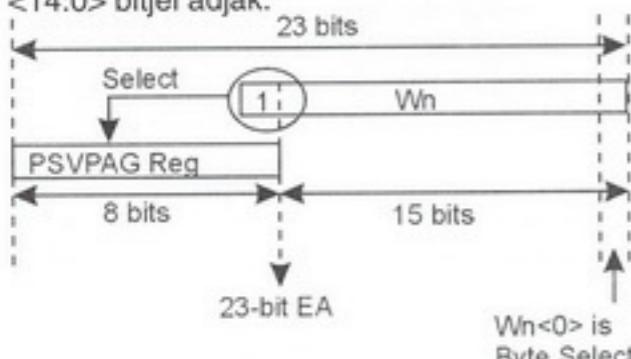
```
TBLRDL.B [W0++],W2 ; KIOLVASSUK PROGMEM[TBLPAG:[W0]]
; TARTALMÁT ÉS W2-BE HELYEZZÜK.
; A MŰVELET UTÁN W0 ÉRTÉKÉT 1-GYEL NÖVELJÜK
```



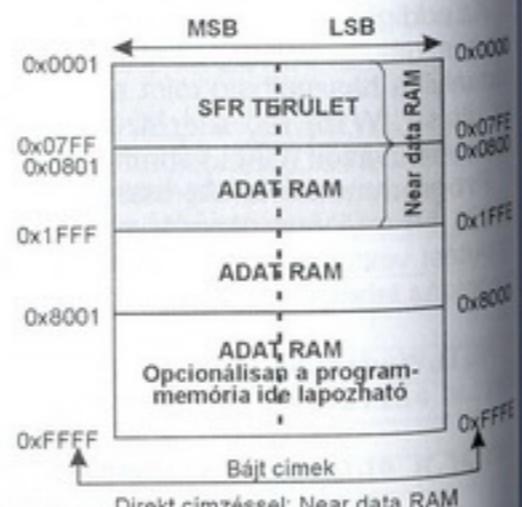
2.42. ábra
PIC24 táblakezelés

3. Programmemória (PS) adatmemoriába (DS) tükrözése

A programmemória 32 kbájtos egybefüggő része az adatmemória felső területére illeszthető, és adatmemória-címzéssel elérhető. Ehhez két feltételnek kell teljesülnie: a PSV bitet 1-be kell állítani (engedélyezés), CORCON<2> = 1, és az adatmemória kezdetét jelöli cím 15. bitje: 1, akkor a PS-ben lévő adat DS-ból elérhető. PSVPAG regiszter <7:0> bitje adja a 23 bites programmemória-cím felső 8 bitjét, míg a cím alsó felét a címzö W0...W15 regiszter <14:0> bitjei adják.



2.43. ábra
PSV címgenerálás



2.44. ábra
PIC24 adatmemória

2. fejezet: A PIC mikrovezérlők felépítése, fejlődése

Példa: legyen az adatmemória kezdete, ahova a programmemóriát lapozzuk: 9000h. (1001_0000_0000_0000b) PSVPAG regiszter tartalma legyen 20h (10_0000). Ekkor a programmemória 10_0000_001_0000_0000_0000 = 1_0000_0001_0000_0000_0000=10100h címűtől kezdődően fog látszani.

A programmemória 0-0x100h címén helyezkedik el a megszakítástáblázat, felette pedig a program utasításait tartalmazó memóriaterület, ez 23 <22:0> bittel címezhető. Mikor a 23. bit értéke 1, akkor a konfigurációs memóráit tudjuk címezni.

Adatmemória: Maximum 65536*8 bit-es méretű lehet, de 32768*16 bites szervezésű memóriának is tekinthető, hiszen az adatok lehetnek bájt vagy 16 bites szó méretük.

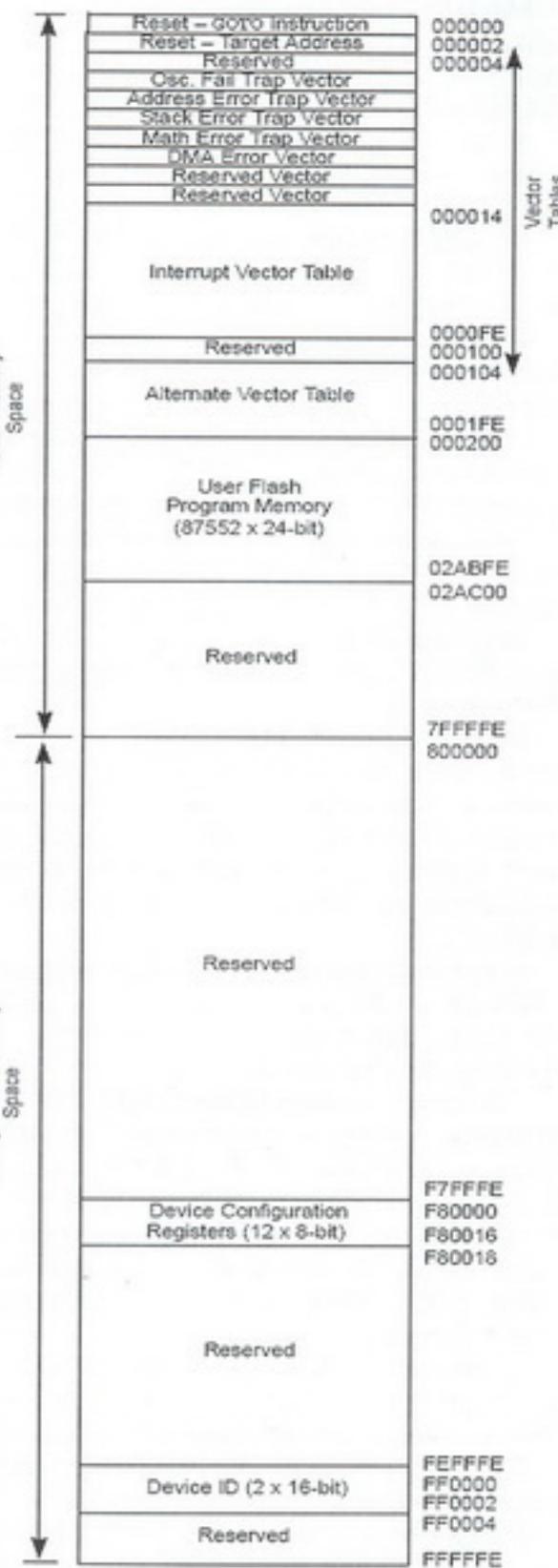
Közvetlen címzés: Az első két kbájt az SFR memória területe. Az ezt követő 6 kbájt általános célra használható adatmemória. Az első 2+6 = 8 kbájt speciális (neve: near data RAM), mert az utasításokban használt fájregiszter címhossza 13 bites. A 0...1FFFh területen lévő terület közvetlenül címezhető.

Közvetett címzésre az itt nem létező SFR regiszter helyett a W regisztereket használjuk. Mivel ezek 16 bitesek, a segítségükkel a teljes RAM-terület elérhető.

A felső 32 kbájt azért fontos, mert ide lehet címezni a programmemória 32 kbájtos szegmensét, és az adatként kezelhető. A memória tényleges nagysága családon belül változó, minden az adatlapokat kell tanulmányozni.

Veremkezelés: szakítottak az eddig megoldással, vagyis nincs különálló veremmemória, aminek szintjei (2-8-31) korlátokat jelentettek az egymásba ágyazott szubroutines és megszakítások kezelésénél. Helyette az adatmemoriában lehet vermet kialakítani az „LNK méret” utasítással, használat után pedig fel lehet szabadítani az ULNK utasítással. A verem mérete a W14, mutatója a W15-ös regiszterben van. Egy külön regiszter, az SPLIM tartalma határozza meg a verem maximális méretét. Túlcordulási hiba lép fel, ha [W15] > SPLIM, és alulcsordulás, ha [W15]<0x800h.

Megszakítás: A PIC18-nál bevezetett eddig legfejlettebb két megszakítási vektorral rendelkező prioritási rendszert teljes mértékben átalakították.



2.45. ábra
PIC24 programmemória

Megjelent a megszakítási vektortábla (*Interrupt Vector Table – IVT*), ahol minden megszakításforrásnak saját vektorcíme van, ami az adott megszakítás kiszolgálása előtt az utasításszámlálóba íródik. Ellenőrzési célokra egy másik IVT-re lehet átváltani. A megszakításba való belépéskor a késleltetés 5 ciklus, míg kilépéskor 3 ciklus.

Különválasztották a rendszer működése közbeni események kezelését a külső, perifériák okozta eseménykezeléstől. Az első típus neve: **trap**, míg a második típus neve: **megszakítás**. Az IVT-ben 8 nem letiltható (nem maszkolható) trap vektor és 54 megszakításvektor van.

Prioritáskezelés: a CPU-nak 16 prioritási szintje van, amiből a 8–15 szint a trapekhez, míg a 0–7 szint a megszakításforrásokhoz van rendelve. IVT-nek saját természetes prioritása van, a növekvő memóriacímek irányában csökken.

Minden felhasználói megszakításforrás prioritásának 7 szintje lehet, ami felülbírája a természetes prioritást. Azok a források szakíthatják meg a CPU futását, amelyeknek a szintje nagyobb, mint a CPU aktuális szintjének (IPL<2:0> bitjei).

A megszakítások egymást megszakíthatják, ennek tiltása az NSTDIS (INTCON1<15>) bit 1-be írásával tiltható. Ha NSTDIS = 1, akkor a CPU prioritási szintje 7-re áll be. Megszakításkor az SRL és PC regiszterek automatikusan a verembe kerülnek.

Regiszterek mentésére/visszaállítására a PUSH(.D) és POP(.D) utasításokat kell használni. A PUSH.S és POP.S utasításokkal a W0...W3 és a STATUS regiszter bitjeit tudjuk menteni (DC,N,OV,Z,C) az árnyékregiszterekbe.

Programozói modell: programozói oldalról a PIC24 a 2.46. ábrán látható regiszterekkel kezelhető.

Munkaregisztertömb (W0...W15): A 16*16 bites regisztertömb elemei adatokat, címeket vagy címtolásokat tartalmaznak. A bájtoperandust használó utasítások csak a 16 bites regiszter alsó 8 bitjét kezelik. A tömb regiszterei a 0-0x1F memóriacímeken is elérhetők.

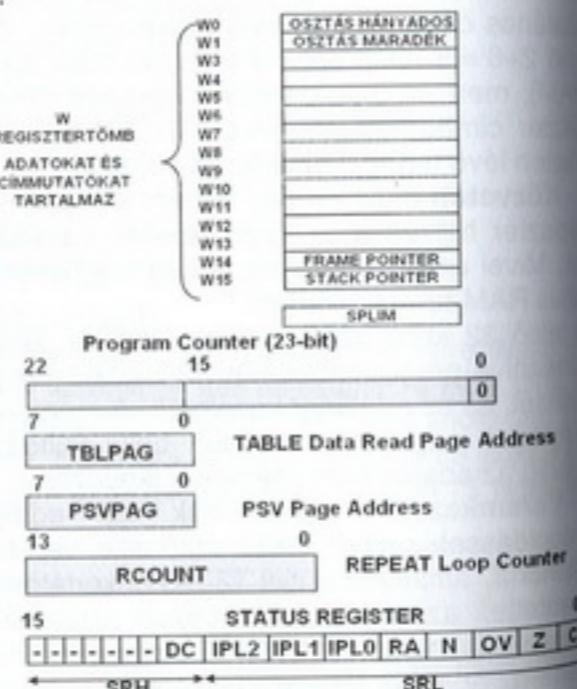
Alapértelmezett munkaregiszter (WREG): A fájlregisztereket használó kétoperandusos utasítások munkaregiszterként a W0 regisztert használják.

Szoftver verem-keretmutató (Frame Pointer): A keret a felhasználó által definiált része a veremnek, amely a lokális változók tárolására szolgálhat. W14 regisztert használjuk erre az LNK és ULNK utasításokkal kapcsolatosan. Ha nem használjuk ilyen célra, akkor általános célú munkaregiszterként használható.

Szoftver veremmutató (Stack Pointer): W15-ös regiszter a szoftver veremmutató, és automatikusan változik az értéke szubrutinhíváskor, megszakításkor. A tömbben való elhelyezése miatt a tartalma könnyen változtatható.

Szoftver veremmutató határregiszter (Splim): A veremmutató értékét korlátozza, amikor túlcordulását megakadályozza.

Programszámláló (Program Counter): A PC 23 bites, 4M*24 bites memóriát képes kihasználni. Az adatmemória címzéssel való kompatibilitás miatt a programmemoriát 16 bites szavakból állónak tekintjük. Mivel ez 24 bites dupla szónak felel meg, a memóriát PC<22:17> címekkel címezük, és a PC minden kettesével növekszik (PC<0> minden nulla). 0x80000000 címen lévő konfigurációs adatokat, egység- és eszközazonosítókat tartalmazza a programmemória a PC-vel nem érhető el, csak a táblakezelő utasításokkal.



2.46. ábra
PIC24 programozói modell

2. fejezet: A PIC mikrovezérlők felépítése, fejlődése

TBLPAG regiszter: A programmemória-cím felső 8 bitjét tartalmazza a táblakezelő utasításoknál.

PSVPAG regiszter: Felső 32 kbájtos adatmemoriára rá tudjuk illeszteni a programmemória 32 kbájtos részét. A PSVPAG regiszterbe írt érték határozza meg, hogy a programmemória melyik 32 kbájtos részét akarjuk adatmemoriaként látni.

RCOUNT regiszter: Ez a 14 bites regiszter a REPEAT utasítás ciklusszámlálója. A REPEAT utasítást követő egyetlen utasítás RCOUNT+1-szer hajtódik végre.

STATUS regiszter: Alsó nyolc bitjében (SRL) vannak a szokásos bitek: C, Z, OV, N, valamint az újak: RA – Repeat utasításvéghajtás aktív bit, <IPL2:IPL0> - CPU prioritásbitek, SRH<0> - DC - half carry.

Utasításkészlet: A következő címzési módok lehetségesek: azonnali, relatív, állandóval történő (literál), közvetlen memória, közvetlen regiszter, indirekt regiszter és regiszter ofszet módok. minden utasítás egy címzési csoporthoz kapcsolódik alapértelmezésben, de minden a hét címzési mód használható az utasításokban.

A legtöbb utasítás végrehajtásánál utasításciklusonként képes az eszköz adatmemória olvasására, munkaregiszter olvasására, adatmemória írására és utasításkiolvasásra a programmemoriából. Ezért lehetővé válik a háromoperandusos utasítások (A + B = C) használata egyetlen ciklus alatt.

Az utasításkészlet áttekintése

A 76 utasításból álló utasításkészlet kilenc, többé-kevésbé jól elkülöníthető csoportra osztható. A legtöbb utasítás 1 ciklus alatt végrehajtódik, de vannak többciklusos utasítások is. Ezek a csoportok:

- mozgató utasítások;
- aritmetikai-matematikai utasítások;
- logikai utasítások;
- forgató/eltoló utasítások;
- bitkezelő utasítások;
- hasonlító / átlépő (compare/shift) utasítások;
- ugró utasítások;
- árnyékregiszter/veremkezelő utasítások;
- vezérlő utasítások.

Az utasítások többsége, a következőkben leírt címzési móddal használhatók. Emlékezzetől, az operandusoknak csupán három típusuk lehetséges, nevezetesen:

- egy állandó (literál), amivel az utasítást végre kell hajtani,
- regiszter (avagy ennek a címe), illetve
- a programmemória címe (ugró utasítások megvalósításához).

Programcímzési módok: ezek a programmemória-címeket tartalmazó utasítások.

- Az egymás utáni utasítások végrehajtásánál az utasításszámláló kettesével növekszik. $PC = PC + 2$.
- GOTO CÍM és CALL CÍM esetén $PC = LIT23$ – a 23 bites programmemória-cím
- GOTO Wn és CALL Wn esetén $PC = [Wn]$, $n = 0 \dots 15$
- BRA CÍM és RCALL CÍM relatív ugrás $PC = PC + 2 * SLIT16$. SLIT16 egy 15bit + előjelű állandó (Signed LITERAL), a relatív távolság.
- BRA feltétel,CÍM esetén: $PC = PC + 2$, ha a feltétel nem teljesül, különben: $PC = PC + 2 * SLIT16$.
- RCALL Wn esetén: $PC = PC + 2 * Wn$

Adatmemória-címzések: az utasítások az adatmemória alábbi módon megcímzett regisztereit használják a műveletek végrehajtásakor.

IMMEDIATE (azonnali) címzés: A művelet operandusa egy állandó.

ADD.B #0X10,W0 ; [W0 ALÓ BÁJTJA]=[W0 ALÓ BÁJTJA]+0X10

XOR W0,#1,[W1] ; W0 ÉS 1 KIZÁRÓ VAGY KAPCSOLATA W1-BE KERÜL

FÁJLREGISZTER-címzés: az adatmemória operanduscíme: 0-0xFFFF (alsó 8 kiszó – ne memory) tartományban van.

DEC 0X1000 ; 1000-ES CÍMEN LÉVŐ ADATREGISZTER TARTALMÁNAK
; CSÖKKENTÉSE 1-GYEL

MOV 0X22FF,WREG ; W0-BA 22FF CÍMŰ REGISZTER TARTALMA KERÜL

KÖZVETLEN REGISZTERcímzés: az operandusok és az eredmény is a W0...W15 regiszterek valamelyike, ezért a címtartomány 0-0x1F.

EXCH W2,W4 ; W2 ÉS W4 REGISZTEREK TARTALMÁNAK FELCSERÉLÉSE

IOR #0X33,W0 ; W0-BA W0 ÉS 33H VAGY KAPCSOLATA KERÜL

ADD W2,W4,W6 ; W6 REGISZTER TARTALMA W2 ÉS W4 REGISZTER
; ÖSSZEGE LESZ

KÖZVETETT REGISZTERcímzés: az operandusok A W0...W15 regiszterek tartalma által megcímezett érték. Címtartomány: 0-0xFFFF. A művelet előtt vagy után van lehetőség ezen regiszterek tartalmának automatikus megváltoztatására (INC: jelölése: ++, DEC jelölése: --). Ezek a változtatások: nincs változtatás, a művelet utáni változtatás (POST), a művelet előtti változtatás (PRE), illetve változtatás egy másik regiszter hozzáadásával.

ADD W1,[-W5],[W8++] ; W5 TARTALMÁT A MŰVELET ELŐTT 1-GYEL
; CSÖKKENTJÜK, MAJD ERRÓL A CÍMRÖL
; KIOLVASOTT ÉRTÉKHEZ HOZZÁADJUK W1
; TARTALMÁT, ÉS AZ EREDMÉNYT A W8
; TARTALMA ÁLTAL KIJELÖLT CÍMŰ REGISZTERBE
; ÍRJUK, MAJD W8 TARTALMÁT 1-GYEL
; MEGNÖVELJÜK.

MOV.B [W0++],[W9--] ; W0 TARTALMA ÁLTAL MEGCÍMZETT REGISZTER
; ALSÓ BÁJTJÁT W9 TARTALMA ÁLTAL
; MEGCÍMZETT ADATMEMÓRIA-HELYRE ÍRJUK,
; MAJD W0 TARTALMÁT 1-GYEL NÖVELJÜK W9
; TARTALMÁT 1-GYEL CSÖKKENTJÜK

MOV [W0+0X20],W1 ; W0 TARTALMÁHOZ 20H-ADUNK, AZ ÍGY
; KÉPZÖDÖTT CÍMEN LÉVŐ TARTALMAT W1-BE
; ÍRJUK

Ez a rövid összefoglaló – ami a PIC18 utasításkészletének ismerete alapján még jobban megérthető – természetesen nem helyettesíti azt, ami az utasításkészlet részletes leírásában megtalálható.

2.9.2. dsPIC termékvonal: DSP PIC

Ez a termékvonal is szétvált:

- a **dsPIC30F** típus 2,5–5,5 V között működik, teljesítménye 30 MIPS, 144 kB flash programmemoriája, 8 kbájt adatmemoriája lehet, a lábszám 18 és 80 között változik. Adat EEPROM-ot is tartalmaz.

- a **dsPIC33F** típus 3,0–3,6 V között működik, teljesítménye 40 MIPS, 256 kB flash programmemoriája, 32 kbájt adatmemoriája lehet, a lábszám 64 és 100 között változik. Az adat EEPROM a programmemoriában emulált, egy hatcsatornás DMA is a perifériák közé került.

2. fejezet: A PIC mikrovezérlök felépítése, fejlödése

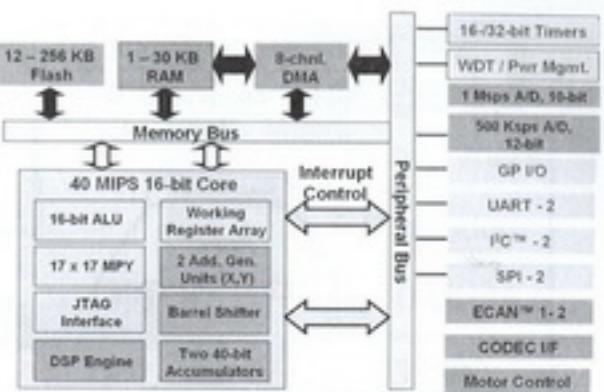
Az utasításkészlet két csoportra bontható: a megszokott és a PIC24-es családnál bemutatott mikrovezérlő (MCU – MicroController Unit) utasításokra, illetve a jelfeldolgozó (DSP – Digital Signal Processing) utasításosztályra. Természetesen ezek az utasítások váltakozva is jól használhatók, és C fordítóhoz vannak optimalizálva.

A 32 kiszóként vagy 64 kbájként címzhető adatmemória két részre bontható: X és Y adatmemoriára (a DSP algoritmusok nagyon gyakran két adatsorral dolgoznak). Mindkét résznek saját címgeneráló egysége (AGU – Address Generation Unit) van.

Az MCU utasítások az X memóriát használják, míg a DSP utasításokkal minden két memóriarész használható. A memória felosztás aránya beállítható.

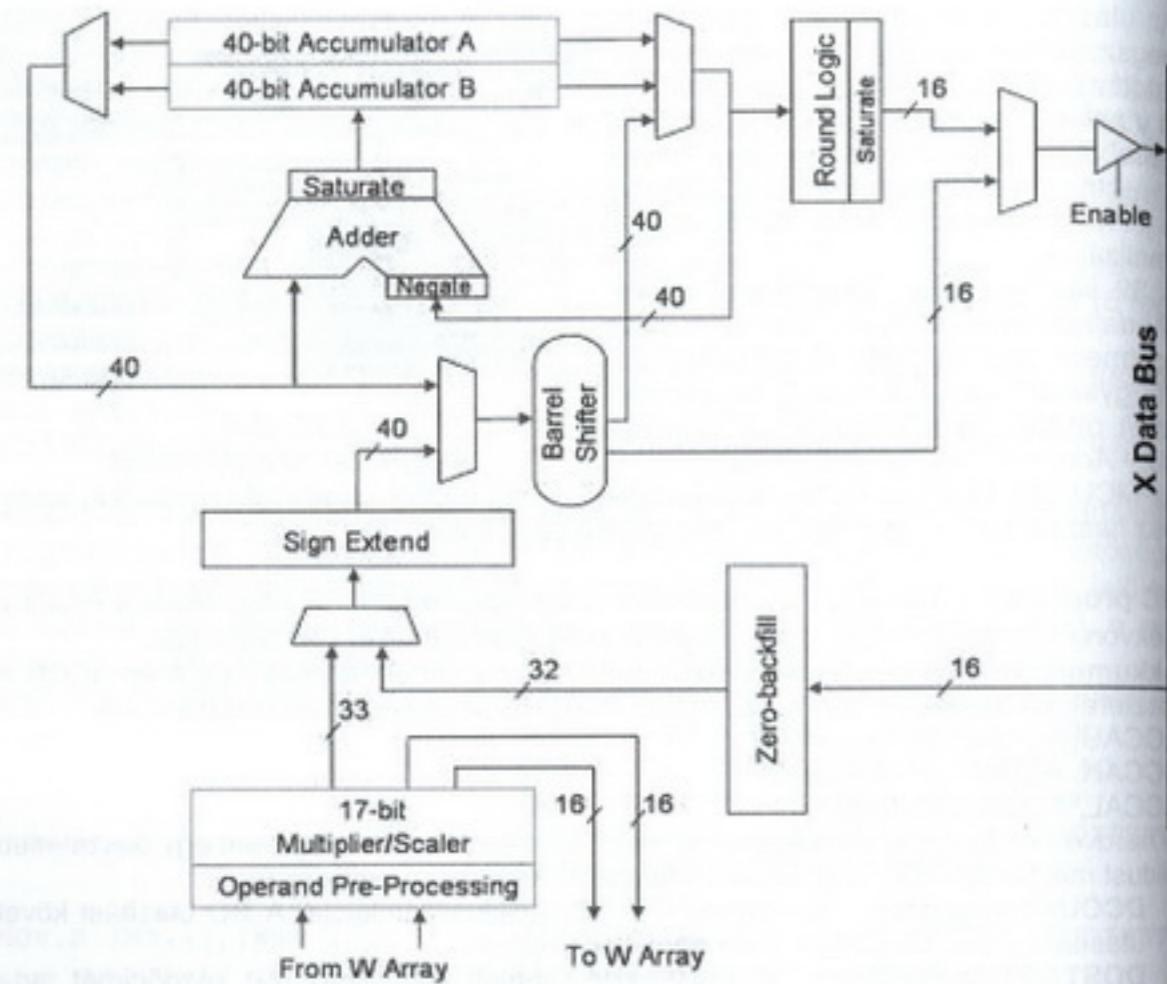
dsPIC programozói modell: Természetesen a szereplő regiszterek nagy része a PIC24-es termékvonalnál bemutatottal egyező, ezért itt csak az eltéréseket szerepeljük.

- Akkumulátor A és akkumulátor B:** A DSP műveleteknél használt ACCA és ACCB regiszterek 40 bitesek, és minden hárrom memóriacímen is elérhető részre oszlik: ACCAU, ACCBU (39–32. bites), ACCAH, ACCBH (31–16. bites), ACCAL, ACCBL (15–0. bites)
 - A hardverben megvalósított egyszerű REPEAT ciklus mellett megjelent egy összetettebb ciklust megvalósító DO utasítás is, a megfelelő kiszolgáló regiszerekkel:
 - DCOUNT regiszter:** A hardveres DO ciklus ciklusszámlálója. A DO utasítást követő utasítássorozat DCOUNT+1-szer hajtódik végre.
 - DOSTART regiszter:** A DO utasítással ismételt utasítássorozat kezdőcímét tartalmazza.
 - DOEND regiszter:** A DO utasítással ismételt utasítássorozat végcímét tartalmazza. Idáig tart a ciklus.
 - STATUS regiszter:** Újabb bitesekkel egészült ki: a 16 bites állapotregiszter az éppen végrehajtott utasításokról tartalmaz információkat. Több ilyen bitcsoportot tartalmaz: MCU ALU STATUS bites, a „szokásos” státusbites: C, Z, OV, N, DC. LOOP státusbites: DA és RA státusbites akkor aktívak (=1), amikor éppen egy REPEAT vagy DO ciklust hajtunk végre.
 - DSP státusbites:** Megszakításprioritási szint státusbites.
 - CORCON regiszter:** Ez a 16 bites CORRe CONtrol regiszter a CPU mag működését vezérli. Illusztrációként néhány változtatható tulajdonság:
 - programmemória adatmemoriára illesztése,
 - ACCA, ACCB akkumulátorok telítettségének (szaturációjának) kezelése,
 - kerekítés (rounding) kezelése,
 - DSP utasítások szorzási módjának kezelése,
 - DO ciklus idő előtti befejezése.
- RESET után a CORCON regiszter beállításai egy szokásos módot definiálnak. Még két tulajdonság van, amit CORCON tartalmaz: az egymásba ágyazott DO ciklusok száma: DL<2:0>, valamint IPL<3> bit, ami a TRAP kiszolgálásakor válik aktívvá.
- SHADOW (árnyék) regiszterek:** DO ciklus paramétereit kerülnek elmentésre.



2.47. ábra

A dsPIC30/33 blokkvázlatára



2.48. ábra
DSP engine (egység)

DSP UTASÍTÁSOK

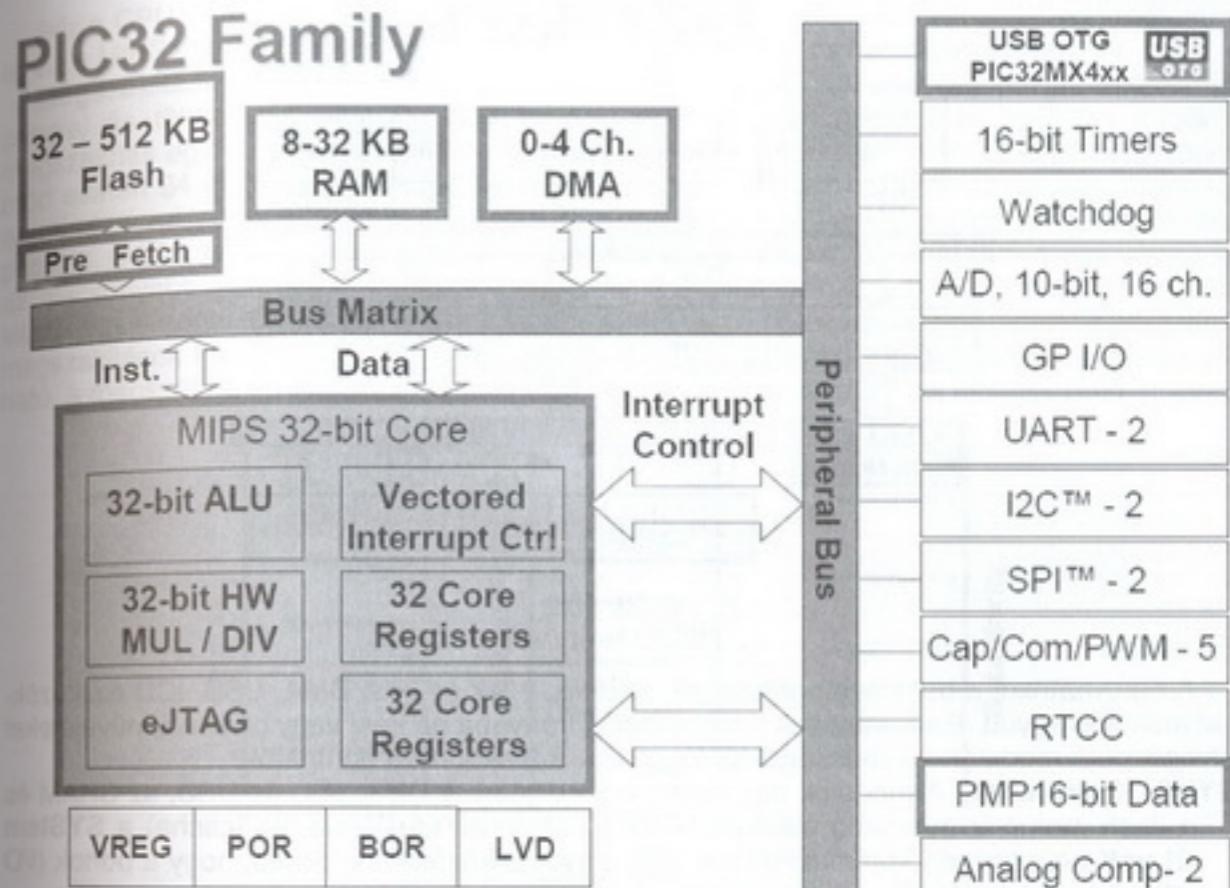
Mindkét, X és Y címtérben túlcsorduló (modulo) címzést használhatunk. Ez lehetővé teszi, hogy a DSP algoritmusokban szereplő címhatárfigyelést elhagyjuk. Az X AGU-val ez a címzés az MCU utasításoknál is használható. Az X AGU ezenkívül támogatja a fordított bitcímzést, amellyel nagyon leegyszerűsödik az FFT algoritmus megvalósítása. Ott ugyanis nem egymás mellett adatokkal kell műveleteket végezni.

A DSP hardveregységben van egy gyors, 17×17 bites szorzó, 40 bites ALU, két 40 bites akkumulátor, valamint egy 40 bites, egy ciklus alatt jobbra vagy balra 16 bittel eltoló ún. barrel shifter. A MAC utasítás és a hozzájuk kapcsolódó utasítások képesek két adatot lehívni a memoriából, míg két munkaregiszter tartalmát összeszorozzuk. Ezek az utasítások az adattér kettéválasztását igénylik.

2.10. A 32/32 BITES MIKROVEZÉRLŐK – PIC32-ES CSALÁD

32/32 A grafikus alkalmazásokhoz és a gyors kommunikációhoz szükséges számítási teljesítmény miatt a Microchip egy nagyobb teljesítményű processzormag használata mellett döntött, körbeépítve a jól bevált 16 bites PIC24-es család perifériakészletével.

A Microchip PIC32 mikrovezérlő az MIPS Technologies cég M4K típusú kis fogyasztású RISC processzormagján alapul, utasításkészletként a kibővített MIPS32 Release 2 verziót használja.



2.49. ábra
PIC32 blokkvázlat

A PIC32 processzorcsaládot úgy terveztek, hogy C-ben programozva optimális kódgenerálást tegyen lehetővé. Több adattípust támogat, egyszerű, de rugalmas címzési módokat használ, ami az optimális fordításhoz szükséges.

A 32 bites utasításkészlet 32 darab általános célú 32 bites regisztert használ valamint további két regisztert a szorzás és osztás műveletekhez.

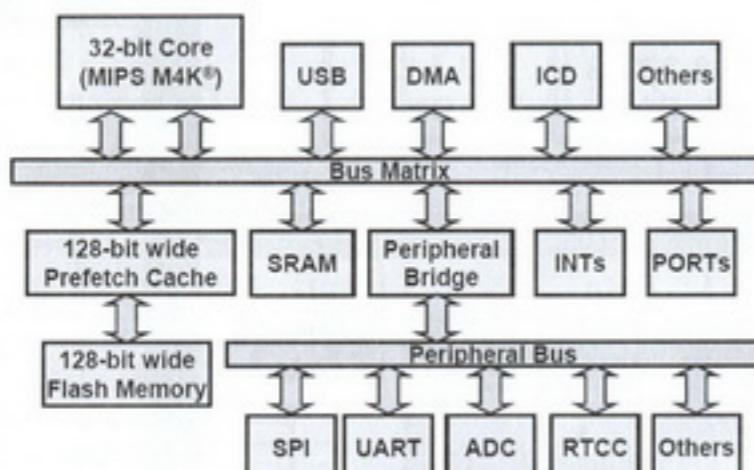
Az M4K processzor is Harvard felépítésű, különálló adat- és utasításbuszzal rendelkezik, ami a buszmátrix (**Bus Matrix**) egységhez kapcsolódik. A Bus Matrix, ami a CPU-val azonos sebességgel működik (ez a SYSCLK frekvencia), lényegében egy nagy sebességű kapcsolórendszer, ami pont–pont kapcsolatot létesít a modulok között.

A PIC32 egy 128 bit széles flash memóriát használ, vagyis 4*32 bit egyszerre kiolvasható utasítást tárol. Ezt segíti egy 128 bites Prefetch Cache (előre lehívó tároló) modul. A modul képes a következő 128 bitet (4 utasítást) a memoriából lehívni, és egy gyors, a CPU magban elhelyezett cache memóriába betölteni. Ez lehetővé teszi, hogy a gyorsabb CPU ne várakozzon a lassúbb flash-ból való olvasásra.

Az M4K mag ötállapotú utasításfeldolgozóval (*pipeline*) dolgozik. minden utasítás végigmegy az öt állapotban, vagyis egy időben öt utasítás végrehajtása folyik, ez órajelenként egy utasítás végrehajtását jelenti. Ez 1,5 MIPS/MHz teljesítményt eredményez.

A memórialeképzés úgy van kialakítva, hogy a program- és adatmemória közös lineáris címzéssel érhető el. A programozó ezért egy címmutatót használhat, ami persze különálló címtartományokra mutat, és adat RAM-ból is történhet utasítás-végrehajtás.

PIC32 Block Diagram



2.50. ábra
PIC32 felépítése

A buszmátrixot a buszmesteregyesek vezélik, ezek a CPU, DMA, USB, ICD eszközökkel működik együtt. Ezek képesek a többi modul irányába az írási vagy olvasási műveleteket elvégezni. A modulok – sebességüktől függően – két csoportra bonthatók:

SYSCLK perifériák: A modulok egy része: a CPU mag, a DMA, ICD és USB, az SRAM és a flash memória gyorsabb elérését biztosító átmeneti utasítástároló (cache) a **SYStem CLocK** sebességen kommunikálnak, akár egymás között is. Érdekes, hogy a portok (I/O lábak) is, bár perifériák, ilyen maximális sebességgel működnek.

PBCLK perifériák: A perifériák: SPI, UART, ADC, RTCC, I2C stb., amelyek kisebb sebességgel működnek, a **perifériabusz**-ra kapcsolódnak. Ennek az órajele: **PBCLK (Peripheral Bus CLock)**. Ennek a frekvenciája megegyezhet a SYSCLK-éval, vagy annak fele, negyede vagy a nyolcada lehet. A perifériabuszt és a buszmátrixot egy szinkronizáló (és lassító) **perifériahíd** áramkör kapcsolja össze. Ha a PBCLK és SYSCLK azonos sebességű, akkor a CPU és a többi buszmester egy ciklus alatt éri el a PBCLK perifériákat. Ha a PBCLK = SYSCLK/8, akkor 8 ciklus kell. A CPU egy ciklus alatt elintézi a perifériába írást, folytatja a végrehajtást, a buszmátrix biztosítja, hogy az írás a perifériába 8 ciklus alatt megtörténjen.

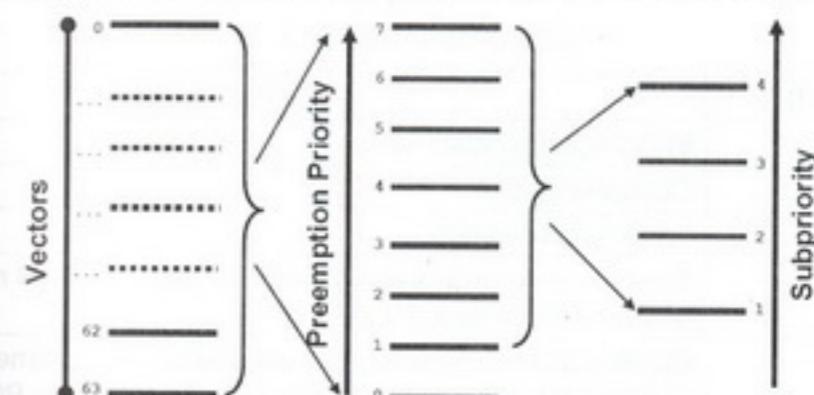
A buszmátrixnak van még egy lényeges tulajdonsága. Mivel egy intelligens, nagy sebességű kapcsoló, ezért amikor egy buszmester elindít egy átvitelt, a buszmátrix létrehozza a pont–pont kapcsolatot a mester és a célmodul között. Míg ez az átvitel zajlik, egy másik buszmester is indíthat egy másik átvitelt egy másik célmodullal. Például miközben a CPU utasításokat hív le a flash-ból a kapcsolódó cache felhasználásával, addig az USB az SRAM-mal cserél (ír vagy olvas) adatot, és ugyanez zajlik a DMA és az UART között is.

Ha közben a CPU is el kívánja érni az SRAM modult, akkor ezt a konfliktust a buszmátrix oldja meg úgy, hogy előbb az egyik folyamat befejeződését várakoztatja. A buszmesterek prioritását a buszmátrix regisztereinek a programozásával tudjuk megvalósítani.

A PIC32 egy nagy teljesítőképességű szorzó és osztó egységet használ, aminek saját pipeline egysége van. Ez azt eredményezi, hogy miközben a CPU szorzás/osztás műveletet végez, folytathatja a következő utasítások lehívását és végrehajtását. Ha a CPU-nak az utasítások végrehajtása közben szüksége van egy éppen számolt operandusra, akkor megvárja az eredményt, és utána folytatja a végrehajtást. A 16*16 és 32*16 bites szorzás 1 ciklus alatt hajtódik végre. Az osztás 11–32 ciklust igényel, ami az osztó és az osztandó méretétől függ.

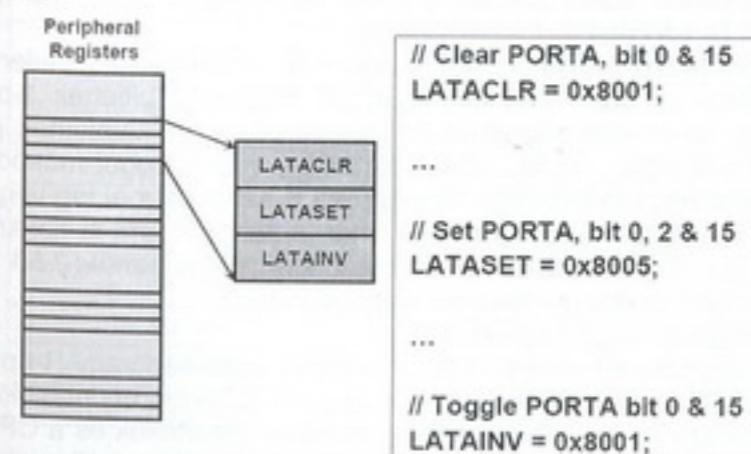
Bár a CPU 32 bites utasításokat hajt végre, képes 16 bites utasításkészlet használatára is. Ez a MIPS16e utasításkészlet és programok méretének a csökkenésével jár.

Megszakítások: A PIC32 egy nagyon rugalmas megszakításvezérlőt használ. Egyszeres (single) és többszörös (multi) vektormódra programozható. Egyszeres módban minden megszakítás egy közös megszakítási címet (vektort) használ (mint a 8/14 családban). Multi mód esetén 64 elemű vektortáblát használ (hasonlóan a 16/24 bites PIC architektúrához), és minden vektorhoz 8 különböző prioritási szint tartozik. (7 a legnagyobb, 1 a legkisebb prioritás, 0 jelzi, hogy az adott megszakítás tiltva van. A magasabb prioritású megszakítások az alacsonyabb szintüket megszakíthatják, de a kisebb szintüek nem. A prioritás mellett létezik egy 4 szintű szubprioritási szint is, ami dönt az egy időben jelentkező, azonos prioritású megszakítások kiszolgálási sorrendjéről. Mivel a megszakítási vektorok báziscíme változtatható, a megszakítási vektortábla a flash-ben bárhol elhelyezhető.

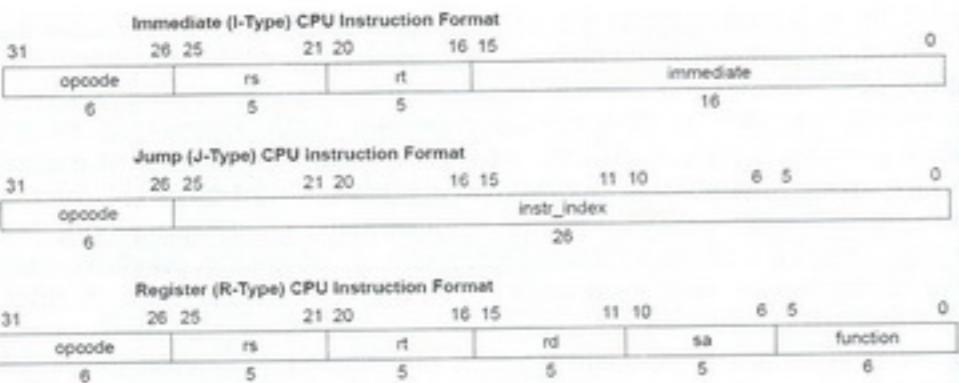


2.51. ábra
PIC32 megszakítás

A 7-es szintű, legmagasabb prioritás esetén még egy **árnyékregiszter-készlet** is működik. Normál esetben, amikor a prioritás kisebb mint 7, a CPU egy elsődleges regiszterkészletet használ. Ha 7-es prioritású megszakítás következik be, akkor a megszakításvezérlő automatikusan átkapcsol az árnyékregiszter-készletre, és utána történik a megszakítás kiszolgálása, majd a visszakapcsolás az elsődleges regiszterkészletre. Ez biztosítja a legmagasabb prioritási szintű megszakítások gyors, szoftveres regiszterelmentés -visszatöltés nélküli kiszolgálását.



2.52 ábra
Gyors bitműveletek



2.53. ábra
PIC32 utasítástípusok

| MEZŐ | BIT | LEÍRÁS |
|-------------|-----|---|
| opcode | 6 | Elsődleges műveleti kód. |
| rd | 5 | Célregiszter-cím. |
| rs | 5 | Forrásregiszter-cím. |
| rt | 5 | Specifikálja a forrás/cél regisztert, vagy a REGIMM műveleti kódhoz tartozó funkciót jelöli ki. |
| immediate | 16 | Előjeles állandó, amit logikai operandusoknál, aritmetikai előjeles operandusoknál, load/store címbájt eltolásánál és PC relatív ugrásoknál használunk. |
| Instr_index | 26 | Index balra eltolva két bittel adja az ugrási célcím alsó 28 bitjét. |
| sa | 5 | Eltolás mértéke. |
| function | 6 | Funkció megadása SPECIAL elsődleges műveleti kód esetén. |

Gyors bitműveletek: Mikrovezérlőknél fontos a portok, illetve az I/O lábak gyors kezelése. Ez a legtöbb 32 bites mikrokontrollernek nem az erőssége. A PIC32-nél alkalmazott megoldás ezen segít. Egy regiszterkészletet használ minden porthoz. Ezek a SET, CLEAR és INVERT elnevezésűek. Ezért például a LATA SFR regiszter kibővül még három SFR regiszterrel: LATACLR, LATASET, LATAINVERT.

A portlábakra vonatkozó műveletet ezek után nem a portláb közvetlen változtatásával valósítjuk meg, hanem az adott „műveletregiszter” megfelelő bitjének 1-be írásával, amit utána a hardver egy órajel alatt végrehajt. Ezt hívjuk atomi bitműveletnek (*atomic bit manipulation*). Mint már említettük, az I/O portok SYSCLK sebességgel működnek. Az INV regiszter alkalmazásával egy labbit billegtetése SYSCLK sebességgel fog végrehajtódni.

Utasítások: Az utasítások magyarázatánál már leírtuk azt, ami itt konkrétan is megjelent: a processzor három különböző formátumú utasítástípust használ (2.53. ábra):

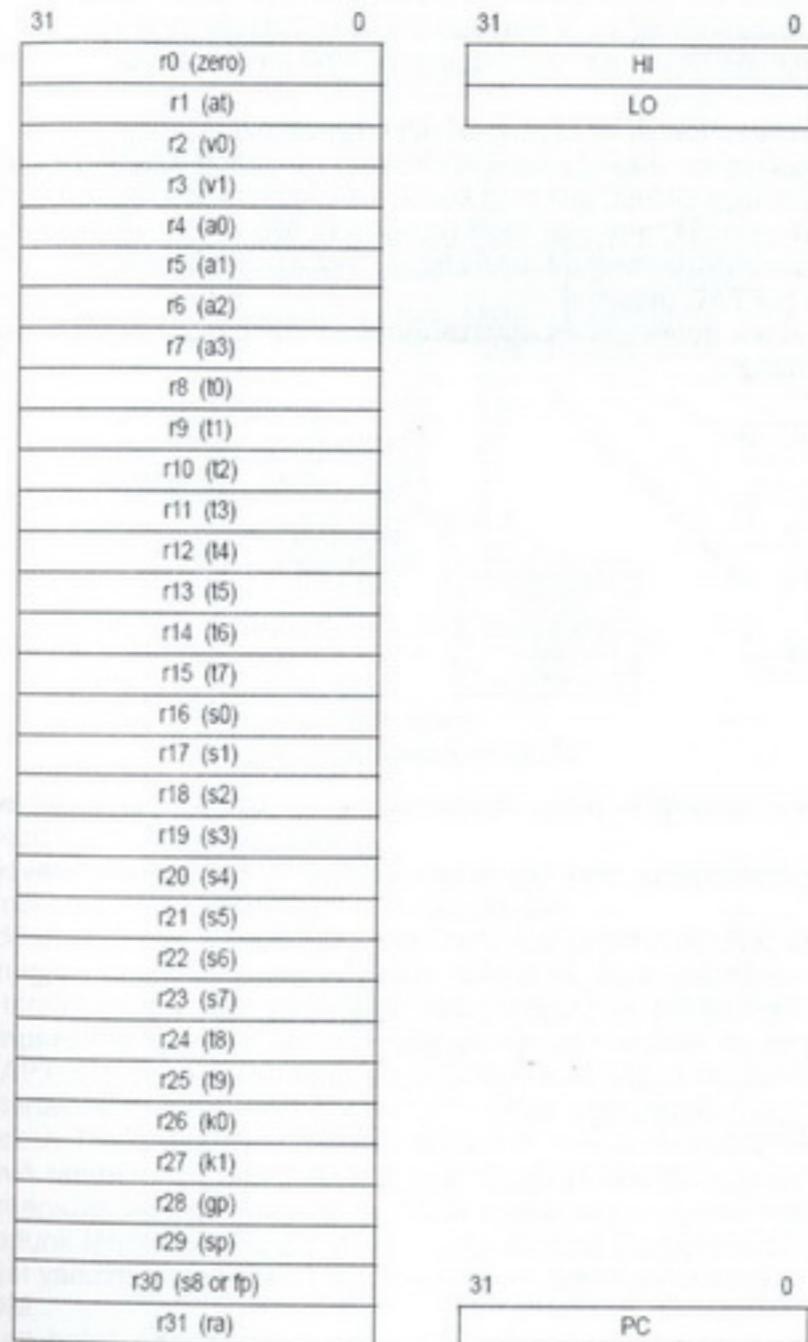
- literált használó utasításokat (immediate vagy I-típust),
- ugró utasításokat (jump vagy J-típust) és
- az adatmemória regisztereit használó utasításokat (registered vagy R-típust).

A legtöbb műveletet regiszterekkel hajtjuk végre. Az ilyen utasításoknak három operandusuk van: két forrás- és egy céloperandus. Ezek az utasítások és a CPU-ban lévő nagy regiszterkészlet teszi lehetővé a hatékony assembler és magas szintű nyelven történő programozást. Gyorsabb és rövidebb programokat írhatunk, mivel a közbenső eredményeket a CPU regisztereiben tarthatjuk, vagyis nem kell folyamatosan a külső memóriába, illetve abból történő írással/olvasással bajlódni.

2. fejezet: A PIC mikrovezérlők felépítése, fejlődése

A műveletek 32 bites operandusait a 32 regiszterből álló tömb címeivel azonosítjuk, minden művelet ezek között zajlik. A szorzás és az osztás eredménye a <HI:LO> regiszterpárban jelenik meg. Az I típusú utasításoknak, van egy immediate (konstans) operandusa, valamint egy forrás- és egy céloperandusa. A J típusú ugró utasítás 26 bites relativ címet használ.

A PIC32 családdal történő programfejlesztést már kizárolag C-ben célszerű végezni, és számos elkészített programmodul segíti a fejlesztést.



General Purpose Registers

Special Purpose Registers

2.54. ábra
PIC32 programozói modell

2.11. ÖSSZEFOGLALÁS

A folyamatos mennyiségi fejlődés mellett, amit a program- és adatmemóriák, adatszélesség, órajel-frekvencia folyamatos növekedése is mutat, az **utasítások hardveres egybekapcsolása** is egyre markánsabban megjelenik.

Ilyen összekapcsolás először a szubrutinhívással kapcsolatosan jelent meg. Hiszen a szubrutin kezdőcímére ugrás megvalósításakor meg kellett valósítani a cím PC-be töltését, valamint a visszatérési cím verembe mentését és a veremmutató változtatását.

A PIC18F mikrovezérlőknél ez a mechanizmus az indirekt címzésnél és a táblakezelésnél is megjelent. Az utasítások az írás/olvasás művelet mellett a mutatók átállítását is elvégzik.

A 16 bites mikrovezérlőknél ez tovább bővült a cikluskezeléssel:

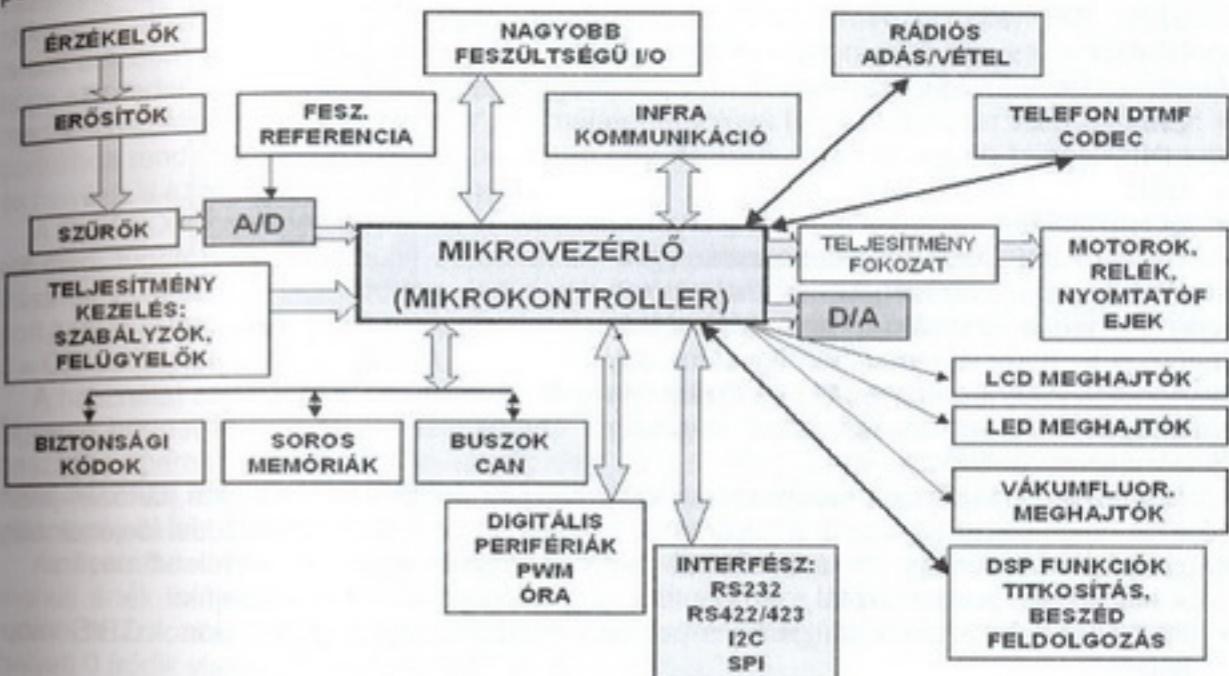
- A **REPEAT** utasítás az utasítás paramétereinek megadott számszor hajtja végre a következő utasítást mint ciklust, belsőleg kezelve a ciklusváltozó figyelését, illetve
- a fejlettebb **DO** utasítás, ami már több utasításból álló ciklusmag végrehajtását végezi a hardveresen. Ide tartozik még a barrel shifter, ami az utasításban megadott számú bit-tolást végez el (**SFTAC** utasítás).

Ezek a megoldások gyorsítják és egyszerűsítik az utasítások végrehajtását, és növelik a számítási teljesítményt.

3. DIGITÁLIS PERIFÉRIÁK

A 3.1. ábrán láthatók a mikrovezérlőkben vagy a környezetükben alkalmazott leggyakoribb periféria-áramkörök.

Röviden összefoglaljuk ezeket a periféria-áramköröket. Célszerű ezeket felépítésük és felhasználási területük miatt három csoportra osztani: digitális, analóg és kommunikációs perifériára.



3.1. ábra
Mikrokontroller-perifériák

A digitális perifériákra jellemző, hogy működésük során kétállapotú jelekkel dolgoznak. Ezeket a következő kategóriába sorolhatjuk:

- **BE/KI (I/O) kivezetések:** a tokok legtöbb kivezetése ilyen perifériaelem, a kivezetések általában bemenetként és kimenetként is használhatók.
- **Számlálók/időzítők:** ezek a perifériák tárolókból kialakított számláló áramkörök, a bemenetükön megjelenő impulzusokat képesek számlálni, és a számlálás eredményét egy regiszterben tárolni, amit a mikrovezérlőben futó program tud felhasználni.
- **Capture/Compare/PWM modulok:** működésükhez számlálókat és regisztereket használnak fel. CAPTURE (kiolvasás): Egy lábon fellépő szintváltás hatására a TMR1 számláló 16 bites tartalmát a <CCPRxH:CCPRxL> 16 bites regiszterbe írjuk. COMPARE (összehasonlítás): A TMR1 számláló 16 bites tartalmát a <CCPRxH:CCPRxL> 16 bites regiszterben lévő tartalommal hasonlítjuk össze. Egyezés esetén egy lábon kiadunk egy jelet, illetve megszakítást generálunk. A PWM modul segítségével impulzusszélességet modulációt tudunk létrehozni. Ez azt jelenti, hogy állandó T periódusidő mellett a bekapsolás W idejét változtatva a kimenő jel középrtékét tudjuk változtatni.
- **Adat EEPROM**

A tokba épített belső adatmemória programból írható-olvasható, és a tartalmát képes megőrizni a tápfeszültség kikapcsolása után is.

Az analóg átalakító perifériák képesek a külvilág felől érkező analóg jeleket digitális bit-csoportra alakítva a mikrovezérlőben futó program segítségével feldolgozni. Ezek:

- A/D-modul
 - analóg komparátormodul
 - alacsony tápfeszültség figyelés (LVD)
- A harmadik csoport a **kommunikációs perifériák** családja, szerepük folyamatosan nő, mert az egységek egymással való kapcsolatát biztosítják.
- SSP – I2C1
 - SSP - SPI
 - USART
 - párhuzamos szolgaport
 - CAN
 - LIN
 - USB
 - ETHERNET

Fontos tudni, hogy a legtöbb periféria működése szoftver segítségével is megvalósítható. Például a számláló funkció megvalósítható úgy is, hogy folyamatosan figyelünk egy bemeneten történő 1-0 vagy 0-1 változást. Ha ez megtörténik, akkor egy belső regiszter tartalmát 1-gyel növeljük.

Akkor miért használunk mégis számlálót, amelynek órajelét használjuk a változás jelzésére? A magyarázat egyszerű: a folyamatos figyelés a futó program működési idejének elég nagy részét felhasználja, és kevesebb idő marad az egyéb programrészletek futtatására.

A perifériákat programozási szempontból négy bitcsoporttal jellemezhetjük:

- **Parancsbitek:** ezekkel állítjuk be a periféria működésmódját (pl. I/O portok TRIS iránybitjei);
- **Státusbitek:** ezeket a periféria állítja, jelezve valamelyen állapotát (pl. az időzítő/számláló túlcordulását jelző bit);
- **Bemeneti adatbitek:** ezek érkeznek a külvilág felől a perifériába;
- **Kimeneti adatbitek:** ezeket küldi ki a periféria a külvilágba.

A perifériával való kapcsolattartás az üzemmódot beállító, tevékenységet indító **parancsbitek**, illetve a periféria állapotát jelző **státusbitek** lekérdezése segítségével történik. A **perifériák állapotának figyelése** két módon valósulhat meg:

- a futó programhurokban folyamatosan figyeljük a státusbit(ek) állapotváltását, ezt **pollingnak** hívják (folyamatos állapotfigyelés);
- a státusbit(ek) állapotváltása **megszakítást** okoz.

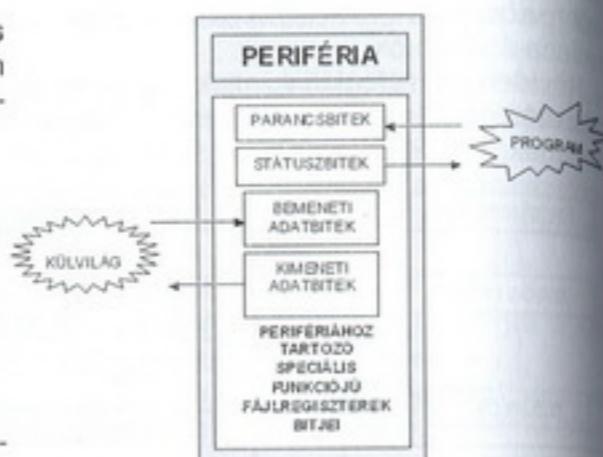
Természetesen ezek a bitek nem önállóan, hanem a PIC-eknél 8 vagy 16 bites regisztekbe szervezve jelennek meg. A bit szabványos (Microchip által definiált) elnevezését és a bitet tartalmazó regiszter nevét, felosztását az adatlapok tartalmazzák.

A szabványos elnevezéseket kell használni, hogy a programot fordító assembler ezeket felismerje.

A következőkben a legfontosabb PIC perifériákat mutatjuk be, alkalmazva az alapelveinket a működés megértése a fontos, a részletek megtalálhatók az adatlapokon. A perifériák között:

- vannak a minden tokban megtalálható közös elemek („szériatartozékok”). Ezek: I/O lábak, 8 bites számláló/időzítő (RTCC vagy TMR0), watchdog (WDT);
- vannak a családvel megvalósító perifériák: TMR1, TMR2 számlálók, A/D, belső adat EEPROM, soros I/O (SCI vagy USART) PWM, CAPTURE/COMPARE modulok, párhuzamos SLAVE port, I2C/SPI interfészű szinkron soros port (SSP), LCD modul stb.

A perifériák a PIC-ek fejlődése során nem változtak jelentősen, a legtöbb 8 bites adatokkal dolgozik, ezért a 16 bites adathosszúság sem jelent lényeges előnyt. A 32 bites adat-



3.2. ábra
Perifériák bitcsoportjai

3. fejezet: Digitális perifériák

szélességű mikrovezérlőben pedig a 16 bites architektúrára kifejlesztett perifériakészletet használták fel. Ezért minden periféria bemutatása során röviden leírjuk azokat a fejlesztések, módosításokat, amelyek az újabb típusokban megjelentek.

3.1. I/O PORTOK

A leggyakrabban használt perifériaelemek az I/O portok. Lényegében a kontroller lóból vezetéseinak állapotát tudjuk a programba beolvasni, illetve a programból a 0 és 1 állapotnak megfelelően, a lóbak feszültségét befolyásolni. Mivel a mikrovezérlökben lévő perifériák is sok esetben kivezetésekkel igényelnek, ez egy adott I/O lóból csak úgy valósítható meg, hogy az eredeti I/O lóból funkció megszűnik, és helyette az alternatív periféria funkció jelenik meg a kivezetésen. Ezt az alternatív használatot lehetővé tévő multiplexer valósítja meg. A portokhoz rendelt I/O regisztereit pontosan úgy kezeljük, mint az egyéb fájregisztereit, és helyileg is azok között helyezkednek el.

A PIC16XXX családnál a portok olvasásakor minden a lóból állapotának beolvasása történik meg, függetlenül attól, hogy bemenetnek vagy kimenetnek van konfigurálva. A portok írása ténylegesen egy D tárolóba (adatlatch-be) történő beírás. RESET hatására minden port bemenet lesz, és a TRIS regiszter tartalmával tudjuk a port irányát beállítani (1 = I[put], 0 = O[put]).

A használat szempontjából fontos a „tárolóba írunk és lábról olvasunk” megállapítás. Ugyanis bármelyik utasítás, amely az I/O regisztereit valamelyiket vagy valamelyik bitjét használja operandusként, a művelet végzése előtt a teljes 8 bites értéket olvassa be a lábról (READ), elvégzi a műveletet (vagyis módosítja a regisztert – MODIFY), majd azt írja vissza az adattárolóba (WRITE).

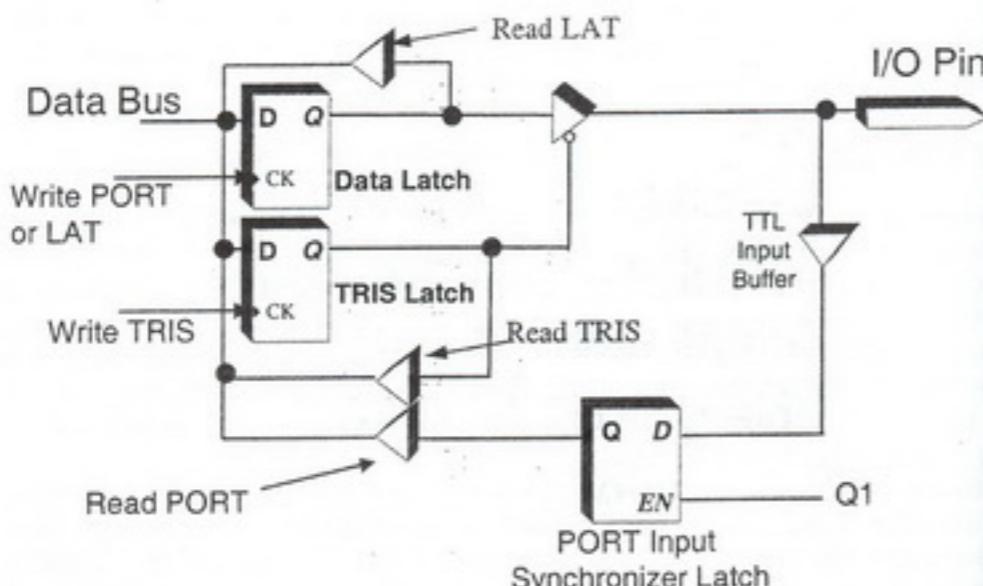
Az ilyen **Read-Modify-Write utasítások** használatakor egy már előzőleg 1-be állított kimeneti érték is megváltozhat, ha a külső áramkör az 1-be állított kimeneti lábat alacsony szintre húzza, akkor az visszaolvasáskor 0-ként jelenik meg, és visszaíráskor az eredeti 1 helyett 0 íródi vissza [AN521, AN528, AN529, AN566].

Több típusnál a bemenetként konfigurált I/O portlábakra (ez általában a PORTB) egy **beli felhúzó ellenállást** kapcsolhatunk, amivel a bemeneti állapotot stabil 1 helyzetben tarthatjuk, és ezt húzzuk le nullára egy, a bemenetre kapcsolt nyomógombbal vagy kapcsolóval.

8/16 A 16XXX típusok B portjának felső négy bitje RB<7:4> megszakítás kiváltására is képes. Ehhez bemenetként kell konfigurálni a portlábakat. Ha ezek közül valamelyiket kimenetként konfiguráljuk, akkor az a lóból nem használható erre a célra. A négy portláb aktuális értékét a CPU egy előbbi értékkal hasonlíta össze, és ha változás történt, akkor megszakítás történik (RBIF = INTCON.0). A megszakítás törléséhez vagy tilljuk a megszakítást (INTCON.3 = 0), vagy az RB portról egy olvasást hajtunk végre, és RBIF-et töröljük.

PIC18XXX be- és kimeneti portok

8/16 Mivel a read-modify-write utasítások miatt a portlábak állapota megváltozhat, (tárolóba írunk, és közvetlenül a lábról olvasunk), ezért a portok felépítését megváltoztatták. Minden port három tárolót tartalmaz. A portlábak irányát lovábbra is a TRISx regiszter határozza meg. A lábra történő írás a **Data Latch – LATx** regiszterbe történő írással valósul meg. A **PORTx** regiszter olvasásakor a lábon lévő értéket kapjuk meg:



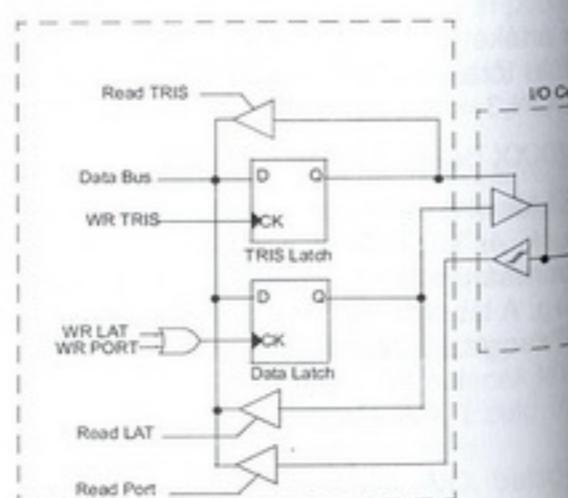
3.3. ábra
8/16 család I/O áramköre

- A LATx regiszter olvasása annak tartalmát adja (amit a portra kiírtunk).
- A PORTx olvasásakor a láb állapotát kapjuk, amit Q1 ütemben a rendszer mindenig a lábról frissít.
- a PORTx vagy a LATx írása mindenig frissíti a LATx regiszter tartalmát, ezért a CLRF LATx, illetve CLRF PORTx utasítások egyenértékűek, hiszen valójában ugyanarról a tárolóról van szó. (A PORTx írásának csak elvi jelentősége van, hiszen inkább olvasni szoktuk...)
- A LATx regiszter teszi lehetővé a portbeolvasást igénylő read-modify-write utasítás végrehajtását, anélkül, hogy a kimenet helyes állapotának megőrzése miatt a portra kiküldött értéket egy tükröregiszterben tárolnánk. Vagyis a read-modify-write utasítások a LATx regiszter használják!

Szabály: a kimeneteket LAT biteknek, a bemeneteket PORT biteknek definiáljuk.

16/24 A 16 bites adatszélességű PIC mikrovezérlőknél hasonló a megoldás, azzal a különbséggel, hogy a külvilág egy Schmitt trigéres bemeneti fokozaton keresztül jut el a mikroprocesszorba. Ez lassan változó bemeneti jelek (például egy hőelem kimenete) helyes detektálását teszi lehetővé.

Természetesen a lábakhoz hozzárendelt multiplexerrel vezérelt esetleges alternatív funkciók megmaradtak.



3.4. ábra
16 bites PIC I/O

3.1.1. Bemeneti változások kezelése a 16/24 bites kontrollereknél

Az előzőekben leírt RB4-7 bemeneteken megjelenő jelváltozás automatikus érzékelését jelentősen továbbfejlesztették. Ez a lábakhoz rendelt CN (Change Notification ~ változás jelzése) tulajdonság. A tok lábszámától függően, max. 24 bemenet esetén, azok megváltozásakor megszakítást generálhat.

Minden, CN-re alkalmas lábat két bittel kezelünk: CNxIE bit engedélyezi (= 1), hogy az adott x láb változása megszakítást generáljon, míg a CNxPUE bittel a bemeneten lévő felhúzó ellenállást kapcsolhatjuk be.

CN használata:

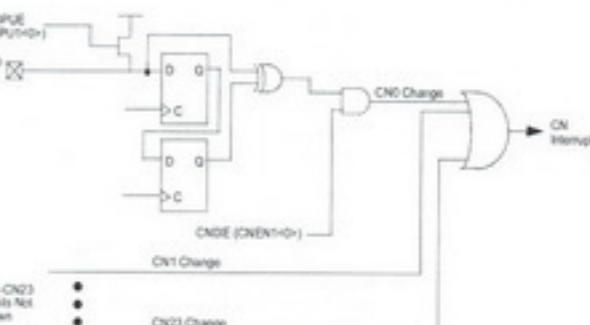
- A kiválasztott CN lábakat bemenetnek állítjuk be, a hozzárendelt TRIS regiszter megfelelő bitjeinek 1-be állításával.
- A kiválasztott CN lábakhoz tartozó megszakítást engedélyezzük: CNxIE = 1.
- A kiválasztott CN lábakhoz tartozó felhúzó ellenállást, ha szükséges, engedélyezzük: CNxPUE = 1.
- Töröljük a CNIF (IFS0<15>) megszakításjelzöt.
- Kiválasztjuk a kívánt megszakításprioritást a CN megszakítás számára :CNIP<2:0> vezérlőbitek (IPC3<14:12>).
- Engedélyezzük a CN megszakítást CNIE (IEC0<15>) = 1.
- Mikor a megszakítás bekövetkezik, be kell olvasnia a CN-hez tartozó PORT regisztereit. Ez töri a változást, és így lehet érzékelni majd a következő állapotváltozást. A jelenleg beolvasott PORT értéket összehasonlíva az előző CN megszakításkor olvasott értékkel, meghatározhatjuk azt, hogy melyik láb változott. A specifikációkban leírtak adják meg, hogy milyen rövid lehet az a lábon megjelenő impulzus, amit még detektálni lehet.

A CN modul a processzor IDLE és SLEEP állapotában is működik. A lábakon fellépő állapotváltozás megszakítást okoz, ami a tokot felébreszti.

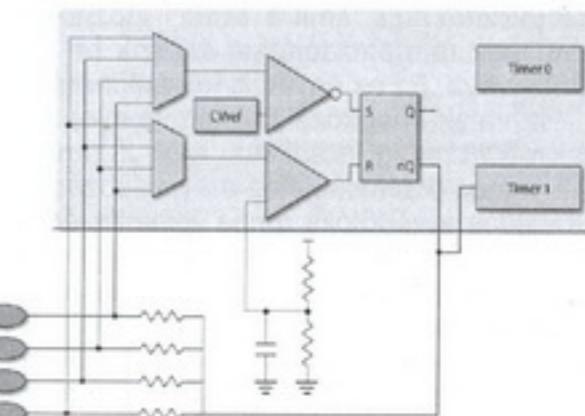
3.1.2. Az érintésérzékelők kezelése

Az érintésérzékelés ma már a hagyományos nyomógombok alternatívájává vált, gondoljunk a legújabb mobiltelefonokra (iPhone) vagy az MP3/MP4 lejátszókra. Nincs mechanikai mozgás (kopás), teljesen zárt, könnyen tisztítható. Ezek a kijelzők az orvostechnikában, járműveknél, az iparban is népszerűek lesznek, mert olcsóak és gyakorlatilag örök életűek.

A PIC mikrovezérlők néhány típusának bemenetei alkalmasak erre a feladatra, ezekről és a megoldásokról a www.microchip.com/mtouch oldalon olvashatunk.



3.5. ábra
CN működés



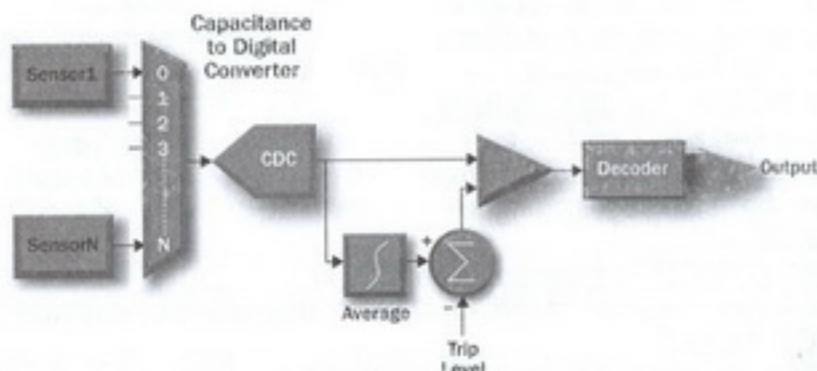
3.6. ábra
A megoldás érzékelőket, oszcillátort és időzítőt használ

Hogyan működik?

Van egy kondenzátorunk, ami két, egymástól elszigetelt vezető felületet tartalmaz. Ezt a két felületet akár nyomtatott áramkörön is ki lehet alakítani, a vezető anyag pedig akár az emberi vér hemoglobinjában lévő vas is lehet. Mivel a környezethez képest mindenben van szort kapacitása (test és föld között), ezért amikor valaki az ujjával közelít a felülethez, annak a kapacitása megváltozik (megnövekszik).

Egy kapacitív elven működő rendszer ezért megfelelő módon kialakított érzékelő felületeket igényel és olyan mérést, amivel a kapacitás megváltozását tudjuk érzékelni és megfelelő módon kiértékelni. A cégek **mTouch** fantázianevű megoldása ezt úgy éri el, hogy a kapacitásváltozás egy oszcillátor frekvenciaváltozásában jelenik meg. Így ha mérjük a frekvenciat, a megváltozásának nagysága jelzi, hogy valaki megközelítette a vezető felületeket.

Az áramkori megoldás a 3.7. ábrán látható.



3.7. ábra
Az mTouch megoldás

Az áramkör legfontosabb része a relaxációs oszcillátor. A mikrokontroller ennek minden elemét tartalmazza, kivéve a szükséges ellenállást. A vezető anyagból lévő érzékelő felület gyakran beborítják valamelyen szigetelő anyagú fóliával.

A kialakított oszcillátor kb. 100 kHz nagyságrendű, amit az ellenállással állítanak be. Pontos frekvenciának itt nincs jelentősége. A frekvenciamérés a Timer0 és Timer1 időzítő felhasználásával történik, majd egy program kiértékeli az eredményt.

A frekvenciamérés eredményét a Timer0 időzítő által generált fix időtartam alatt beérkezett jelszám adja, amit a Timer1 időzítő számol meg. Mivel a kapacitás értékét nem tudjuk ezért egy referenciaértéket állítunk be a névleges kapacitáshoz tartozó frekvenciát figyelembe véve. Ez az egymást követő mérések átlagolásával történik.

Ha valaki megközelíti (megnyomja) az érzékelő felületet, a kapacitás megnövekszik, a frekvencia pedig lecsökken, amit a Timer1 tartalmának csökkenése is jelez. Azaz a billentyűnyomás érzékeléséhez ezt a változást kell kiértékelnünk.

Sajnos a védőfólia rontja a mérés érzékenységét, mert ilyenkor kisebb változásokat tapasztunk. Ezért ha lehet, ne használunk védőfóliát.

Ezek után a mérési eredményeket átlagoljuk, általában 16 mérését. Természetesen átlagolással történik az alapfrekvenciához tartozó számérték meghatározása is. Két választható átlagolási módszer lehetséges:

Kapuzott átlagolás (gated average): ennél a módszernél az átlagolás megáll, ha a mérési eredményben hirtelen nagy változás van, míg lassú változás esetén átlagol.

Lassú átlagolás (slow average): Ilyenkor minden időszakban átlagolunk.

Egy külön grafikus program segítségével egy kiválasztott érzékelőt le tudunk tesztelni, amivel az érzékelési készüléket be tudjuk állítani.

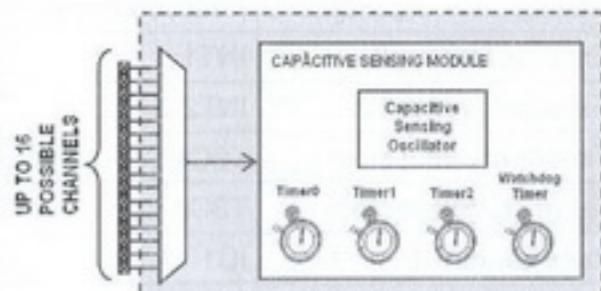
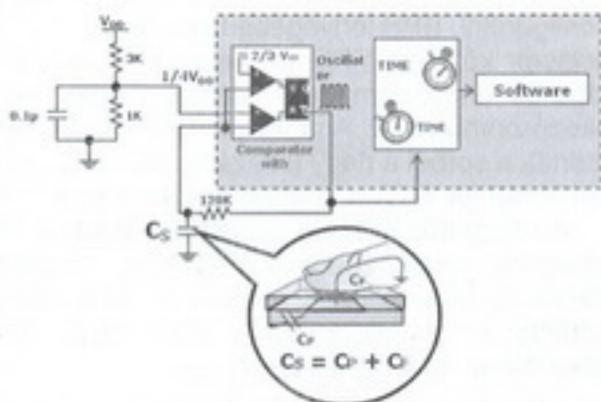
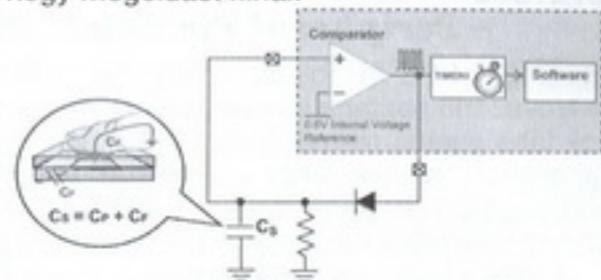
A Microchip a kapacitív érzékelésre jelenleg négy megoldást kínál:

Ha a tok rendelkezik egy komparátor áramkörrel, akkor azzal kialakítható egy relaxációs oszcillátor, aminek a frekvenciája a kapacitás változásával módosul, és ez a változás egy időzítő segítségével mérhető.

Ez a megoldás jól használható, ha csak egy kapacitív érzékelésre van szükség, és célszerű a kis lábszámu PICF10F204/206 családot használni.

Második megoldásként a komparátort kiegészítjük egy beépített S-R tárolóval

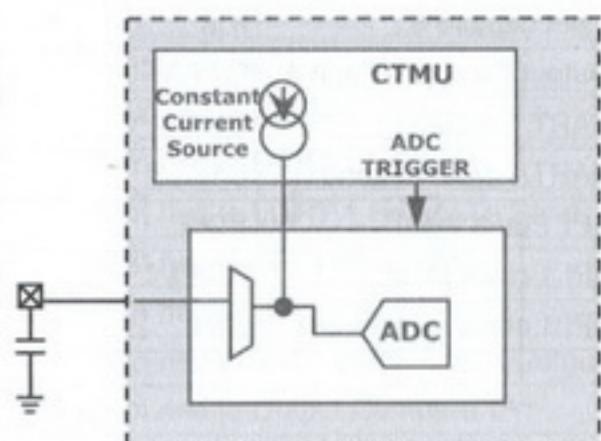
Ez a kiegészítés több lábon is megvalósítható, és a kapacitásváltozás detektálását egyszerűbbé teszi. Több tok is rendelkezik ilyen periféria kiegészítéssel a 8/14-es mikrovezérlőknél, ezek a PIC16F887, PIC16F690 és PIC16F616 típusok. A megoldás néhány kapacitív érzékelő bemenet használatakor előnyös.



A kapacitásérzékelő modul (Capacitive Sensing Module – CSM) használata. Ez minden PIC16F722/3/4/6/7 (PIC16F72X) eszközben megtalálható, és leegyszerűsíti az kapacitív érzékeléshez szükséges áramkörök és programozási feladatot. Az érzékelésre 8 vagy 16 csatorna áll rendelkezésre:

- 16 csatorna a PIC 16F724/727 és PIC 16LF724/727 modellek esetében;
- 8 csatorna a PIC 16F722/723 és PIC LF722/723/726 modellek esetében.

A 16/24 PIC mikrokontroller családtól kezdve megjelent egy külön, töltési időt mérő egység (Charge Time Measurement Unit – CTMU) periféria, amely maximum 16 csatornás lehet, a tokban (PIC24FJ256GA110) egy pontos változtatható áramú áramforrás van. A kapacitív érintés érzékelő a CTMU töltésmérési üzemmódját használja. A mérő rendő kapacitást egy állandó áramú generátorral tölti, és a kapacitás feszültsége lineárisan növekszik. Ezt a feszültséget egy A/D átalakítóval mérjük. Ha minden állandó ideig töltünk, akkor a kondenzátor feszültsége függ a kapacitás nagyságától. Ha csak a kapacitás változása érdekes, (megérintettük a gombot vagy nem), akkor ennek detektálása nem igényel kalibrációt.



3.1.3. Funkciók más lábakhoz rendelése – Peripheral Pin Select (PPS)

16/24 A PPS megoldás lehetőséget biztosít a tervezőnek annak kiválasztására, hogy a mikrovezérlő periféria kivezetései melyik lábra kerüljenek. Ez a multiplexelés a mikrovezérlő sok lába esetén megvalósítható. A megoldás különösen a nagy lábszámú, sok perifériával rendelkező mikrovezérlők esetén hasznos. A korlátozott lábszám miatt ez a multiplexelés eddig is létezett, de fix lábakra történt, ami sokszor problémákhoz vezetett. (Egyik periféria lábhoz rendelése miatt nem lehet egy másik perifériát használni).

A PPS tulajdonságú mikrovezérlőknél sok kivezetés jelölése RPn (*Remappable Peripheral*) amihez perifériakivezetéseket tudunk rendelni, és amit a tok működése közben is át tudunk konfigurálni. PPS lényegében digitális multiplexereket helyez a digitális perifériák és a lábkivezetések közé. A PPS-sel kezelhető perifériák: soros kommunikációt végzők (UART, SPI), időzítők órajel-bemenetei, capture, compare perifériák és a szintváltozáskor megszakítást okozó bemenetek. Analóg perifériák ilyen módon történő használata nem lehetséges. Kimeradnak a sorból a nagy lábszámú perifériák, például a Parallel Master Port. Kimeradnak azok a perifériák (pl. I2C) is, amelyek működése I/O állapotváltást igényel a lábon.

A magyarázatokban szereplő adatok a PPS képességű PIC24FJ64GA004 mikrovezérlő családra vonatkoznak. A PPS max. 72 perifériát tud 26, illetve 16 lábra multiplexelni 44, illetve 28 lábú tokozás esetén. A lábhozzárendelést külön a periféria-bemenetek és külön a periféria-kimenetek esetén elvégezhetjük. Az adott mikrovezérlő esetén ezt a 3.8. és a 3.9. ábra táblázata foglalja össze.

| Név | Function name | Register | Configuration bits |
|--------------------------|---------------|----------|--------------------|
| External interrupt 1 | INT1 | RPINR0 | INT1R[4:0] |
| External interrupt 2 | INT2 | RPINR1 | INT2R[4:0] |
| Timer2 external clock | T2CK | RPINR3 | T2CKR[4:0] |
| Timer3 external clock | T3CK | RPINR3 | T3CKR[4:0] |
| Input capture 1 | IC1 | RPINR7 | IC1R[4:0] |
| Input capture 2 | IC2 | RPINR7 | IC2R[4:0] |
| Input capture 7 | IC7 | RPINR10 | IC7R[4:0] |
| Input capture 8 | IC8 | RPINR10 | IC8R[4:0] |
| Output Compare Fault A | OCFA | RPINR11 | OCFAR[4:0] |
| UART 1 receive | U1RX | RPINR18 | U1RXR[4:0] |
| UART 1 clear to send | U1CTS | RPINR18 | U1CTSR[4:0] |
| SPI 1 data input | SDI1 | RPINR20 | SDI1R[4:0] |
| SPI 1 clock input | SCK1 | RPINR20 | SCK1R[4:0] |
| SPI 1 slave select input | SS1 | RPINR21 | SS1R[4:0] |

3.8. ábra
PPS bemenetek

A PPS-képességet három SFR blokk és egy vezérlőbit kezeli. Ezek három funkciót valósítanak meg: az első bekapcsolja (lock), illetve kikapcsolja (unlock) a láb-hozzárendelés megváltoztatását. A második és a harmadik funkció a bemeneti és kimeneti hozzárendelések megadására szolgál.

Bemenetek hozzárendelése: a perifériához rendelt 5 bites konfigurációs mező tartalmazza, hogy melyik RPn lábhoz rendeljük az adott perifériabemenetet. Az RPINRx regiszterek szolgálnak a hozzárendelések kezelésére. Az ábrán az U1RX bemenethez tartozó illusztráció látható, még a 3.8. ábra táblázata a választható perifériabemeneteket foglalja össze.

Kimenetek hozzárendelése: a bemenetektől eltérően a kimenetek hozzárendelése a lábak alapján történik: a kimeneti vezérlő regiszterben lévő érték határozza meg azt, hogy melyik perifériakimenet melyik RPn lábon jelenjen meg.

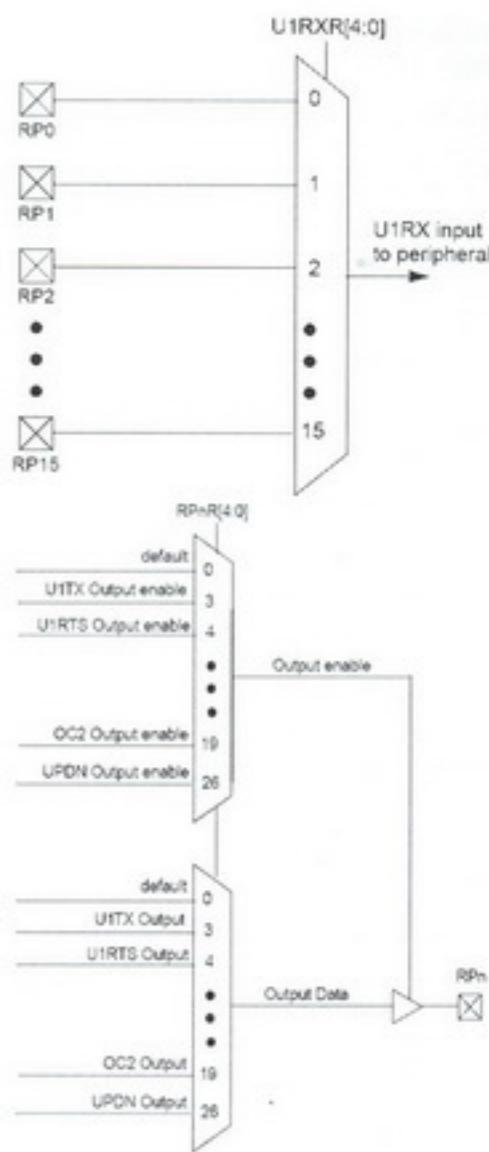
Az RPORx regiszterek szolgálnak a kimeneti hozzárendelések kezelésére. Az RPINRx regiszterekhez hasonlóan minden regiszter egy 5 bites mezőt tartalmaz, ami a perifériálábakat egy RPn lábhoz rendeli.

A perifériák kimeneteinek listája egy 00000 értéket is tartalmaz. Ezt választva az alapértelmezés szerinti perifériakimenet jelenik meg a lábon. Két bit vezérli a funkció-hozzárendelő regiszterek írását. Az első bit, IOLOCK használt arra, hogy megakadályozzuk ezen regiszterek írását.

Ha írni akarunk, akkor az IOLOCK bitet törölni kell, és az OSCON regiszterbe egy írási védelmet kikapcsoló utasítássorozatot kell írni. A második bit neve IOL1WAY, és a CW2 konfigurációs regiszterben található. Ha IOL1WAY bit értéke 1, akkor a hozzárendelő regiszterek csak írhatók, mikor a tok RESET állapotból éled. Mikor IOL1WAY = 1, akkor IOLOCK nem törölhető.

| Function | RPnR<4:0> | Output name |
|----------|-----------|---------------------------------------|
| NULL | 00000 | RPn ried to default port pin |
| U1TX | 00011 | RPn ried to UART 1 transmit |
| U1RTS | 00100 | RPn ried to UART 1 Ready To Send |
| SDO1 | 00111 | RPn ried to SPI 1 data output |
| SCK1OUT | 01000 | RPn ried to SPI 1 Clock Output |
| SS1OUT | 01001 | RPn ried to SPI 1 Slave Select Output |
| OC1 | 10010 | RPn ried to Output Compare 1 |
| OC2 | 10011 | RPn ried to Output Compare 2 |

3.9. ábra
PPS kimenetek



A következő, C nyelvű assemblerbetéket tartalmazó példa szolgál a leírtak illusztrációra. (Hogy konkrétan melyek az RPn lábak, azt a mikrovezérlő adatlapja tartalmazza.)

```
// PÉLDA: UART 1 BE ÉS KIMENETEINEK ÁTKONFIGURÁLÁSA
//*****
// REGISZTEREK VÉDELMÉNEK KIKAPCSOLÁSA (UNLOCK)
//*****

ASM VOLATILE ( "MOV #OSCCONL, W1 \N"
"MOV #0X45, W2 \N"
"MOV #0X67, W3 \N"
"MOV.B W2, [W1] \N"
"MOV.B W3, [W1] \N"
"BCLR OSCCON, 6");
//*****

// BEMENETEK KONFIGURÁLÁSA
//*****
// U1RX RP0 JELÜ LÁBHOZ RENDELÉSE
//*****
RPINR18BITS.U1RXR = 0;
//*****
// U1CTS RP1 LÁBHOZ RENDELÉSE
//*****
RPINR18BITS.U1CTSR = 1;
//*****
// CKIMENETEK KONFIGURÁLÁSA
//*****
// U1TX RP2-HÖZ
//*****
RPOR1BITS.RP2R = 3;
//*****
// U1RTS RP3-HÖZ
//*****
RPOR1BITS.RP3R = 4;
//*****
// REGISZTERVÉDELEM VISSZAKAPCSOLÁSA (LOCK)
//*****
```

ASM VOLATILE ("MOV #OSCCONL, W1 \N"
"MOV #0X45, W2 \N"
"MOV #0X67, W3 \N"
"MOV.B W2, [W1] \N"
"MOV.B W3, [W1] \N"
"BSET OSCCON, 6");

3.2. PÁRHUZAMOS I/O PORT

A 40 kivezetéses, közepes teljesítményű PIC mikrovezérlök általában tartalmaznak egy 8 bites, párhuzamos szolgaportot (PSP – Parallel Slave Port), amelynek segítségével a mikrovezérlő könnyen illeszthető mikroprocesszoros rendszerekbe.

A párhuzamos szolgaport 11 mikrovezérlő-kivezetést használ fel, 8 kivezetés csatlakozik a másik mikroprocesszoros rendszer adatbuszára, a fennmaradó három pedig a vezérlőjeleket fogadja:

- CS – eszközkiválasztó jel (Chip Select), amelynek aktív állapota ("0") kiválasztja az adott mikrovezérlőt (a többi közül) az adatátvitelre;
- RD – adatkiolvasást kezdeményező jel (Read), amelynek aktív állapota ("0") jelzi a mikrovezérlőnek, hogy a mikroprocesszor kéri a soron következő adatot az adatbuszra;
- WR – adatbevitelt kezdeményező jel (Write), amelynek aktív állapota ("0") jelzi, hogy a mikroprocesszor által az adatbuszra írt adat az adott mikrovezérlő számára szól. Az adatbuszra kapcsolódó kétirányú I/O kivezetések TTL bemenetük.

8/16 A jelenleg gyártott mikrovezérlök esetében a párhuzamos slave portot a **PORTD** 8 és a **PORTE** port 3 kivezetése valósítja meg. A bejövő és kimenő 8 bites adatot a **PORTD** regiszterből kell kiolvasni, illetve oda kell beírni.

A párhuzamos slave port a **TRISE** regiszter **PSPMODE** vezérlő bitjének logikai "1"-re állításával engedélyezhető, de emellett biztosítani kell, hogy a párhuzamos slave portot megvalósító kivezetésekre semmilyen más periféria ne legyen engedélyezve.

A párhuzamos slave port áramköre a **CS**, és bármely másik vezérlő jel együttes aktív állapota esetén az adott **PIRx** regiszter **PSPIF** jelzöbitjét logikai "1"-re állítja és, ha a **PIEx** regiszter **PSPIE** engedélyező bitje által engedélyezve van, akkor megszakításérést generál.

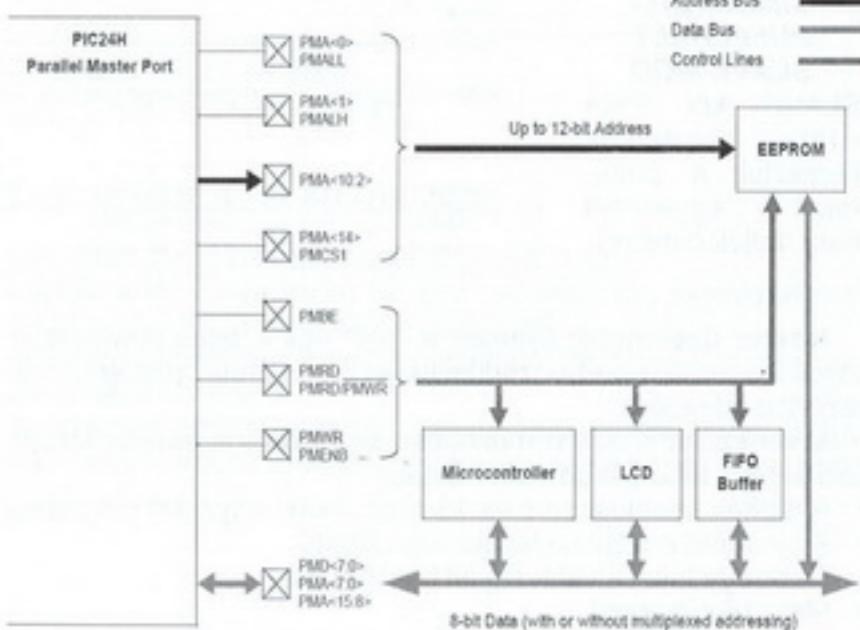
A párhuzamos slave port SLEEP állapotban is működhet. Az adatbevitel, illetve adatkiolvasás megtörténtére generálódó megszakítás kilépteti a mikrovezérlőt a SLEEP állapotból.

16/24 16 bites mikrovezérlőknél ezt a perifériát továbbfejlesztették, és a kommunikációt végrehajtó szolga működés mellett a kommunikációt vezérlő mester funkcióval is ellátták.

A párhuzamos mestерport (**Parallel Master Port – PMP**) 8 bit adatszélességű I/O eszköz, amit párhuzamos adatátvitelt felhasználó kommunikációs perifériákhoz, LCD kijelzőkhöz, külső memoriákhoz és mikrovezérlőkhöz történő illesztésekre fejlesztettek ki. Mivel a párhuzamos perifériákhoz való illesztésnek igen sokfajta megoldása van, a PMP modul nagymértékben konfigurálható.

Jellemzők

- Nyolc adatvonal, maximum 12 programozható címponnal, egy tokat választó vonal.
- Kétfajta vezérlés: külön olvasó és író jelek vagy olvasó/író jel külön engedélyezéssel. Automatikus cím-növelés vagy -csökkenés.
- Programozható Cím/Adat multiplexelés. Vezérlőjelek aktív szintje programozható.
- Régebbi Parallel Slave Port (PSP) támogatása.



3.10. ábra
PMP modul

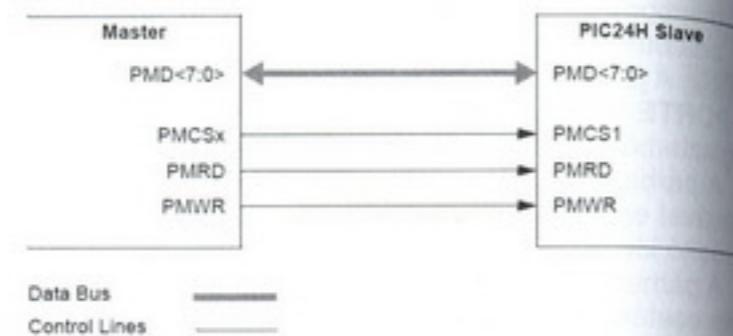
- Továbbfejlesztett PSP támogatása: címtámogatás, négybájtos puffer, programozható várakozó állapot.

A modul képes különféle szolga és mester módok szerinti működésre.

Szolga üzemmód: Ilyenkor a PMP egy 8 bites adatbuszt biztosít a szükséges vezérlőjelekkel. Hárrom szolga üzemmód van: a hagyományos, pufferelt, illetve a címezhető pufferelt üzemmód. Nyolc adatvonalat használ, címezhető módban két címvonalat és három beállítható aktív szintű vezérlővonalat (írás, olvasás, modul kiválasztása).

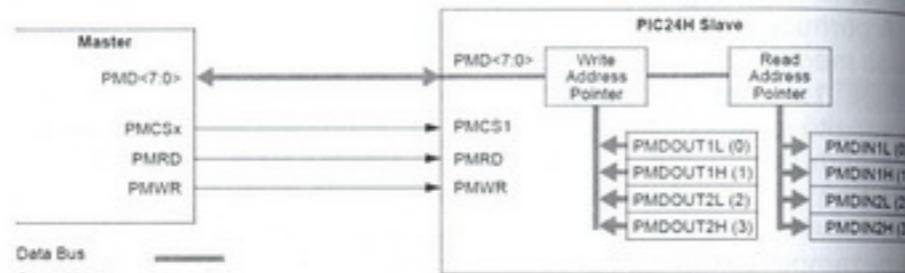
HAGYOMÁNYOS SLAVE MÓD

A mester aszinkron módon használja a modult. Ír bele vagy olvas belőle CS, WR, RD vezérlőjelekkel.



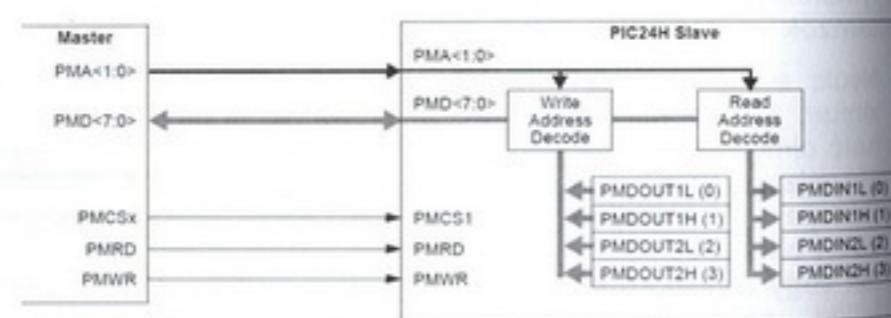
PUFFERELT SLAVE MÓD

Megegyezik az előzővel, de az írás és olvasás a 4 bájtos PDMOUT és PDMIN puffer regisztereiken keresztül történik.



CÍMEZHETŐ PUFFERELT SLAVE MÓD

SLAVE két PMA <1:0> bemenetén keresztül a puffereket egyenként meg tudjuk címezni.

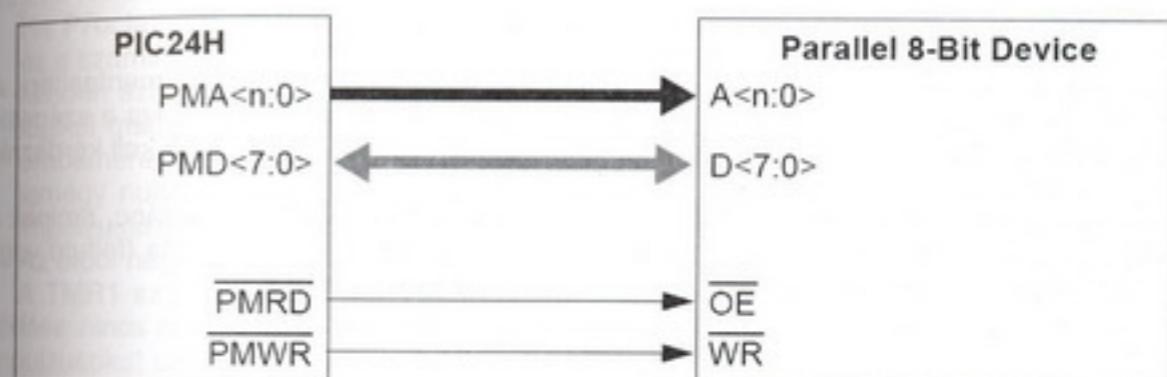
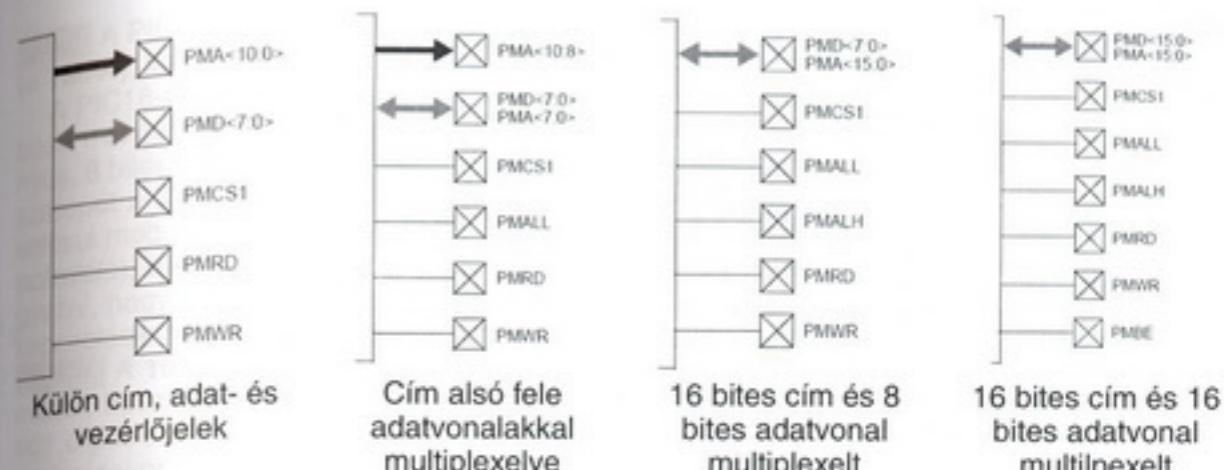


Mester üzemmód: Ilyenkor a PMP egy 8 bites adatbuszt mellett egy 16 bites címbuszt biztosít a szükséges vezérlőjelekkel, memóriák, perifériák és szolgaként működő mikrovezérlök elérésére.

Mivel számos, eltérő működésű párhuzamos eszköz létezik, a mester konfigurálásának sokkal több lehetősége van. Például:

- A 8 bites adatbuszon 8 és 16 bites átvitel egyaránt megvalósítható.
- A cím/adat multiplexelés konfigurálható.
- Csak egy tokot kiválasztó jel használata.
- Max. 16 címvonal.
- Címek automatikus csökkentése/növelése 1-gyel.
- minden vezérlővonal aktív állapota beállítható.
- Programozható várakozási idő a műveletek között.

A következő táblázatban összefoglaltuk a mesterként történő konfigurálás néhány lehetőségét, a 3.11. ábrán pedig egy példán is bemutatjuk.

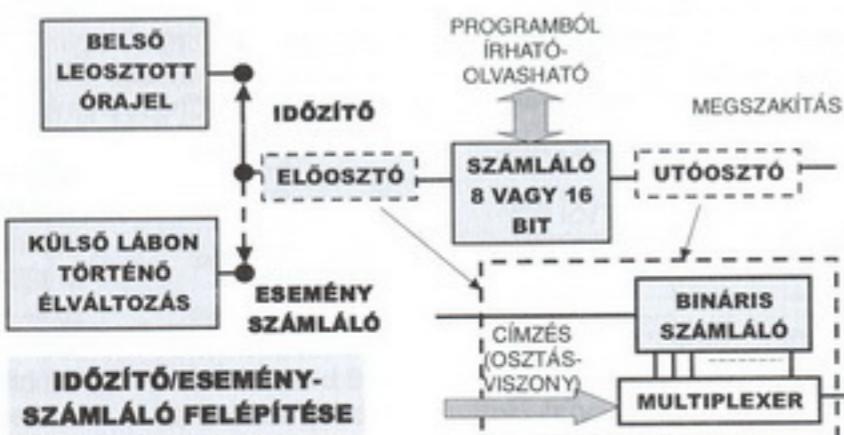


3.11. ábra
Példa: EPROM illesztés

3.3. IDŐZÍTŐ/ESEMÉNYSZÁMLÁLÓ MODULOK

A Mikrochip a számlálók fejlesztése során három típust határozott meg:

- „A” típusú számláló: működhet a kis fogyasztású 32 kHz oszcillátorral, aszinkron módon, külső órajelforrást használva. Ilyen a Timer1 jelzésű számláló.
- „B” típusú számláló: C típusú számlálóval összekapcsolható 32 bites számlánccá. Ilyenek a Timer2 és a Timer4 jelzésű számlálók.
- „C” típusú számláló: B típusú számlálóval összekapcsolható 32 bites számlánccá. Egy adott mikrovezérlőben legalább egy C típusú számlálónak megvan az a képessége, hogy A/D konverziót indítson el.

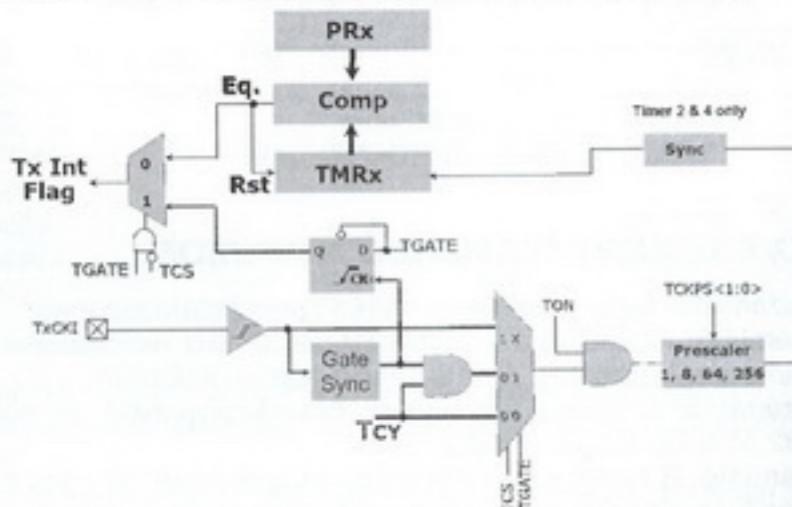


3.12. ábra

Időzítő/eseményszámláló blokkvázlata

A számláló áramköröket perifériaként használva a processzort tudjuk tehermentesíteni. A számláló bemenetére egy külső esemény esetén megjelenő jelváltást kapcsolva a számláló önállóan képes a külső események számlálására, és a processzornak csak le kell kérdeznie a számláló tartalmát.

Foglaljuk össze a működésüket a 3.12. ábrán! Lényegében egy számlálólánc, aminek a léptető impulzusa vagy egy I/O lábon megjelenő külső feszültségszint-változás (felfutó vagy lefutó él), vagy a tokot működtető oszcillátor leosztott frekvenciájú jele.



3.13. ábra

16 bites mikrovezérlők számláló áramköre

Általánosságban a 8 vagy a 16 bites bináris számláló léptetőjelét egy előosztón keresztül kaphatja, és a kimenete is egy esetleges utóosztón halad még át. Az elő-, illetve az utóosztó értéke a periféria konfigurálásával állítható be.

Ha kontrollert működtető leosztott órajelet juttatunk erre a számlálóbemenetre, akkor a számláló tartalma és az órajel periódusidejének a szorzata az eltelt idővel arányos. A számláló által átfogható számlálási tartomány növelése miatt a számlálót alkotó flip-flopokból álló számlálólánc hosszát növelik meg. Például nem 8 bites, hanem 16 bites számlálót használnak: ennek mind a felső, mind az alsó 8 bitje elérhető a programból.

A következőkben tekintsük át a PIC-eknél használatos számláló/időzítő áramköröket. Három típusuk van, ezek elnevezése: TMR0, TMR1, valamint TMR2.

8/16 A PIC18-as család már öt számlálót tartalmaz, ez a 8/16 bites TMR0, a 8 bites TMR2 (= TMR4), illetve a 16 bites, egymással csaknem azonos TMR1, TMR3 modulok.
A PIC18-as családban a TMR0 modul is már 16 bites számlálóval működik. A régebbi 8 bites működéssel való kompatibilitás fenntartása érdekében, alapállapotban a hagyományos, 8 bites egységként funkcionál. A konfigurálásához szükséges OPTION_REG regiszter szerepét a T0CON regiszter vette át. A 16 bites működés a T0CON<T08bit> = 1 esetén valósul meg. Ilyenkor valójában egy 24 bites számlálót kapunk (8 bites előosztó + 16 bites számláló), ami nagyon tág határokkal rendelkező időzítést tesz lehetővé. Fontos megjegyezni, hogy az előosztója különálló, és már nem osztzik a WDT egységgel, ezért az átkapcsolására már nincsen szükség.

16/24 A 16 bites mikrovezérlőknél már öt, közel hasonló képességű, 16 bites számláló modult használhatunk. Mindegyik számlálóhoz egy 16 bites periódusregiszter kapcsolódik egy digitális komparátoron keresztül, amely a számláló és a periódusregiszter tartalmának egyezésekor megszakítást generálhat, és a számláló tartalmát törli. Felépítése a 3.13. ábrán látható. Nézzük a működést:

- Ha PRx tartalma megegyezik TMRx tartalmával, akkor megszakítás generálódik (Tx Int) és a számláló törlődik (Rst).
- minden számlálónak van kapuzási lehetősége, amivel könnyen meg lehet határozni egy logikai magas szintű jel hosszát. Mikor a kapuzott számlálót engedélyezzük, és TxCKI órabemenet magas, akkor a rendszerórajel (TCS) lejteti a TMRx számlálót. Mikor TxCKI lemege nullára, akkor egy megszakítás generálódik, és az előosztóval leosztott órajel-impulzusszám a számlálóból kiolvasható, vagyis az impulzushossz meghatározható.

Az ötöből négy számláló: TMR2+3 és TMR4+5 összekapcsolható 32 bites számlálóvá.

A TMR1 számláló képes aszinkron számlálóként is működni. Ez azt jelenti, hogy a működése nincs szinkronizálva a belső órajelhez, nélküle működik. Ezért például akkor is tud impulzusokat számlálni, mikor az eszköz SLEEP állapotban van, és amikor a számláló értéke egyezik a periódusregiszterével, a megszakítás felébreszti SLEEP-ből a kontrollert. Mivel ehhez a számlálóhoz kapcsolható egy külső, kis frekvenciájú oszcillátor, ezért ennek órajelével működhett aszinkron módban.

3.4. NAPTÁR/ÓRA MODUL (RTCC)

16/24 Nagyon gyakori feladat az idő pontos követése, hiszen bizonyos események bekövetkeztének idejét adhatjuk meg, vagy éppen tárolhatunk bizonyos időpontokat. Erre szolgál a Real Time Clock and Calendar (RTCC), vagyis a valós idejű óra- és naptáramkör, amit a Microchip önálló perifériaként az újabb, 16 és 32 bites bites családjába épít be.

Működését egy oszcillátorfokozat biztosítja, ez adja a működéshez szükséges 1 Hz-es jelét. Az óra/naptár funkciót ezzel a jellet léptetett számlálólánc biztosítja, amely az aktuális másodperc/perc/óra/nap/hónap/év értékeit tartalmazza. A riasztási (alarm) időpont egy feltölthető regisztersorba kerül, és az aktuális időt tartalmazó számlálólánc megfelelő regisztereivel lesz összehasonlítva.

Az RTCC programozói szempontból nagyon kevés regisztert használ:

- **RTCC vezérlő regiszterek**
RCFGCAL: RTCC Kalibráló és Konfigurációs Regiszter
PADCAL1: Lábkivezetést konfiguráló Regiszter
ALCFGRPT: Riasztásokat beállító Regiszter
- **Aktuális dátum/időt tároló RTCC adatregiszterek:**
RTCVAL (ha RTCPTR<1:0> = 11); aktuális év
RTCVAL (ha RTCPTR<1:0> = 10); aktuális hónap és nap
RTCVAL (ha RTCPTR<1:0> = 01); az aktuális hétnapja és órája
RTCVAL (ha RTCPTR<1:0> = 00); az aktuális perc és másodperc

- Riasztási dátum/időt tároló regiszterek

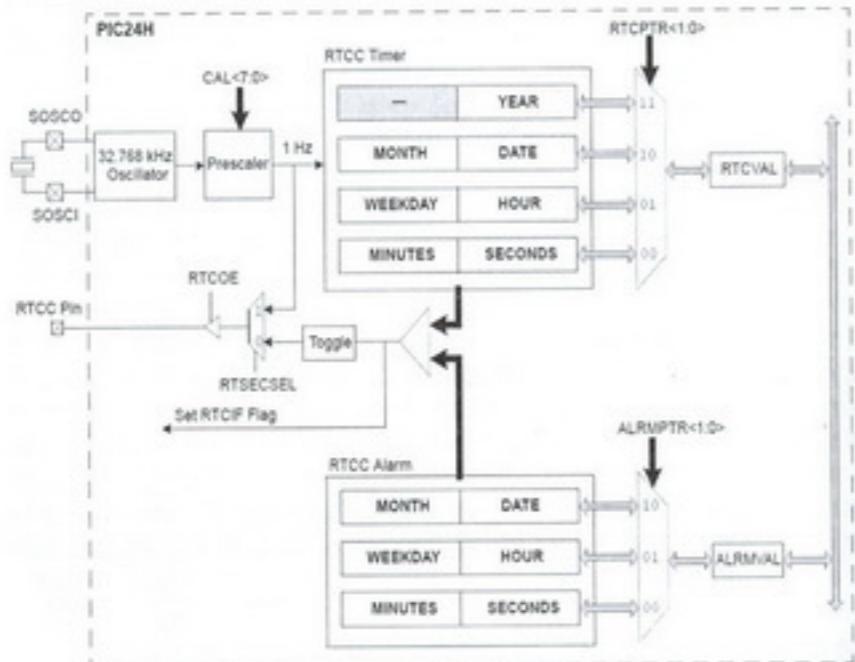
ALRMVAL (ha ALRMPTR<1:0> = 10): Riasztás hónapja és napja

ALRMVAL (ha ALRMPTR<1:0> = 01): Riasztás hete, napja és órája

ALRMVAL (ha ALRMPTR<1:0> = 00): Riasztás perce és másodperce

A 16 bites RCFGCAL regiszter felső 8 bitjeivel lehet a számlálólánc adott regisztereit megcímzni, abba beleírni, illetve a tartalmát kiolvasni. Egy bittel lehet a modul működését engedélyezni/tiltani, beállíthatjuk, hogy a felhasználó módosíthatja-e a számlánc tartalmát, az RTCC kimeneti láb engedélyezett-e. Ezen a lábon vagy az 1 mp-es időalap, vagy a riasztás okoz állapotváltozást (PADC₁ regiszter 1-es bitje). Az alsó 8 bit állításával tudjuk az óra pontosságát beállítani (kalibrálni).

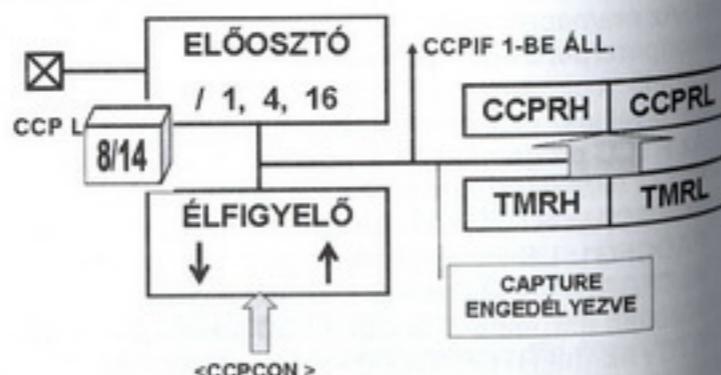
ALCFGRPT bitjei határozzák meg a riasztás időpontját és gyakoriságát.



3.14. ábra

3.5. A CCP MODULOK FEJLŐDÉSE

8/14 CAPTURE (esemény elfogása, elkapása): Egy lábon fellépő esemény (pl. szintváltás = felfutó vagy lefutó él) hatására egy számláló (TMR) 16 bites tartalmát a speciális CCPR capture regiszterbe írjuk. Egy előosztóval lehet csak minden 4., illetve 16. eseményre reagálnunk.



3.15. ábra
CAPTURE mód

COMPARE (összehasonlítás):

Egy számláló 16 bites tartalmát egy 16 bites regiszterben lévő tartalommal hasonlítjuk össze. Egyezés esetén egy adott kimeneti lábon eseményt, illetve megszakítást generálunk.

A 8 bites mikrovezérlőknél két közös Capture-ComPare (CCP) modult integráltak a tokba, mert a CAPTURE-COMPARE működés egymást kizárt.

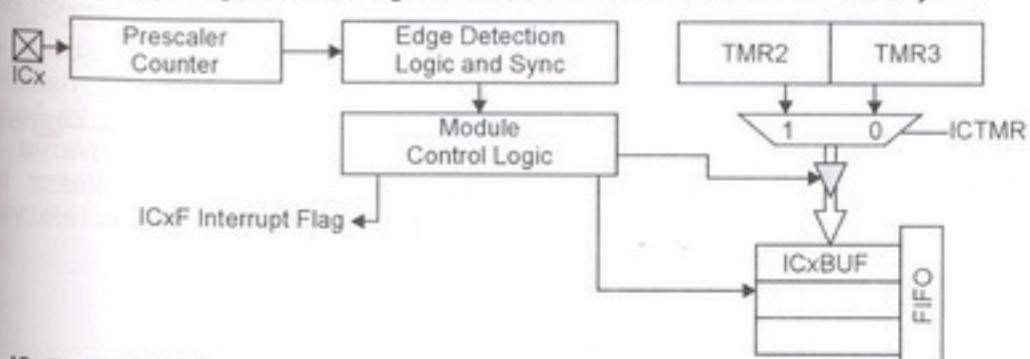
COMPARE működés egymást kizárt.
 (Ugyanazt az áramkörkészletet használta
 mind a **CAPTURE**, mind a **COMPARE** mód.) Csak a TMR1-es számláló lehetetlenné tette a CCP
 modulok időalapja, a PIC18F család megjelenésétől kezdve a TMR3 számlálót is lehet vá-
 lasztani. A CCP modulok még PWM jelek előállítására is konfigurálhatók.

A PIC18-as családban az egyik CCP modult úgy alakították ki, hogy alkalmas legyen motormeghajtó áramkörök vezérlésére is. Ezek az Enhanced (továbbfejlesztett) CCP modulok, rövidítve ECCP modulok.

A 16 bites mikrovezérlők megjelenésével a modulokat átalakították/különválasztották: megjelentek az önálló **Input Capture (IC)** modulok, illetve az önálló **Output Compare (OC)/PWM** modulok, amelyekből maximum nyolcat tartalmazhatnak a tokok. Azt, hogy egy adott típusban hány ilyen (max. 8) modul található, az adattlapok tartalmazzák. A továbbiakban a modulsorszámot az „x” jelöli.

3.6. AZ INPUT CAPTURE MODUL

16/24 32/32 A 16 bites PIC mikrovezérlőkben az áramkör már önálló modulként jelenik meg. A modul blokkválatán láthatók a legfontosabb részek: **ICx** – külvilág felőli lábkivezetés – előosztó (*prescaler*) az éldetektáló és az elváltást a belső órajelhez szinkronizáló egység – a kiválasztott TMR2 vagy TMR3 számláló tartalmát az esemény bekövetkezésekor a négy-szintű puffer következő tárolójába írjuk. A modul konfigurálására a 16 bites **ICxCON** regiszter szolgál. Az esemény megszakítást is generálhat, amit az **ICxIF** bit 1-be állása jelez.



ICxCON<IC|1:IC|0>

- 11: Interrupt on every fourth capture event
- 10: Interrupt on every third capture event
- 01: Interrupt on every second capture event
- 00: Interrupt on every capture event

ICxCON<ICM2-ICM0>

- 111: Input Capture functions just as an interrupt pin while in Sleep or Idle mode
- 101: Capture on every sixteenth rising edge
- 100: Capture on every forth rising edge
- 011: Capture on every rising edge
- 010: Capture on every falling edge
- 001: Capture on every edge (both rising and falling)
- 000: Capture module is turned off

3.17. ábra
Input Capture modul

Az adott lábon fellépő milyen események okoznak kiolvasást?

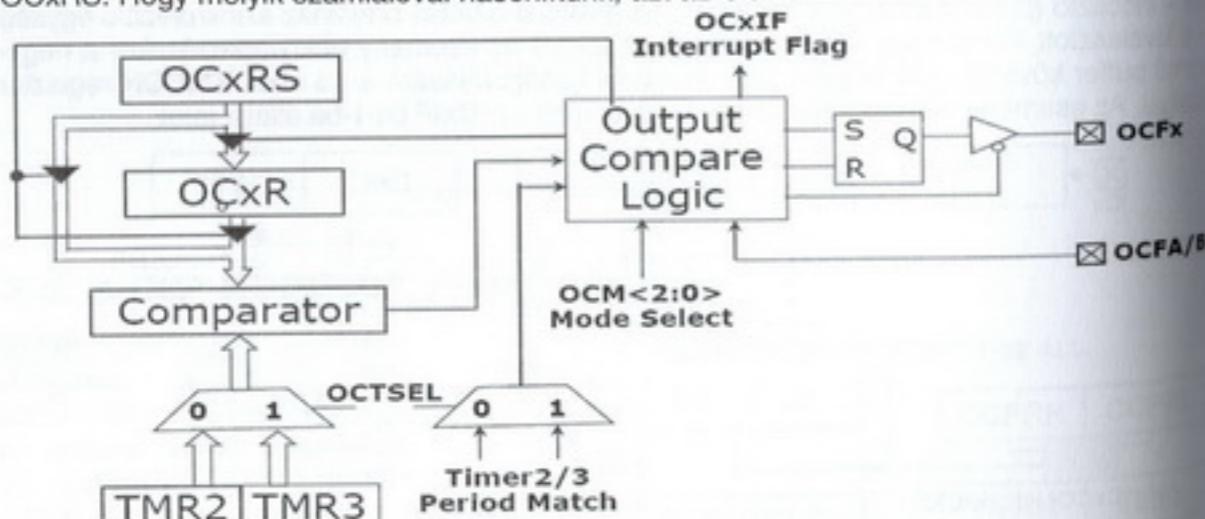
- 1) Egyszerű capture esemény
 - A számláló regiszter aktuális értékének rögzítése a capture puffer regiszter tárolójába az ICx lábon megjelenő jel minden lefutó élénél.
 - A számláló regiszter aktuális értékének rögzítése a capture puffer regiszter tárolójába az ICx lábon megjelenő jel minden felfutó élénél.
- 2) Kiolvasás capture regiszterbe minden (fel- és lefutó) élnél.
- 3) A bemeneti lábon lévő előosztón megjelenő események esetén történő capture regiszterbe mentés.
 - A számláló regiszter aktuális értékének rögzítése a capture puffer regiszter tárolójába az ICx lábon megjelenő jel minden negyedik felfutó élénél.
 - A számláló regiszter aktuális értékének rögzítése a capture puffer regiszter tárolójába az ICx lábon megjelenő jel minden tizenhatodik élénél.

Ha a számláló regiszter és a CCPR regiszter tartalma megegyezik, akkor – a konfigurálástól függően a következő tevékenységek valamelyike hajtódiák végre: CCPx láb törlése, CCPx láb 1-be állítása, megszakítás generálása (CCPx láb állapota nem változik), speciális esemény kiváltása (CCPx láb állapota nem változik): CCP1 törlí TMR1 számítlót, CCP2 törlí TMR1 számlálót és az A/D átalakító GO/DONE bitjét.

Az adatlapok ismertetik az egységek pontos jellemzőit, valamint a programozásukhoz szükséges részletes információkat.

3.7. OUTPUT COMPARE (OC) / PWM MODUL

16/24 32/32 Ez a modul a szokásos Compare működés mellett úgy lett kialakítva, hogy könnyen lehessen különböző típusú PWM jelek előállítására használni. Az egység, amiből maximum 8 lehet egy tokban (OC1, OC2,...OC8) a következő föbb részekből áll: Két 16 bites regiszterbe lehet az összehasonlítandó értéket tölteni: OCxR, illetve a másodlagos OCxRS. Hogy melyik számlálóval hasonlítunk, azt az OCTSEL értéke dönti el.



3.18. ábra
Output Compare modul vázlatá

Az OCx modul működésmódját az OCxCON regiszter <OCM2:OCM0> bitjei határozzák meg, a következő táblázat szerint:

OCxCON<OCM2:OCM0>

111: PWM mód, hibabittel

110: PWM mód hibabit nélkül

101: OCx kezdetben alacsony, folyamatos impulzust generál

100: OCx kezdetben alacsony, egy impulzust generál

011: Az OCx minden egyezéskor ellen-tettjére vált

010: OCx kezdetben magas szintű, egyezéskor alacsony szintű lesz

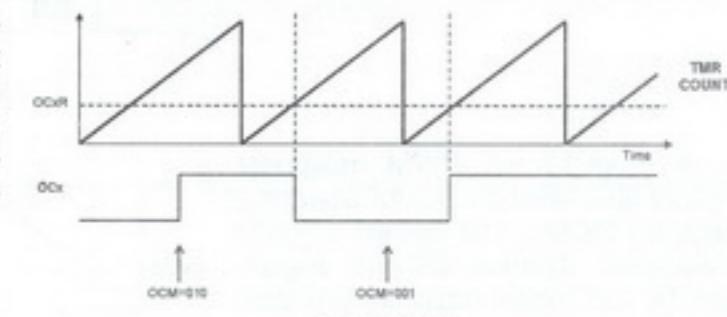
001: Az OCx kezdetben alacsony szintű, egyezéskor magas szintű lesz

000: A Compare modul kikapcsolva

A modullal lehetséges egyetlen kimeneti impulzus, illetve impulzussorozat előállítása. A lábon megjelenő eseménykor egy megszakítás is generálódhat, amit az OCxIF bit 1-es állapota jelez. A következő ábrákon összefoglaltuk az OC modul üzemmódjait.

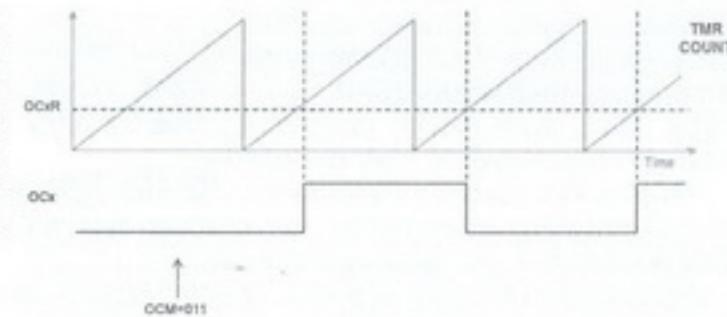
Az ábrákon a számlálóban lévő érték növekedését, majd túlcordulását a fűrészfog alakú jel mutatja. Az OCxR regiszterben lévő komparálandó értéket a vízszintes szaggatott vonal jelzi. Amikor ezt a vonalat metszi a fűrészfog, akkor a számláló és a komparálandó érték azonos lesz, bekövetkezik az egyezés.

Itt két eseményt mutatunk be, amit a beállítható OCM bitek határoznak meg. **OCM = 010** esetén az OCx láb alapállapota magas, és egyezéskor alacsony szintre vált, míg **OCM = 001** esetén éppen fordított: alaphelyzet alacsony, egyezéskor magas szintre váltás.



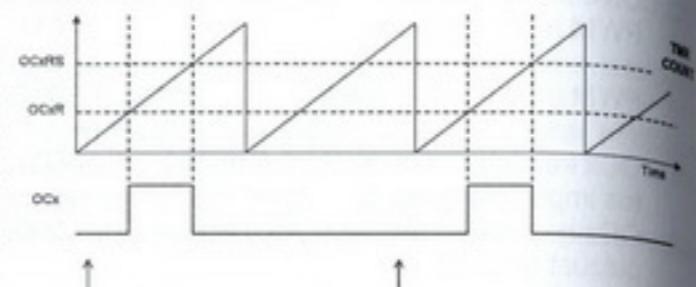
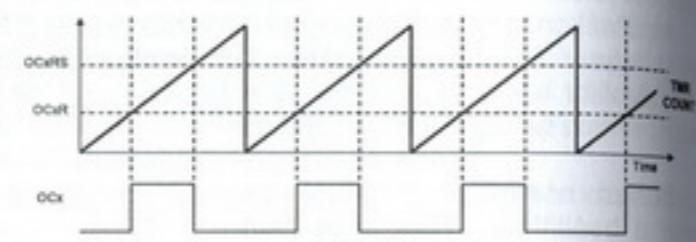
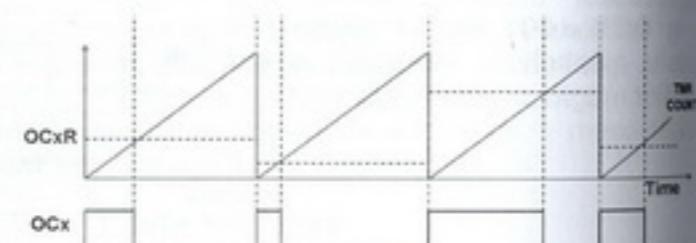
3.19. ábra
OC set/clear módok

OCM = 011 esetén minden egyezéskor az OCx láb ellenettjére vált (toggle mód). Fontos észrevenni, hogy ebben az esetben a lábon megjelenő jel periódusideje állandó, OCxR tartalmának változtatásával csak a számláló állapotához mért fázishelyzetét tudjuk változtatni.



3.20. ábra
OC váltásmód

A második, OCxRS regiszter felhasználásával kettős komparálást használunk. A két regiszter tartalmának a különbségét kell a számláló periódusidejével megszorozni, hogy megkapjuk az OCx lábon megjelenő egyetlen impulzus hosszát. Újabb impulzus az OCxCON regiszter újraírásával váltható ki. Kezdetben OCx láb alacsony szintű.

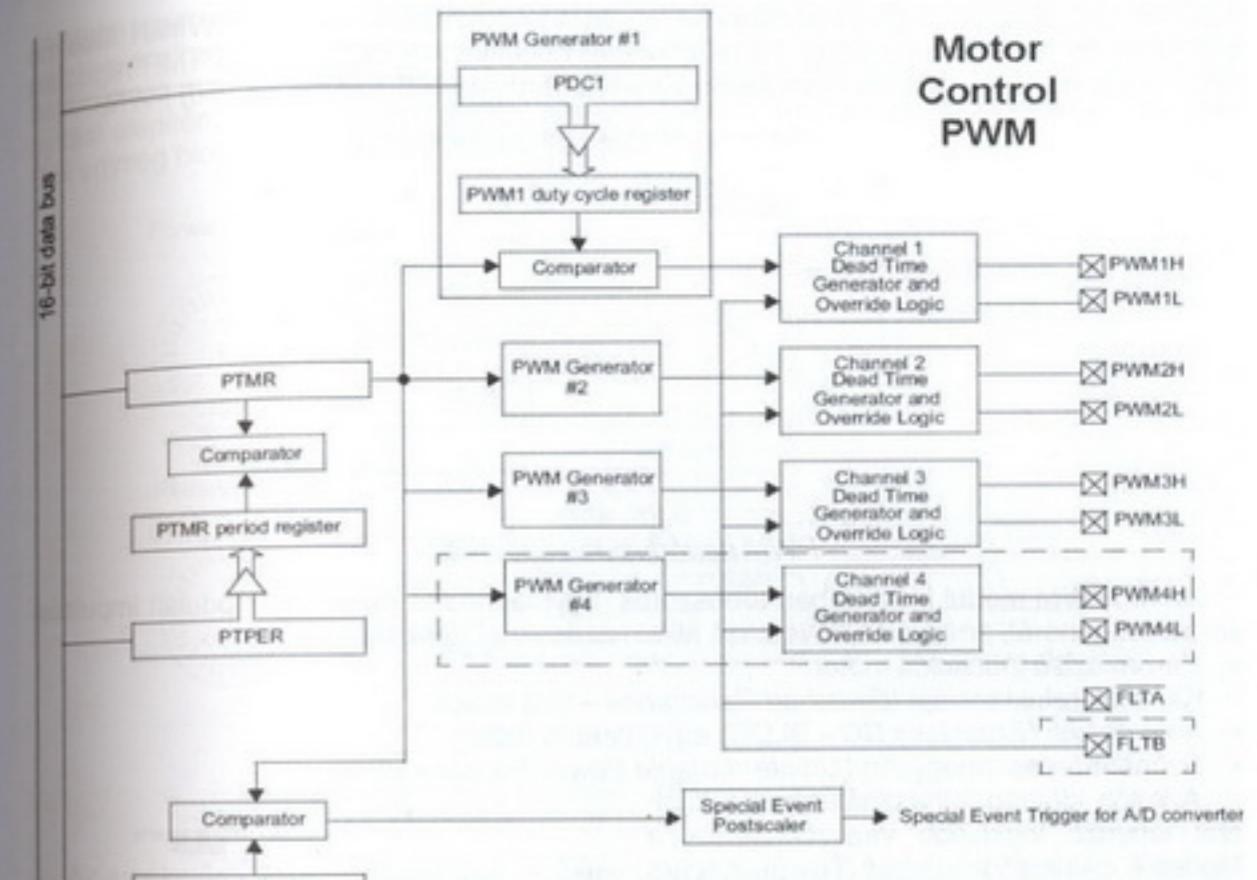
3.21. ábra
OC egy impulzusmód3.22. ábra
OC impulzussorozat-mód3.23. ábra
Egyszerű PWM mód

Az ábrán már folyamatos impulzussorozatot generálunk. A komparáló regiszterek feltöltése után OCM = 101 beállításával, majd OCxCON regiszter feltöltésével indítjuk el a folyamatot.

A modullal két PWM működési módot is előállíthatunk. Az ábra mutatja be OCM = 110 esetén a PWM működést. Ilyenkor OCxRS regisztert fel kell tölteni azzal az értékkel, ami a számláló periódusidejével szorozva megadja az OCx láb magas állapotban tartózkodásának idejét a perióduson belül. minden számláló túlcordulásakor OCxRS regiszter értéke a komparálandó OCxR regiszterbe íródik, ilyen módon megakadályozva az érték perióduson belüli megváltozását, ami hibás kitöltési tényezőt (duty cycle) adna.

OCM = 111 esetén a PWM működése már egy bemeneti hibabit állapotától is függ. A hibavédelem négy csatornához van közösen hozzárendelve: az OCFA bit az 1-4., míg az OCFB bit az 5-8. OC modulhoz tartozik.

Ha ezeken a lábakon a modul 0 szintet érzékel, akkor a hozzá tartozó PWM kimeneti bitet magas impedanciás állapotba helyezi. Ezért a felhasználó a fel- vagy lefutó ellenállással olyan kimenetjelszinet állít be a lábon, amilyet csak akar. Az állapot azonnal létrejön, és a PWM leáll. Ez a FAULT (hiba) állapot mindaddig fennmarad, amíg a külső FAULT feltétel fennáll, és amíg a PWM-et újra nem engedélyezzük OCM<2:0> újraírásával. A FAULT állapot az OCxIF megszakítást jelző bitet is 1-be állítja.

3.24. ábra
MCPWM blokkvázlat

Az OUTPUT COMPARE modullal kapcsolatos további ismeretek a használt tok adatlapján, továbbá a DVD-mellékleten megtalálhatók.

3.8. MOTOR CONTROL PWM (MCPWM)

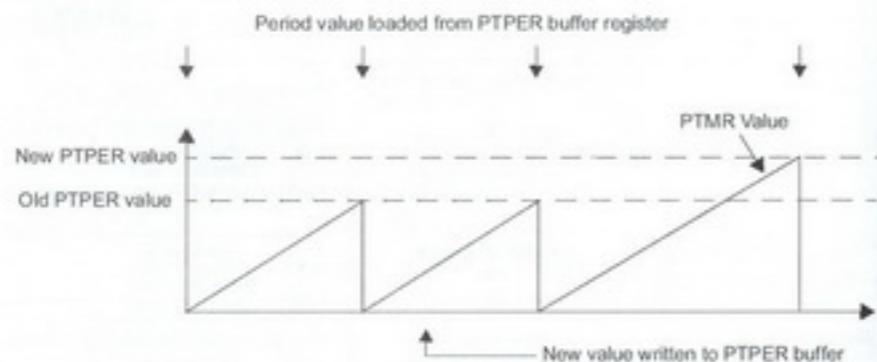
Mivel az elektronikus motorvezérlések szerepe folyamatosan nő, a Microchip a 16 bites mikrovezérlőihez ilyen célokra egy külön perifériát fejlesztett ki: ez az MCPWM (3.24. ábra). A modul három vagy négy PWM generátoregységet tartalmaz, ahol közösen megadhatjuk a PWM jelek periódusidejét.

16/24 32/32 Az MCPWM modul az órajelét a rendszerórajelből kapja, annak 1/4/16/64 leosztásával. Ez a PTCON PWM idő-alapvezérlő regiszter beállításával határozható meg. A beállított órajel lépteti a PTMR számlálót. A periódusidőt a PTPER regiszterbe írt értékkel állíthatjuk be (3.24. ábra). Az egység képes egyetlen PWM pulzus- vagy impulzussorozat előállítására.

A 3.25. ábrán látható, hogy egyetlen impulzus előállításakor a PWM számlálója kezd felé számolni, amikor a PTEN bitet egybe állítjuk. Mikor a számláló eléri a PTPER regiszterbe beállított értéket, a PTMR regiszter nullázódik, és a PTEN bit törlödik.

Hagyományos PWM esetén, a komparátor működése miatt a jel aktív szintje minden a számláló túlcordulásakor indul. Ez az érle igazított (edge aligned) üzemmód. Ez látható a

3.26. ábrán, ahol a PDCx regiszterekbe írt értékek határozzák meg a PWMxH lábakon megjelenő jel kitöltési tényezőjét. Lehet azonban másfajta megoldás is, ha PTMR regiszter fel-le számláló üzemmódban használjuk. Ez a középre igazított (*centre aligned*) PWM.



3.25. ábra
PWM periódusidő-változtatás

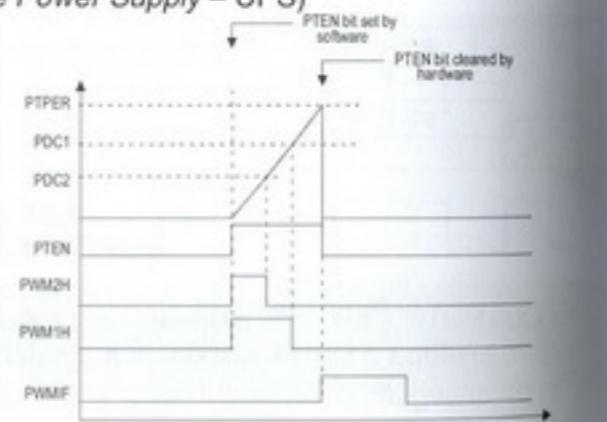
Az MCPWM modul lényegében többszörös, egymáshoz szinkronizált modulált impuluszsorozatot generál, amelyek a következő alkalmazásoknál használhatók:

- Háromfázisú induktív motor
- Kapcsolt reluktanciájú (*Switched Reluctance – SR*) motor
- Kefenélküli (*Brushless DC – BLDC*) egyenáramú motor
- Szünetmentes tápegység (*Uninterruptable Power Supply – UPS*)

A külön időalap felhasználásával a kitöltési tényező legkisebb változtatásának a lépése a ciklusidő fele lehet ($T_{CY}/2$). Két láb tartozik minden PWM generátor modulhoz, amelyek képesek egymás inverzeiként vagy egymástól függetlenül működni.

A vezérelt félvezető eszközök véges kapcsolási ideje miatt beépített holtidő-generáló áramkör működhet. minden PWM-kivezetés aktív jelszintje programozható. A kimenet lehet érte igazított, középre igazított, illetve egyes impulzus. A modulokhoz tartozó bemeneti (hiba) bitek állapotának figyelésével a PWM kimeneti bitjei letiltathatók.

Az időalap és egy komparátor segítségével AD átalakítás elvégzése is kiváltható.



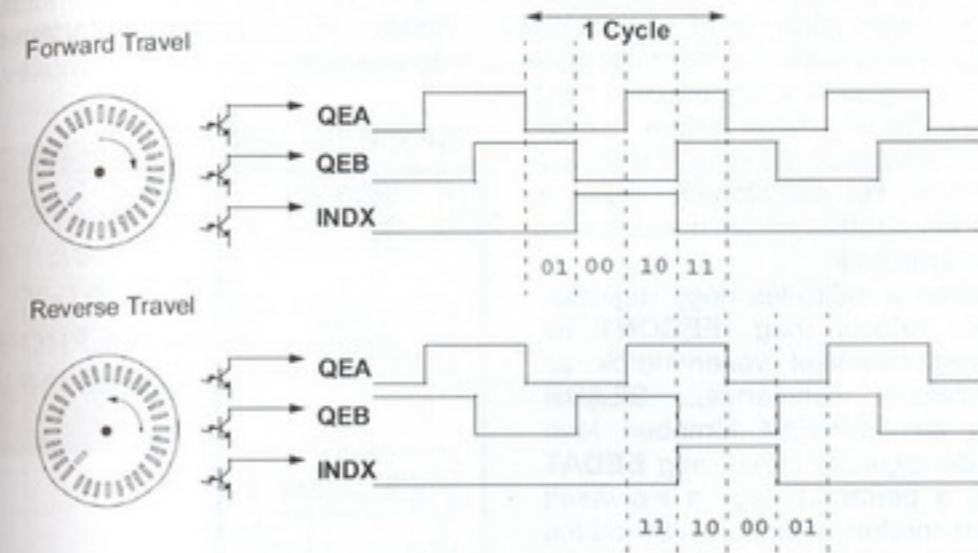
3.26. ábra
Egyetlen PWM impulzus előállítása

A MOTOR CONTROL PWM modullal kapcsolatos további ismeretek a használt tok adatlapján, továbbá a DVD-mellékleten megtalálhatók.

3.9. FORGÁSÉRZÉKELÉS – QEI

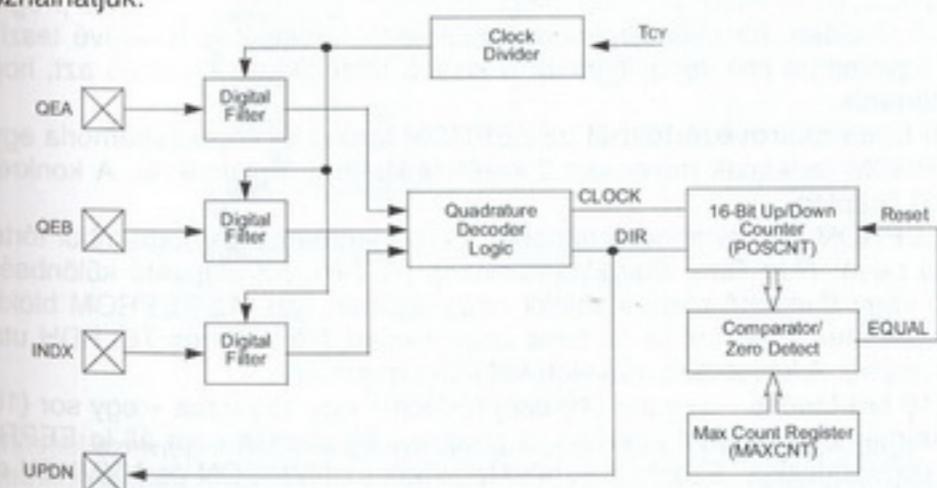
16/24 32/32 Motorvezérlésekben egy olyan bemeneti perifériára is szükség van, ami a forgás irányát, sebességét, helyzetét tudja érzékelő felhasználásával meghatározni. Ez az egység a *Quadrature Encoder Interface*, a továbbiakban QEI.

Maga az érzékelő a motor tengelyére szerelt tárcsa, amelyen résék vannak, és megfelelő optokapuk elhelyezésével állnak elő az érzékelő jelek. Három jel közül QEA és QEB jelekből határozható meg a forgás iránya, míg az INDX jel fordulatonként egy jelet ad, amit viszonyítási alapként vagy fordulatszámmal arányos jelként használhatunk. A jeleket feldolgozó QEI egység blokkvázlata a 3.28. ábrán látható.



3.27. ábra
QEI jelek

Az érzékelőről érkező jeleket egy digitális szűrőn keresztül fogadja az egység. Kimenetként a forgásirányt jelző bitet adja vissza. Az egység működéséhez csupán három 16 bites regisztert használ: a QEICON vezérlő regisztert, a pozíciót tartalmazó POSCNT regisztert, valamint a MAXCNT regisztert, aminek tartalmát a POSCNT regiszterrel való összehasonlításra használhatjuk.



3.28. ábra
QEI blokkvázlata

A QEI modullal kapcsolatos további ismeretek a használt tok adatlapján, továbbá a DVD-mellékleten megtalálhatók.

3.10. EEPROM MODUL

A belső EEPROM adatmemóriát a mikrovezérlő programmal írhatja és olvashatja, és kikapcsolt állapotban is megörzi a benne tárolt adatokat. Az EEPROM adatmemória a mikrovezérlő felprogramozásakor is feltölthető kezdeti tartalommal. A memória írási ciklusainak száma korlátozott, aminek végén elveszti a tárolási képességét. Értéke kb. százezer és egymillió írási ciklus körül van. Ez az érték a körülményektől is függ: az élettartam nő, ha a működési hőmérséklete, a használt tápfeszültség alacsony.

Fontos a megfelelő programozás használata, az EEPROM adatait tartsuk a RAM-ban, ott írjuk, olvassuk, és csak a kikapcsoláskor mentsük. Ha megoldható, akkor az adatok minden újraírás során másik memoriaterületre kerüljenek.

Alapesetben a működés négy regiszter segítségével valósul meg. **EECON1** és **EECON2** regiszterekkel vezérelhetők az írás-törlés-olvasás események. **EEADR** tartalmazza az EEPROM tömbben lévő kiválasztandó regiszter címét, míg **EEDAT** tartalmazza a beírandó vagy a kiolvasott adatot. Ilyen módon maximum 256 bájtos adat kezelhető. Ez könnyen bővíthető, ha bevezetjük az **EEADRH** és **EEDATH** regisztereket. Ekkor az <EEADRH:EEADR> már 16 bites címet tartalmazhat, és minden címen a <EEDATH:EEDAT> regiszterpár 16 bites tartalmát használhatjuk. Vagyis a maximális EEPROM blokk: 65536*16 bit!

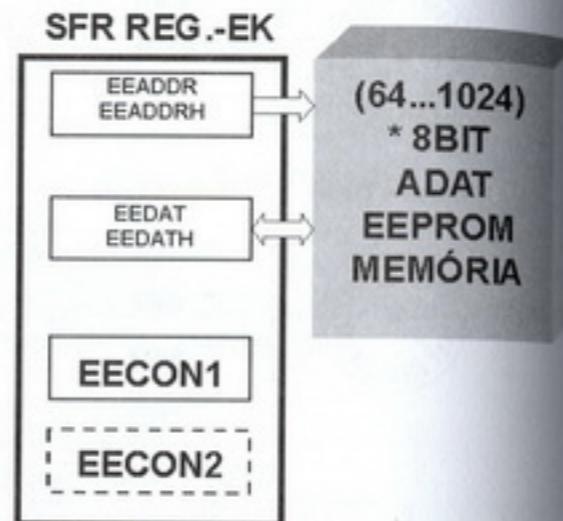
Mivel a flash memóriák használatával lehetővé vált a programmemória programból történő írása-olvasása, ezért alternatív lehetőségekkel lehetséges az adatoknak a programmemoriában történő tárolása. Ez akár nagyobb adatblokkok tárolását is lehetővé teszi (pl. adatgyűjtés), de figyelembe kell venni ilyenkor a kisebb írási ciklusszámot és azt, hogy az írási blokkokban történik.

16/24 A 16 bites mikrovezérlőknél az EEPROM terület a programmemória egy része. A 16 bites EEPROM tartalmak maximum 2 kiszót (4 kbájtot) foglalnak el. A konkrét méret a típusról függ (l. adattáblázat).

Az adat EEPROM programozása hasonló a programmemória programból történő írásához, ennek a neve: *Run-Time Self Programming* (RTSP). Az alapvető különbség az egy-szerre írható vagy törlhető memóriablokk nagyságában van. Az EEPROM blokk táblázási/olvasási művelettel érhető el (a 16 bites adatok miatt TBLWTH és TBLRDH utasításokat nem kell használni). A következő műveleteket lehet használni:

Egy szó (16 bit) törlése – egy sor (16 szó) törlése – egy szó írása – egy sor (16 szó) írása. A programmemória írásától eltérően, a programvégrehajtás nem áll le EEPROM programozáskor vagy törléskor. Ezekhez a műveletekhez az NVMCON és NVMKEY regisztereket használjuk. A programozó szoftvernek kell figyelnie a törlési vagy írási művelet befejezését, a következő három módszer valamelyikével:

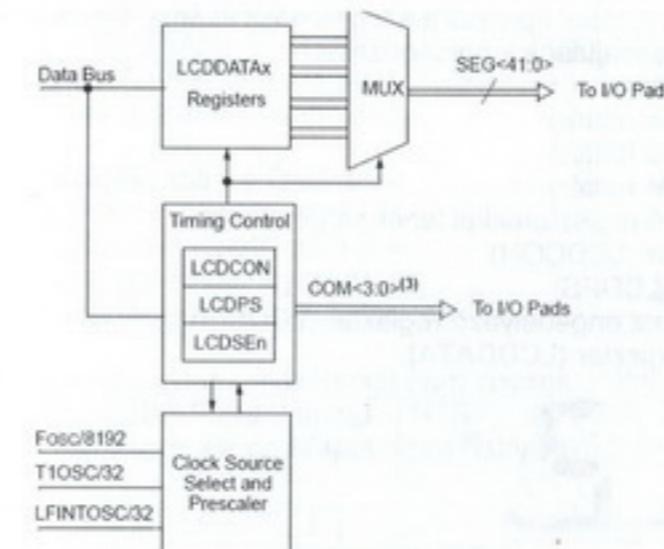
- WR bit (NVMCON<15>) figyelése. WR bit törlődik, ha a művelet kész.
- NVMIF bit (IFS0<12>) figyelése. NVMIF bit 1-be áll, ha a művelet kész.
- NVM megszakítás engedélyezése, és az esemény lekezelése.



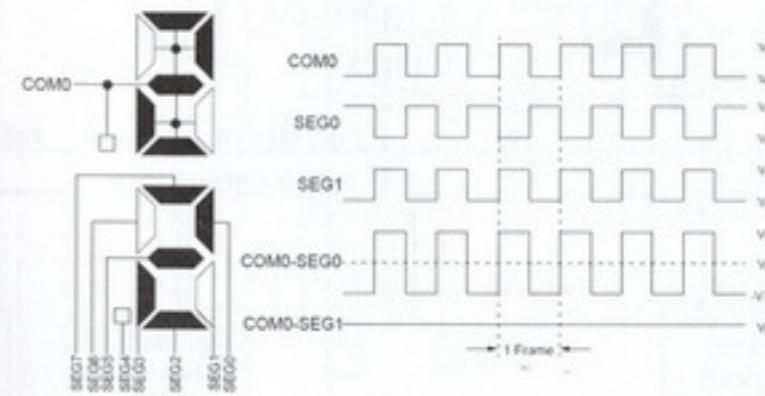
3.29. ábra
EEPROM-kezelés

3.11. LCD VEZÉRLÉS

8/14 8/16 Az LCD a *Liquid Crystal Display* – folyadékkristályos kijelző – kifejezés rövidítése. Működésének az a lényege, hogy a kis szálaknak tekinthető molekulacsoporthoz rendeltetlen állapotban nem verik vissza a fényt, míg rendezve, párhuzamosan elrendeződve visszaverik azt. Ezt a rendezettséget egy elektromos erőtér tudja létrehozni. A szerves anyagból álló folyadékkristály anyagot két üveglap között helyezik el, és az üveglapokra nagyon vékony, ezért átlátszó fémréteget visznek fel.



3.30. ábra
LCD-kezelés blokkvázlata



3.31. ábra
Statikus vezérlés

Az egymással szemben lévő üvegfelületetekben alakítják ki a szegmenseket. A vezérléshez szimmetrikus jeleket használnak, mert fontos, hogy a jelek középpontjában nulla legyen, az esetleges egyenfeszültség károsítja a kijelzőt.

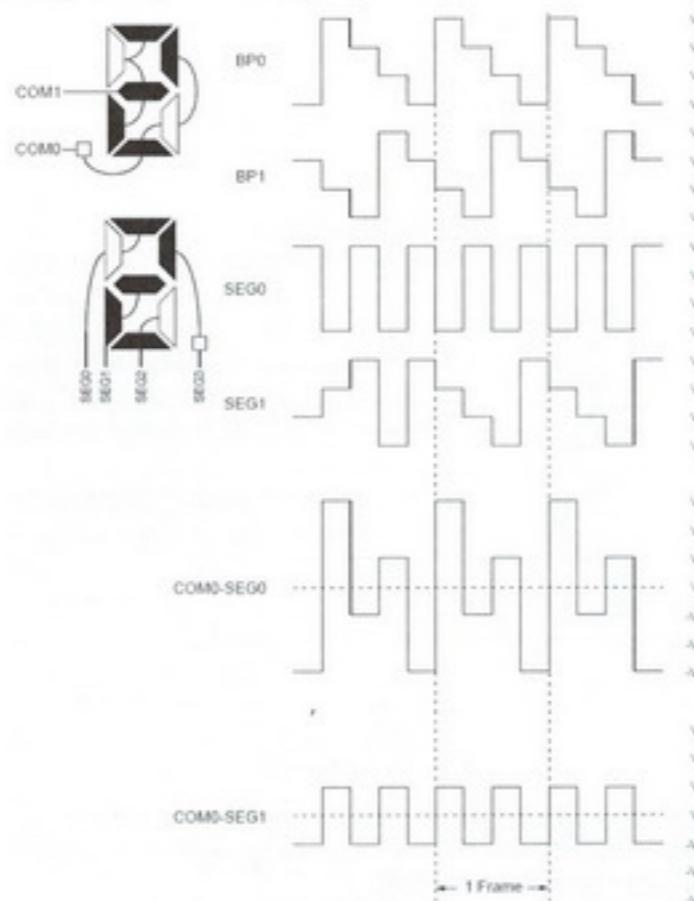
A statikus vezérlési módszernél minden egyes szegmensvonal egy szegmenst vezérel, ahogy a 3.31. ábrán látható. A példán a kijelző 2-t jelzi ki, a szegmens és a közös sík beiktatási módja is látható. A kiválasztott szegmensekre jutó, COM0-SEGn kivezetések közé eső feszültség értéke +Vdd és -Vdd maximális értékű negyszögfeszültség, tehát a szegmens visszaveri a fényt. A legalsó sorban (COM0-SEG1) a ki nem választott szegmens feszültsége 0 V.

A kétszeres időmultiplexelt, $\frac{1}{2}$ MUX vezérlésnél minden egyes szegmensmeghajtó vonal két szegmenst vezérel. A kiválasztott szegmens ilyenkor a vezérlési periódusidő felében kapja meg a teljes feszültséget, tehát a szegmens a periódusnak ebben a felében lesz látható. Az időtartam második felében a szegmensre fél feszültség jut, ami nem képes a kristályt láthatóvá polarizálni.

A Microchip PIC vezérlői közül a PIC16F913/914/916/917/946 típusokban van LCD vezérlő periféria, amely képes statikus és multiplexelt vezérlőjeleket adni az LCD kijelzőknek. Például a PIC16F913/916 típusokban max. négy közös háttérhez (COM0-COM3), legfeljebb 16 szegmens tartozhat. A többi típusnál a szegmensek száma: 24, illetve 42.

A következő LCD meghajtások lehetségesek:

- statikus (1 közös háttér)
 - 1/2 multiplex (2 közös háttér)
 - 1/3 multiplex (3 közös háttér)
 - 1/4 multiplex (4 közös háttér)
- A modult a következő regiszterekkel lehet programozni:
- LCD vezérlő regiszter (LCDCON)
 - LCD fázis regiszter (LCDPS)
 - Max. 6 LCD szegmens engedélyező regiszter (LCDSEN)
 - Max. 24 LCD adatregiszter (LCDDATA).



3.32. ábra
Időmultiplex-vezérlés

A legegyszerűbb olyan LCD panelek használata, amelyeknek egy közös háttérszegmensek van, mert ezt statikusan lehet vezérelni.

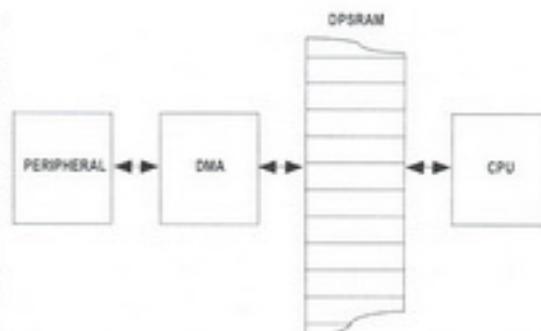
3.12. DMA – KÖZVETLEN MEMÓRIA-HOZZÁFÉRÉS

16/24 A DMA (Direct Memory Access) megoldás már régóta ismert. A lényege az, hogy a CPU által végrehajtott programmal történő átvitelvezérlés helyett a memória és a periféria között közvetlen átvitel zajlik. A Microchip 16 bites mikrovezérlöinél jelent meg ez a megoldás, növelve a rendszer teljesítményét.

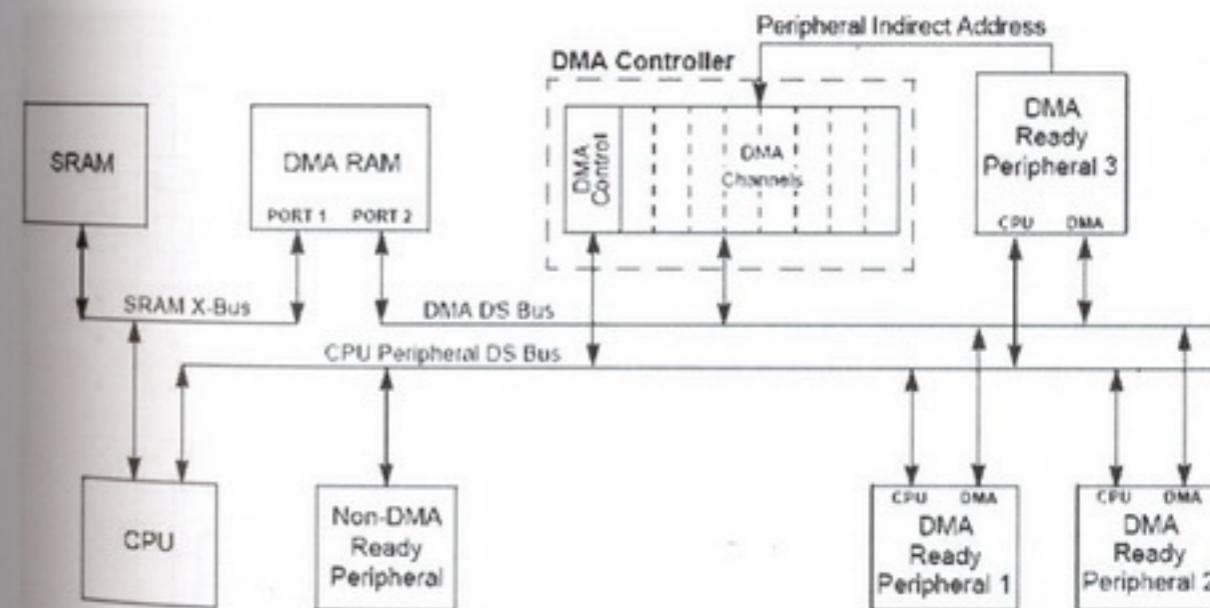
A DMA vezérlő mozgatja az adatokat a perifériák adatregisztereitől és a RAM között. A PIC DMA modul két irányból elérhető RAM-ot (*dual-ported SRAM memory – DPSRAM*) és regisztereit használ, amivel a DMA saját cím- és adatbuszán a CPU-tól teljesen független adatátvitelt valósít meg.

A hagyományos megoldásokban a DMA és a CPU osztottak a memóriához vezető cím- és adatbuszon, és DMA „ciklusokat lopott” amikor a CPU helyett használta a buszokat. Ilyenkor a CPU-nak fel kellett függesztenie a saját buszforgalmát.

A CPU és a DMA vezérlő egymás működését nem zavarja. Például ha a CPU programot ír a memoriába (*Run-Time Self-Programming – RTSP*), az RTSP művelet végéig a CPU egyetlen utasítást sem hajt végre. Ez az állapot nincs hatással a DMA adatforgalmára.



3.33. ábra
DMA átvitel



3.34. ábra
PIC 16/24 DMA
(A CPU és a DMA címbuszok nem szerepelnek az ábrán)

Fontos jellemzői:

- Hatékony adatátvitel a periféria SFR-ek (UART vételi reg. ad. stb.) és a DMA RAM között.
- Nyolc függetlenül kezelhető DMA csatorna használható: UART, SPI, DCI, Input Capture, Output Compare, ECAN, ADC perifériák esetén.
- Oda-vissza másolás az SFR regiszterek és a DMA RAM között.

- Minden DMA csatornánál:
 - szó vagy bájt adatátvitel: periféria → DMA RAM vagy DMA RAM → periféria második szint
 - DMA RAM terület közvetett címzése Auto Post Incrementtel (ha kell)
 - periféria közvetett címzése: néhány perifériánál a DMA RAM írási/olvasási címeit a periféria adhatja
 - egylépéses blokkátvitel: DMA leáll egy blokkátvitel után
 - folyamatos blokkátvitel: DMA RAM puffer kezdőcímének újratöltése minden befejezet blokkátvitel után
 - pingpong mód: két puffer felváltva töltögetése
 - blokkátvitel auto/kézi inicializálása
 - minden csatorna 32 lehetséges adatforrás és cél közül választhat.

A DMA vezérlő nyolc független csatorna működését támogatja. A csatornák képesek a kiválasztott perifériával történő adatcserére. A DMA-val támogatott perifériák: CAN, 10 és 12 bites AD, SPI, UART, Input Capture, Output Compare.

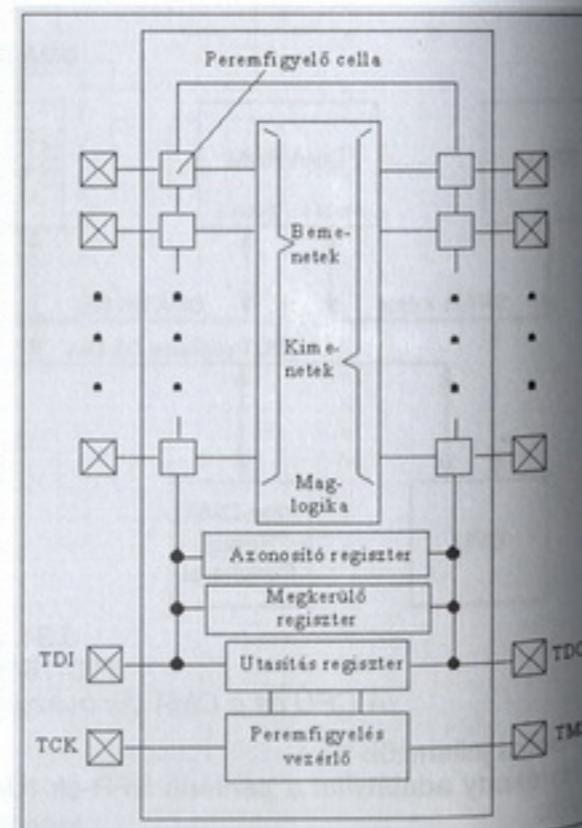
A DMA átvitelt az időzítők, illetve külső megszakítások indíthatják. A DMA csatornák egyirányúak, azaz írás és olvasás esetén a perifériához két DMA csatornát kell rendelni.

Ha több csatorna egy időben akar adatokat átvinni, akkor ez a csatornasorszámon alapuló fix prioritási séma alapján történik. minden csatorna maximum 1024 adatból álló blokkot képes átvinni, ezután egy megszakítást generál, jelezve a CPU-nak, hogy a blokk feldolgozható.

3.13. JTAG PEREMFIGYELŐ MODUL

16/24 32/32 Ahogy növekszik egy nyomtatott áramköri lapon elhelyezett alkatrészek száma, annál bonyolultabb és nehezebb a köztük lévő összeköttetéseket ellenőrizni. Az időigény mellett a csatlakoztatás is problémás: egyszerre esetleg több száz igen kis méretű pontra kell ideiglenesen, jó vezetést biztosító módon rákapcsolódni. Ilyenkor egy rugalmasan kialakított tűagyra fektetik az áramköri panelt, a belső pontokra vizsgáló jeleket adnak, és mérik a kimenetek viselkedését.

Egyesübb csatlakozást használó módszert dolgozott ki a Joint Test Action Group (JTAG) szervezet, aminek a neve: **boundary scan testing – peremfigyeléses vizsgálat**, amit később szabványosítottak: IEEE 1149.1-2001, „IEEE Standard Test Access Port and Boundary Scan Architecture”. Azóta számos mikrovezérlő-gyártó, köztük a Microchip is bevezette a 16 bites adatszélességű PIC mikrovezérlőknél.



3.35. ábra
Peremfigyelésre felkészített IC

„Minden chip-csatlakozópont és külső láb közé egy peremfigyelő cella kerül, ezek a célok egy léptető regisztert alkotnak. Az így létrejött léptető regiszter bemeneti pontja a TDI (Test Data In), kimeneti pontja a TDO (Test Data Out). Ezekben kívül egy órajel (TCK, Test Clock) és egy üzemmód-vezérlő pont (TMS, Test Mode Select) jelenik meg. A peremfigyelő cellákba tetszőleges tartalmat lehet beléptetni, a cella összeköthető az IC lábbal vagy a maglogikával – vagy egyszerre mindenkelővel (ez a teszten kívüli, normál állapot).

Az IC már gyártás közben is tesztelhető – csak négy ponton kell csatlakozni a chiphez –, ellenőrizhető a tokozás után és a felhasználás helyén is. A bemenetekhez tetszőleges vizsgáló jeleket lehet beléptetni, a kimeneti értéket a kimeneteknél lévő cellák befogadják, s azok onnan kiléptethetők. A felhasználói környezetben a peremfigyelésre alkalmas IC-k sorba köthetők (kaszkádosíthatók), s így a teljes panelnek is csak négy csatlakozópontja lesz a teszteléshez! A Boundary Scan alkalmas a késztermék tesztelésére is az élesztés során, használatbavételkor, sőt akár üzem közben is. A peremfigyelésre automatikus hibakereső megoldások is építhetők. Azt is lehetővé teszi, hogy pl. soros adatkezelésű EEPROM-okba a bekapsolás után töltünk tartalmat – ha az EEPROM pl. peremfigyelésre alkalmas BUSZ meghajtóra csatlakozik. Használják ezt a megoldást PLD (programozható logikai áramkör) üzem közben történő felprogramozására is.

Bár ez a módszer elég bonyolult, de hasznos, és ma gyakorlatilag minden mikroprocesszor, digitális jelprocesszor tartalmazza a peremfigyelő részleteket.” [Dr. Madarász László: A soros adatkezelés előretörése a digitális elektronikában. GAMF közleményei, Kecskemét, XIII. évf., 1996-97.]

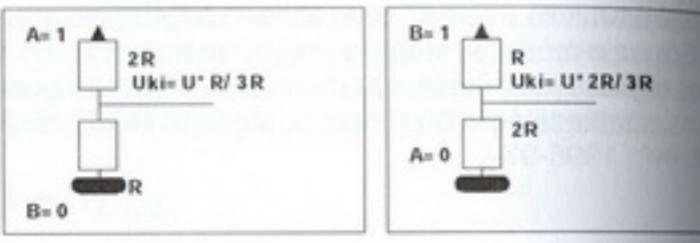
4. ANALÓG PERIFÉRIÁK

A mikrovezérlőkkel nagyon sokszor kell analóg jeleket feldolgozni. Erre az analóg-digitál (A/D) átalakítók szolgálnak. Ha digitális jeleket kell analóggá alakítani, akkor digitál-analóg (D/A) átalakítót használunk. Többfajta megoldás lehetséges, ezért a következőkben a Microchip megvalósításait tárgyaljuk. Az A/D és D/A átalakítók mellett ide tartoznak még az analóg komparátorok, illetve az ezekből felépített funkcionális egységek. Kétségtelen, hogy a legnagyobb jelentősége az analóg-digitál átalakítónak van, mert ezek teszik a külvilág felől érkező számos analóg jelet (feszültség, nyomás, hőmérséklet stb.) a digitális környezetben feldolgozhatóvá.

4.1. A D/A ÁTALAKÍTÁS ELVE: 4 BITES D/A ÁTALAKÍTÓ

Első lépésként készítsünk egy olyan áramkört, amelynek kimeneti feszültsége arányos a rákapcsolt bináris bitcsoport számértékével! A táblázat:

| BA | Uki |
|----|------|
| 00 | 0 |
| 01 | U/3 |
| 10 | 2U/3 |
| 11 | U |



Ez megvalósítható a fenti ábra szerint kapcsolt két ellenállással. Mivel az 1 szintnek az U feszültség felel meg, és a 0 a föld, ezért láthatók a helyettesítő vázlatok a BA = 01 és a BA = 10 esetre.

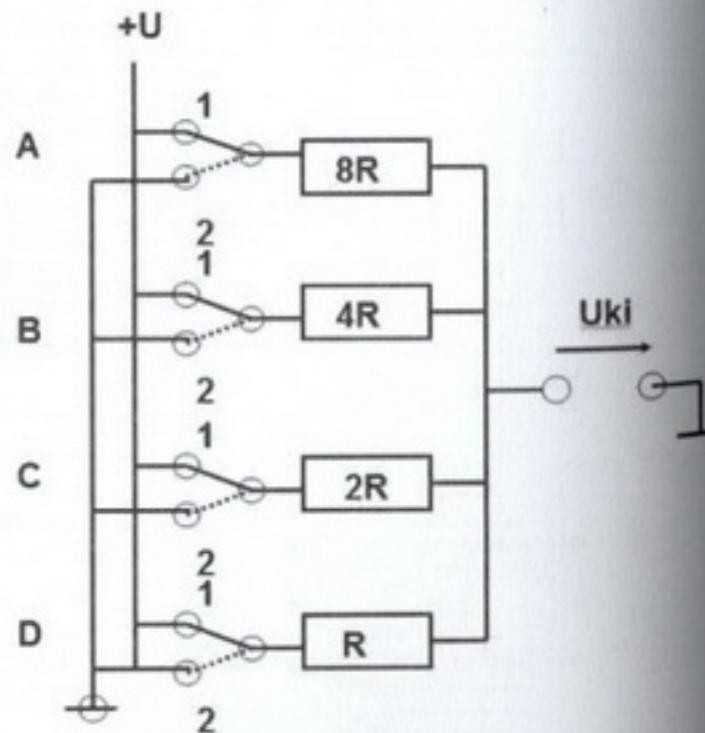
A 4 bites átalakító hasonló módon épül fel, de a 4 bit miatt négy ellenállásból álló létrahálózatot hozunk létre.

Ha minden kapcsoló „2” (GND) állásban van, a kimenő feszültség:

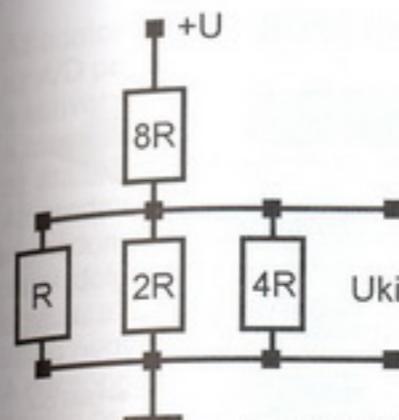
$$U_{ki} = 0$$

Ha minden kapcsoló „1” (+U) állásban van, a kimenő feszültség:

$$U_{ki} = +U$$



4.1. ábra
4 bites D/A átalakító



Ha az „A” kapcsoló „1” állásban, a többi pedig „2” állásban van, a kimenő feszültség:

$$U_{ki} = U \frac{Rx2Rx4R}{(Rx2Rx4R) + 8R} = U \frac{\frac{4}{7}R}{\frac{60}{7}R} = \frac{1}{15}U$$

mert $Rx2Rx4R = (2/3) \times 4R = 4/7R$.

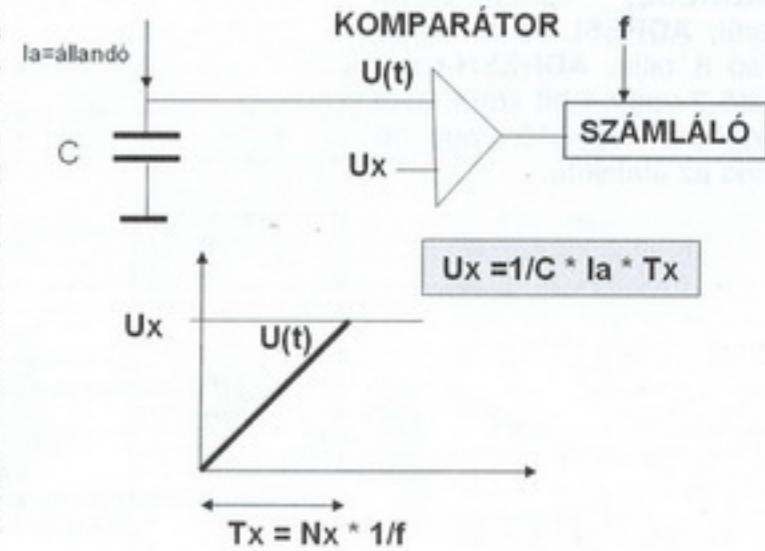
Az x operátor a „replusz” művelet jele.

Ilyen módon alakítható át egy bitcsoport analóg feszültséggé. Nyilvánvaló, hogy minél több bitet használ az átalakító, annál finomabb lesz a felbontás.

4.2. MEREDEKSÉG (SLOPE) A/D ÁTALAKÍTÓK

A **slope** módszernél egy kondenzátort töltünk egy állandó árammal, mindaddig, amíg a töltendő kondenzátor kezdeti nulla feszültsége el nem éri a mérendő feszültséget (U_x). Ezt a komparátor kimenetének átváltása jelzi. Ez az idő – ha áramgenerátoros (állandó áramú) a töltés – arányos lesz a mérendő feszültséggel.

Az idő számszerű értékét úgy kapjuk meg, hogy az átalakítás kezdetén nullázott számlálót léptetjük egy „f” frekvenciájú órajellel. A Microchip PIC14000 családjában van ilyen konverter, 16 multiplexelt analóg csatornát tud kezelni, a felbontása választhatóan 10-12-14-16 bit, és ennek megfelelően a konverziós idő is 0,25 ms és 16,4 ms között változik [AN621, AN624, AN626]. A tok az analóg komparátort, az áramgenerátort és a számlálót tartalmazza, csupán egy külső kondenzátort kell alkalmaznunk, és egy rövid program segítségével történik az átalakítás.



4.2. ábra
Slope A/D

4.3. FOKOZATOS KÖZELÍTÉSŰ A/D ÁTALAKÍTÓK

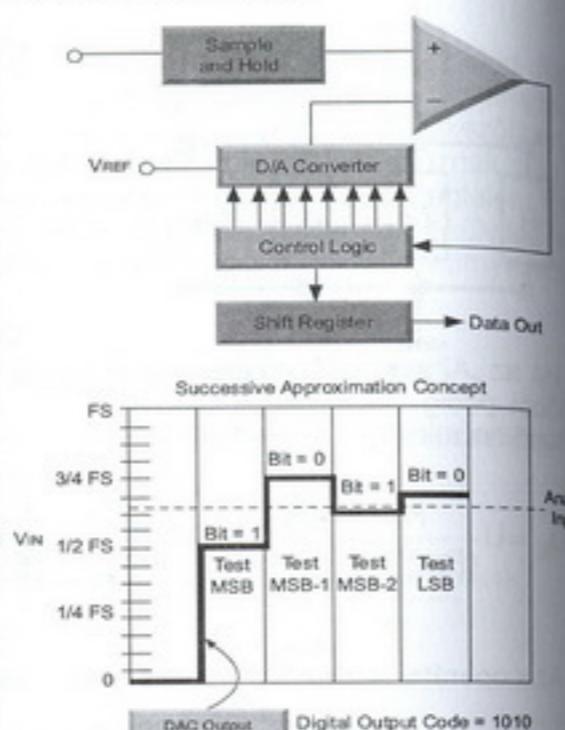
A szukcesszív approximációs (fokozatos közelítésű) konverter (átalakító) a feszültség meghatározásához a jól ismert, legoptimálisabb felezéses módszert használja. Első lépében azt határozzuk meg, hogy a mérődő feszültség a referenciafeszültség felénél kisebb vagy nagyobb. Ha az utóbbi, akkor a mérődő feszültség legmagasabb helyi értékű bitje 1, ha kisebb, akkor 0. A következő lépésként a megmaradó tartományt ismét felezzük, és hasonló döntéseket hozunk. Így például a 8 bites eredmény nyolc felezés-összehasonlítás-bitállítás lépéssel kapható meg. Az összehasonlításban szereplő állítható feszültséget egy D/A átalakító adja. Fontos megjegyezni, hogy egy-egy ilyen összehasonlítás időtartama 1-2 μ s, és ezt jelöljük T_{AD} -vel.

Sok feladat igényel több analóg csatornát, ezért a Microchip megoldásának a 4.2. ábrán látható áramkör az alapja. Egy fokozatos közelítésű átalakító áramkört, egy maximum 16 csatornás analóg multiplexert tartalmaz a periféria. A referenciafeszültség is választható: minden bemenete lehet belső (VDD-VSS) vagy külső. Az A/D bemenetén található egy mintavező tartó egység – Sample and Hold – röviden S/H. Ez egy kapacitás, ami biztosítja, hogy a konverzió alatt már ne változzon a mérődő feszültség. A felbontás 8-10-12 bites.

A mérés során az eredmény az [ADRESH] :[ADRESL] regiszter-párba kerül, ADRESL-be a mérés alsó 8 bitje, ADRESH-be a felső 2 vagy 4 bit kerül, attól függően, hogy 10 vagy 12 bites az átalakító.

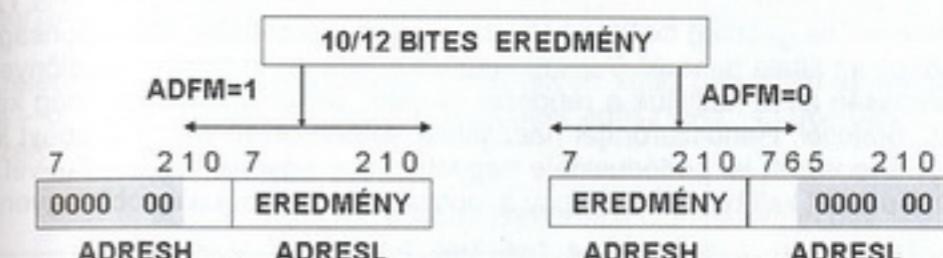


4.4. ábra
A/D átalakító



4.3. ábra
A fokozatos közelítés elve

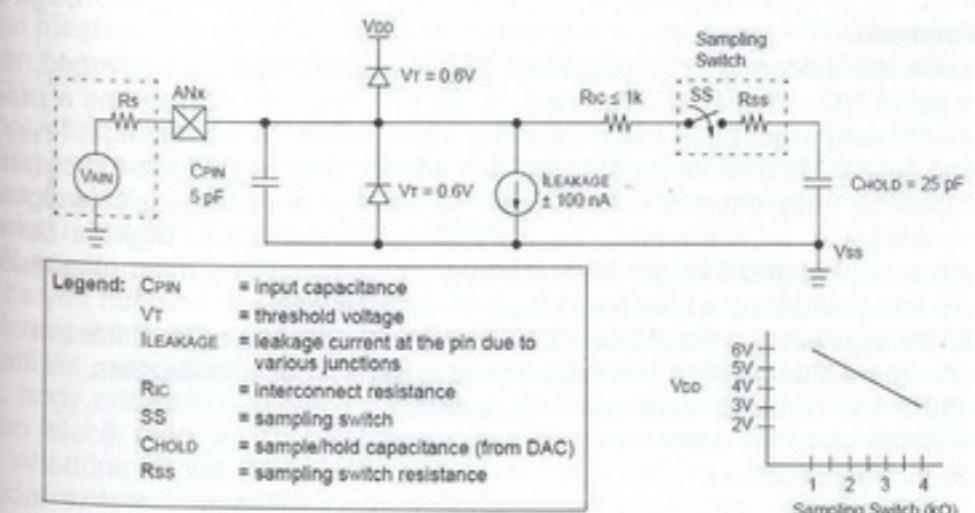
- Az adatokat/vezérlőjeleket az ADCONx ($x = 1, 2, 3$) regiszterek tartalmazzák:
- az A/D perifériát bekapcsoló bit: ADON;
- a konverziót indító/befejezést jelző bit: GO/DONE;
- a mérendő csatornát kiválasztó bitcsoport: CHS3:CHS0;
- az átalakító T_{AD} idejét meghatározó órajelet beállító bitek: ADCS2:ADCS0;
- Az eredmény formátumát meghatározó bit: ADFM. Azért vezették be ezt a tervezők, mert ilyen módon a 8 bites adatábrázolással összhangban az átalakítás 8 bites eredményével azonnal dolgozhatunk. Természetesen ha a 10 bites átalakítás esetén az alsó két bitjét elhagyjuk, akkor a pontosság romlik, illetve ha a felső két bitet, akkor csak a referenciafeszültség negyede lehet a mérődő feszültség.
- A bitcsoport meghatározza, hogy melyik láb legyen
- analóg bemenet: PCFG3:PCFG0,
- illetve külső referenciabemenet: VCFG1:VCFG0.



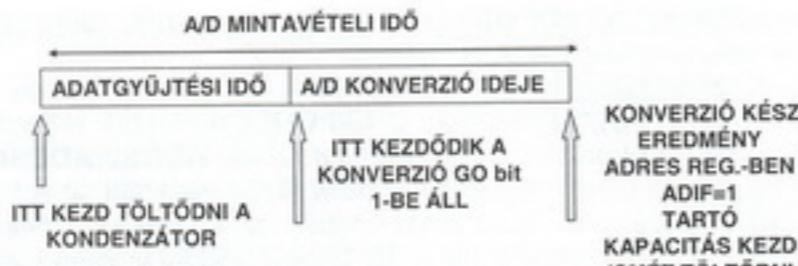
4.5. ábra
Az eredmény formátuma

A 4.6. ábrán látható az analóg bemenetek áramkori modellje. A modell legfontosabb eleme a C_{HOLD} kapacitás (a mintavező-tartó [sample and hold] áramkör), mert ennek V_{AIN} feszültségre való teljes feltöltődése esetén lesz pontos a mérés. A töltődés sebességét a bemeneti forrásellenállás R_S , illetve a belső R_{IC} és a kapcsoló R_{SS} ellenállások összege korlátozza. Az R_{SS} értéke a tápfeszültségtől is függ (l. 4.6. ábra). A bemeneti láb kis értékű árama az R_S ellenálláson átfolyva V_{AIN} értékét módosítja (offset).

Ezért az analóg jelforrás maximális értéke 2,5 k Ω lehet. Ha ennél kisebb, akkor C_{HOLD} gyorsabban töltődik, vagyis lerövidül a mintavétel ideje. A folyamat a 4.7. ábrán, részletei a 4.8. ábrán láthatók.



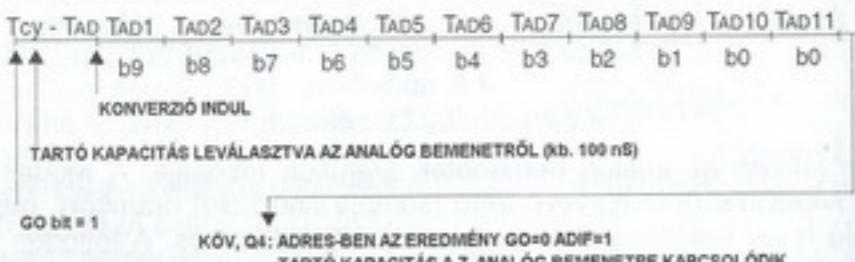
4.6. ábra
Analóg bemenet áramkori modellje



4.7. ábra

A/D konverzió

Az A/D átalakító az órajelét két forrásból kaphatja: vagy egy belső RC oszcillátor adja a T_{AD} -t meghatározó órajelet, vagy a rendszer órajeléből, annak leosztásával állítjuk elő. Fontos, hogy ennek összhangban kell lennie a katalógusban szereplő minimális T_{AD} értékkel. Természetesen ha gyárilag beállított RC oszcillátort használunk, azt biztonságosan úgy állították be, hogy az általa generált T_{AD} idő megfelelő lesz, és még az is az előnye, hogy ha energiatakarékosság miatt leállítjuk a rendszer órajelét, akkor a konverter még képes konvertálni az RC órajellel. Rendszerórajel használata esetén olyan osztásviszonyt kell kiszámolnunk, hogy a leosztott jel periódusideje nagyobb vagy egyenlő legyen T_{AD} -vel. Mivel ez az érték kb. 1,6 μ s, azt kell biztosítani, hogy a leosztott érték ennél nagyobb legyen.



4.8. ábra

Konverzió az időben

Például 20 MHz-es rendszerórajel esetén, ahol a periódusidő 50 ns, minimum a 32-es osztásviszonyt kell választani, mert $16 \cdot 50 \text{ ns} = 1,6 \mu\text{s}$. Mivel az RC oszcillátor periódusideje 4 μ s, jól látható, hogy ilyen megoldással az átalakítás sebessége és gyakorisága 250 százalékkal növelhető!

A konverziós idő csatornánként 16-20 μ s. Sleep alatt is folyhat a konverzió, ha az A/D órajelének a belső RC oszcillátort választjuk, és az A/D kész jele ébresztheti a processzort. A nagyobb pontosság érdekében használhatunk külső referenciafeszültséget (V_{REF}). A konverter bemenő feszültségtartománya V_{SS} és V_{REF} között lehet. A port lábai programból konfigurálhatók (analóg vagy digitális ki/bemenet). Az analóg bemeneteknek konfigurált lábak digitális kimenetként is működhettek, ha a TRIS bitjeiket töröljük. Digitális bemenetként konfigurált lábra analóg feszültséget téve a bemeneti puffer árama miatt túlterhelődhet. Az A/D megfelelő használatához a következő lépések szükségesek:

- 1) A/D modul konfigurálása a megfelelő ADCON regiszter bitjeinek a beállításával:
 - analóg bemeneti és referenciafeszültséghez tartozó lábak kiválasztása,
 - a mérendő A/D bemeneti csatorna kiválasztása,
 - az A/D átalakítás órajelének a megadása,
 - A/D modul bekapsolása.
- 2) Ha használjuk, akkor az A/D modul megszakításának a beállítása:
 - ADIF bit törlése, ADIE bit egybe állítása,
 - ADIP prioritási bit beállítása,
 - GIE/GIEH vagy PEIE/GIEL bit egybe állítása (ha megszakítást használunk).

- 3) A szükséges adatgyűjtési ideig várakozás
- 4) Konverzió indítása
 - GO/DONE bit 1-be állítása.
- 5) Várakozás, amíg a konverzió be nem fejeződik
 - vagy (polling) GO/DONE bit figyelése, mikor lesz 0, vagy ADIF bit figyelése, mikor lesz 1,
 - vagy megszakításos perifériakezelés használatakor várakozás az A/D megszakításra.
- 6) Ha megszakítást használunk, akkor az [ADRESH]:[ADRESL] regiszterek olvasása, ADIF bit törlése
- 7) A következő konverzióhoz az 1-es vagy 2-es pontra ugrás.

A belső referenciafeszültség használata nagyon kellemes, ha tudjuk biztosítani, hogy a tápfeszültség ne változzon, de telep használatakor ez nem biztosított. Ha 5 V-os a tápfeszültség, akkor a felbontás: $5 \text{ V}/1024 = 4,88 \text{ mV}$. Ha N bináris értéket mérünk, akkor a $N \cdot 4,88$ szorzást kell elvégeznünk a mért analóg feszültség meghatározására, ami elég nehézkes.

Külső referencia esetén ezért célszerű valamelyen kettő hatványának megfelelő értéket választani – pl.: $V_{ref} = 4,096 \text{ V}$ –, mert a felbontás egész lesz: $4,096/1024 = 4 \text{ mV}$, amivel már egyszerű számolni.

VCFG1:VCFG0: Voltage Reference Configuration bits

| | A/D V_{REFH} | A/D V_{REFL} |
|----|---------------------|---------------------|
| 00 | V_{DD} | V_{SS} |
| 01 | External V_{REF+} | V_{SS} |
| 10 | V_{DD} | External V_{REF-} |
| 11 | External V_{REF+} | External V_{REF-} |

4.3.1. Fokozatos közelítésű átalakítók fejlődése

8/16 Az előző részben lényegében a 8/12 és 8/14 bites PIC mikrovezérlőknél használt megvalósítást ismertetük. Az első A/D konverterekek 8 bitesek voltak.

A PIC18-as család bevezetésével az átalakítót továbbfejlesztették, de a kompatibilitás érdekében megtartották az eddig használt megoldást is úgy, hogy ADCON regiszterek bitjeit a helyükön hagyták. A továbbfejlesztés során a következő változások történtek: (Compatible 10 bit A/D):

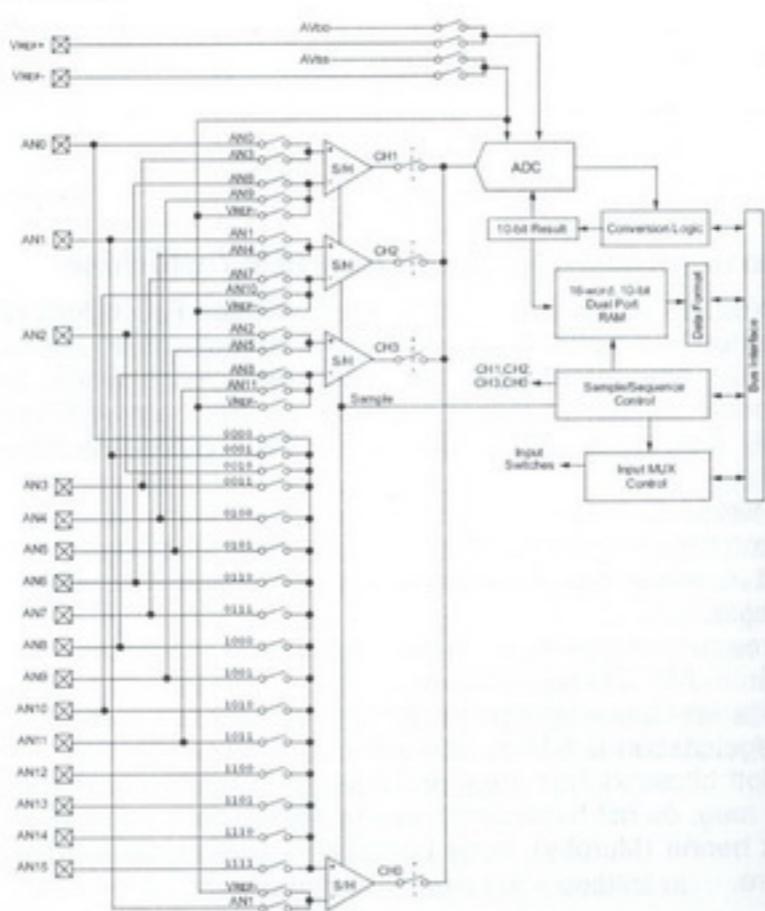
- 8 bit helyett 10 bites felbontás,
 - 8 analóg csatorna helyett maximum 16 analóg csatorna lehetséges.
- A 18-as család új, már a régievel nem kompatibilis 10 bites átalakítói a következő új tulajdonságokkal rendelkeznek:
- 8 analóg csatorna helyett maximum 16 analóg csatorna lehetséges.
 - Kettő helyett három ADCON regiszter van.
 - A legfontosabb a lábak analóg csatornához rendelésének a megváltoztatása. Mivel a kompatibilis megoldásban a 4 bites lábkonfiguráció 16 lehetőséget adott, amiben kiosztották, hogy adott bitcsoporthoz hány analóg, hány digitális csatorna tartozik, ez melyik lábakon jelenik meg, és mit használunk az átalakítás során referenciafeszültségnak, biztosak lehetünk benne (Murphy), hogy pontosan a nekünk szükséges konfiguráció nem áll rendelkezésre.

Bit 3-0: PCFG3:PCFG0: A/D Port Configuration Control bits

| | AN 15 | AN 14 | AN 13 | AN 12 | AN 11 | AN 10 | AN 9 | AN 8 | AN 7 | AN 6 | AN 5 | AN 4 | AN 3 | AN 2 | AN 1 | AN 0 |
|------|-------|-------|-------|-------|-------|-------|------|------|------|------|------|------|------|------|------|------|
| 0000 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 0001 | D | D | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 0010 | D | D | D | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 0011 | D | D | D | D | A | A | A | A | A | A | A | A | A | A | A | A |
| 0100 | D | D | D | D | D | A | A | A | A | A | A | A | A | A | A | A |

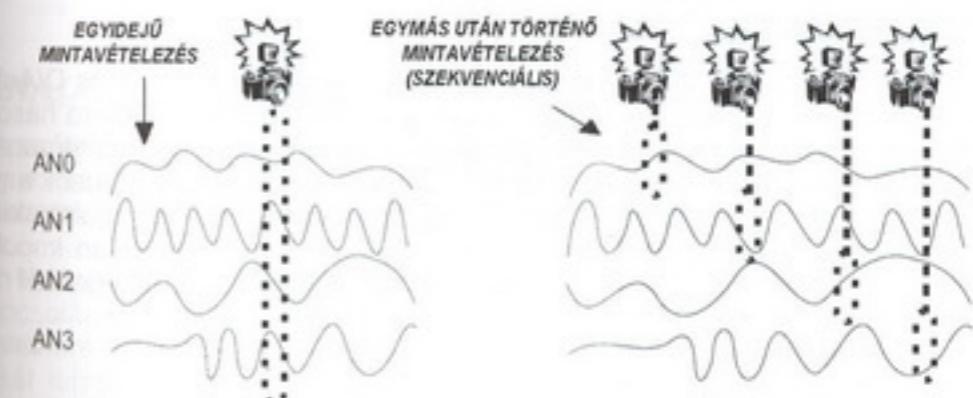
A megoldás: a külső/belső referencia választását egy bitpáros határozza meg – VCFG1: VCFG0. Külső referencia használatakor általában az AN3-ra kell kapcsolni a VREFH, és AN2-re a VREFL pontokat. Az analóg és digitális csatorna kiválasztása és elrendezése katonás lett, értelemszerűen folytatódik. Még ennél is jobb megoldás, ha egy bemenet analóg vagy digitális voltát egy erre a célra használt regiszter adott pozícióban lévő bitje dönti el, hasonlóan a portok bemeneti/kimeneti konfigurációját meghatározó TRIS porthoz, csak itt a regiszter bitjei azt határozzák meg, hogy az adott láb analóg vagy digitális legyen.

16/24 32/32 A 16 és 32 bites PIC mikrovezérlőkben új 12 bites és továbbfejlesztett 10 bites A/D modulok vannak.

4.9. ábra
10 bites A/D a 16 bites PIC-eknél

- A 10 bites A/D átalakító legfontosabb jellemzői:
max. 1 Msps (Mega sample per second – egymillió minta másodpercenként) konverziós sebesség,
- max. 16 analóg bemenet,
- külső referenciafeszültség lábak,
- négy unipoláris differenciál S/H áramkör,
- max. négy analóg bemenet egyidejű mintavételének lehetősége,
- automatikus mintavételi mód,
- több, választható mintavételt indító jelforrás,
- 16 szavas puffer a konverziók eredményének tárolására,
- több, választható puffertöltési mód,
- a konverziós eredmények négy alakban történő formázási lehetősége,
- a konverter működhet a CPU SLEEP és IDLE üzemmódjában is.

Az átalakító több lehetőséget biztosít a konverzióra. A legegyszerűbb a szokásos: egy S/H áramkör és analóg multiplexer. Sokkal hatékonyabb megoldás, hogy az átalakítóban elhelyeztek négy S/H áramkört, ami maximum négy csatorna **egy időben történő (szimultán)** mintavételezését teszi lehetővé. A szimultán/szekvenciális mintavételt a SIMSAM (ADCON1<3>) bit határozza meg. A konverzió a SAMP vezérlő bit 1-be állításával indítható, de ez automatizálható a hardverrel (Auto-Sample Mód). Ezt az ASAM (ADCON1<2>) bit bekapcsolásával érhetjük el. A konverzió indítását több hardverféléthez köthetjük, indíthatjuk a SAMP bittel, és használhatjuk az Auto-Sample módot. Ilyenkor a mintavétel gyakoriságát egy számláló programozásával tudjuk beállítani. Mivel az eredmények egy 16 regiszteres pufferbe kerülnek, ezért beállítható, hogy hány (1...16) konverzió után legyen megszakítás. Ez az SMPI ADCON2<5:2> bitek programozásával állítható.

4.10. ábra
Szimultán és szekvenciális mintavétel

- Ezt továbbfejlesztett 10 bites A/D modult hat regiszterrel kezeljük:
- ADCON1-3: szerepük az A/D modul működésének a vezérlésében van.
 - ADCHS: A/D bemeneti csatorna választása. Kiválasztjuk azokat a bemeneti lábakat, amelyek az S/H áramkörökhez csatlakoznak.
 - ADPCFG: A/D portkonfigurálás: melyik láb legyen analóg, és melyik legyen digitális.
 - ADCSSL: A/D bemeneti letapogatást választó regiszter. Itt választjuk ki, hogy melyik bemenetek feszültségét konvertáljuk.
- A konverziók eredménye egy 16 szavas, csak olvasható ADCBUF regisztertömbbe kerül. A 10 bites eredmény a kiolvasás előtt 4 választható alak valamelyikére formázható. Ezt két bit: FORM<1:0> (ADCON1 <9:8>) állításával választhatjuk ki.

| Puffer tartalma | → | d09 d08 d07 d06 d05 d04 d03 d02 d01 d00 |
|--|---|---|
| Kiolasás | | |
| Integer 0 ... 1023 | | 0 0 0 0 0 d09 d08 d07 d06 d05 d04 d03 d02 d01 d00 |
| Signed integer -512...511 | | d09 d09 d09 d09 d09 d09 d08 d07 d06 d05 d04 d03 d02 d01 d00 |
| Fractional (1.15) 0.000...0.999 | | d09 d08 d07 d06 d05 d04 d03 d02 d01 d00 0 0 0 0 0 0 |
| Signed fractional (1.15) -0.500...0.499 | | d09 d08 d07 d06 d05 d04 d03 d02 d01 d00 0 0 0 0 0 0 |

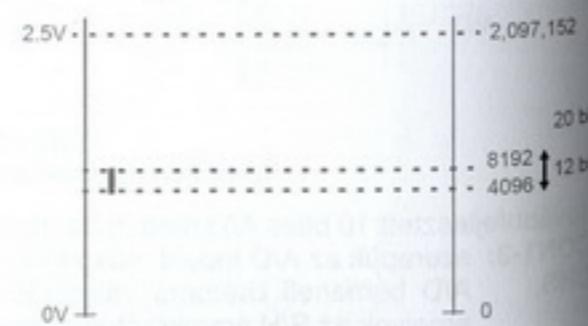
4.11. ábra
A/D eredményformátumok

Számos típusú A/D konfiguráció és konverzió megvalósítható, amit a 16 bites regiszterek bitjeinek az állításával érhetünk el. Ez elég bonyolult. Ezért a Microchip a referencia-kézikönyv 10 bites A/D konvertert bemutató fejezetében számos mintapéldát ismertet.

4.4. DELTA-SZIGMA ÁTALAKÍTÓK

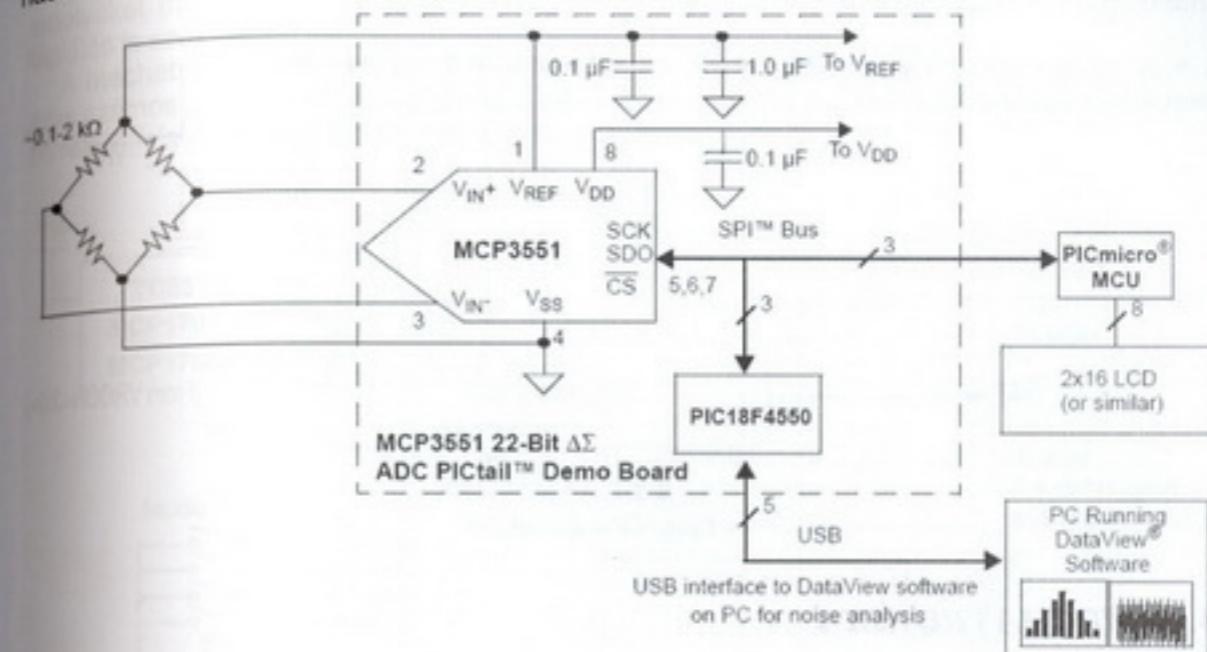
Impulzussűrűség-modulátoroknak tekinthetők, amivel nagy felbontású A/D és D/A átalakítás lehetséges. A/D átalakítás esetén a módszer a feszültségvezérelt oszcillátorra hasonlít, ahol a vezérlő feszültség a mérrendő érték, és egy negatív visszacsatolás biztosítja az átalakítás linearitását. Az oszcillátor kimenete egy impulzusorozat, amiben az impulzusok amplitúdója állandó: V, hossza: dt. Az impulzusok közti szünet időtartama változó, az átalakítandó feszültségtől függ. Ennek képezhető az integrálja. Vagyis kis feszültség olyan impulzusorozatot generál, ahol a pulzusok között nagy a szünet, míg nagyobb feszültségeknél ritkább.

Úgy is megfogalmazhatnánk, hogy a szünetek átlaga fordítottan arányos a bemenő feszültséggel. Ha az impulzus hossza nagyon rövid (ezt hívjuk a szabályozástechnikában Dirac-deltának), akkor egy adott átlagolási periódusban az impulzusok száma (ez összeadás, jele: Σ - szigma) arányos lesz a bemenő feszültséggel. Ezért a neve delta-szigma átalakító. A Microchip által gyártott egyik jellegzetes típus az **MCP3551** jelzésű, amely 22 bites konverziós eredményt ad, ami SPI illesztésen keresztül érhető el. Ilyen felbontás mellett már komoly zajok is fellépnek, amit megfelelő szűréssel és jelanalízissel lehet kézben tartani. Az igen nagy felbontás miatt nem szükséges az analóg jelek konverzió előtti erősítése, amit a 4.13. ábra is illusztrál.



4.12. ábra
12 bites – 22 bites tartomány

A Microchip által forgalmazott MCP 3551 22 bit ADC PICtail demókártya kitűnő lehetőséget biztosít az áramkör jellemzőinek megismerésére, amit az ingyenes DataView program használata is segít.



4.13. ábra
Delta-szigma demókártya használata

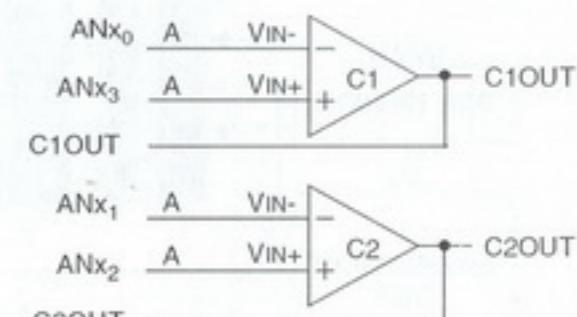
4.5. ANALÓG KOMPARÁTOROK

Egy komparátor áramkör a kimenetén attól függően ad ki alacsony vagy magas feszültséget, hogy a két bemenete közti feszültség milyen irányú (vagyis melyik bemenetén van nagyobb feszültség). A PIC-eknél a komparátorperiféria két, C1, illetve C2 jelű komparátort tartalmaz.

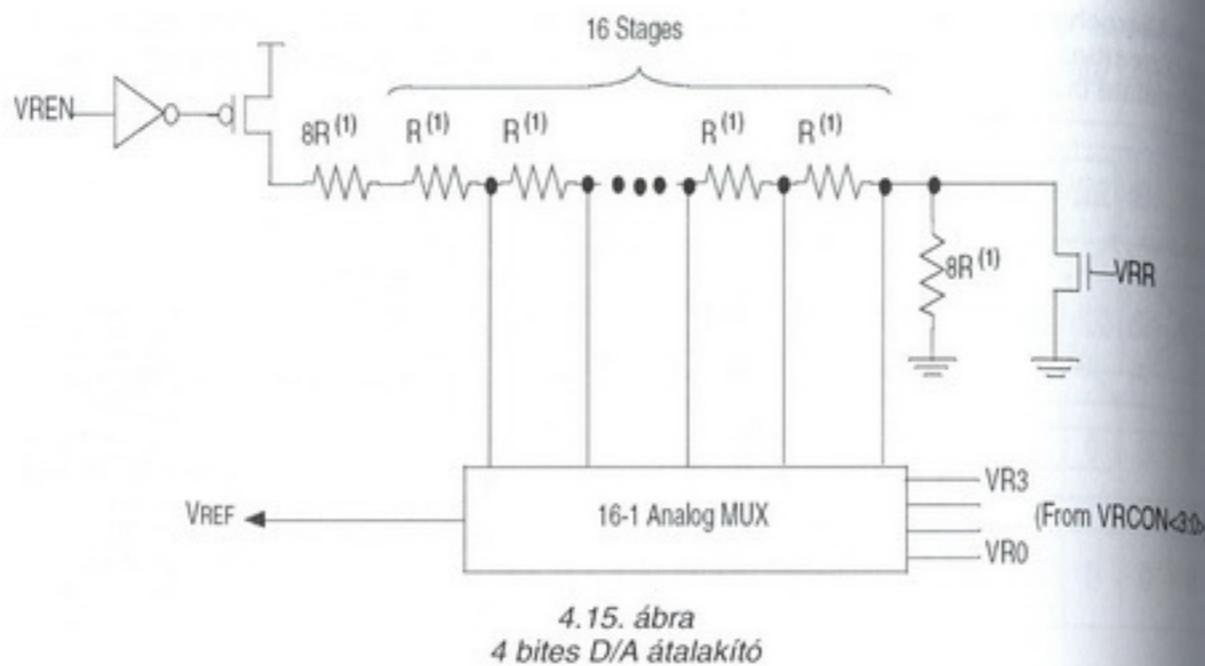
A két komparátor különféle módon történő összekapcsolásával összesen 8 működési mód alakítható ki (reset komparátor, kikapcsolva, két független komparátor, közös referenciajú stb.). A komparátorokhoz a CMCON regiszter van hozzárendelve.

A komparátorokat, amennyiben az egyik bemenetükre egy programmal változtatható feszültséget, míg a másikra a mérrendő feszültséget kapcsoljuk, feszültségmérésre is használhatjuk. Ezért minden komparátor-modulhoz tartozik egy 16 lépében programozható referenciafeszültség-modul.

V_{REF} kimenetén a 16 szintból álló feszültség valamelyike jelenik meg, a szint nagyságát VRCON<VR3:VR0> négy bitjének bináris értéke adja. V_{REN} kapcsolja (ON/OFF) a feszültséget a referencia-áramkörre, lényegében D/A átalakítónak használható. A működést a VRCON regiszter tartalma határozza meg.

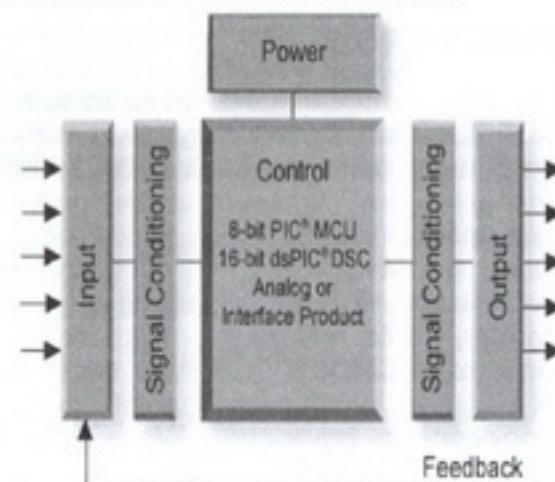


4.14. ábra
Analóg komparátorok



4.6. MECHATRONIKA

A mechatronika a mechanikai – gépészeti – berendezésekhez elektronikát ad, bizonyos mechanikai megoldásokat elektronikával vált fel. Az autóipar az egyik területe a mechatronika alkalmazásának. A hagyományos mechanikus elven működő termosztátokat, gyűjtőszabályozókat, fékeket elektronikus eszközök váltották fel.

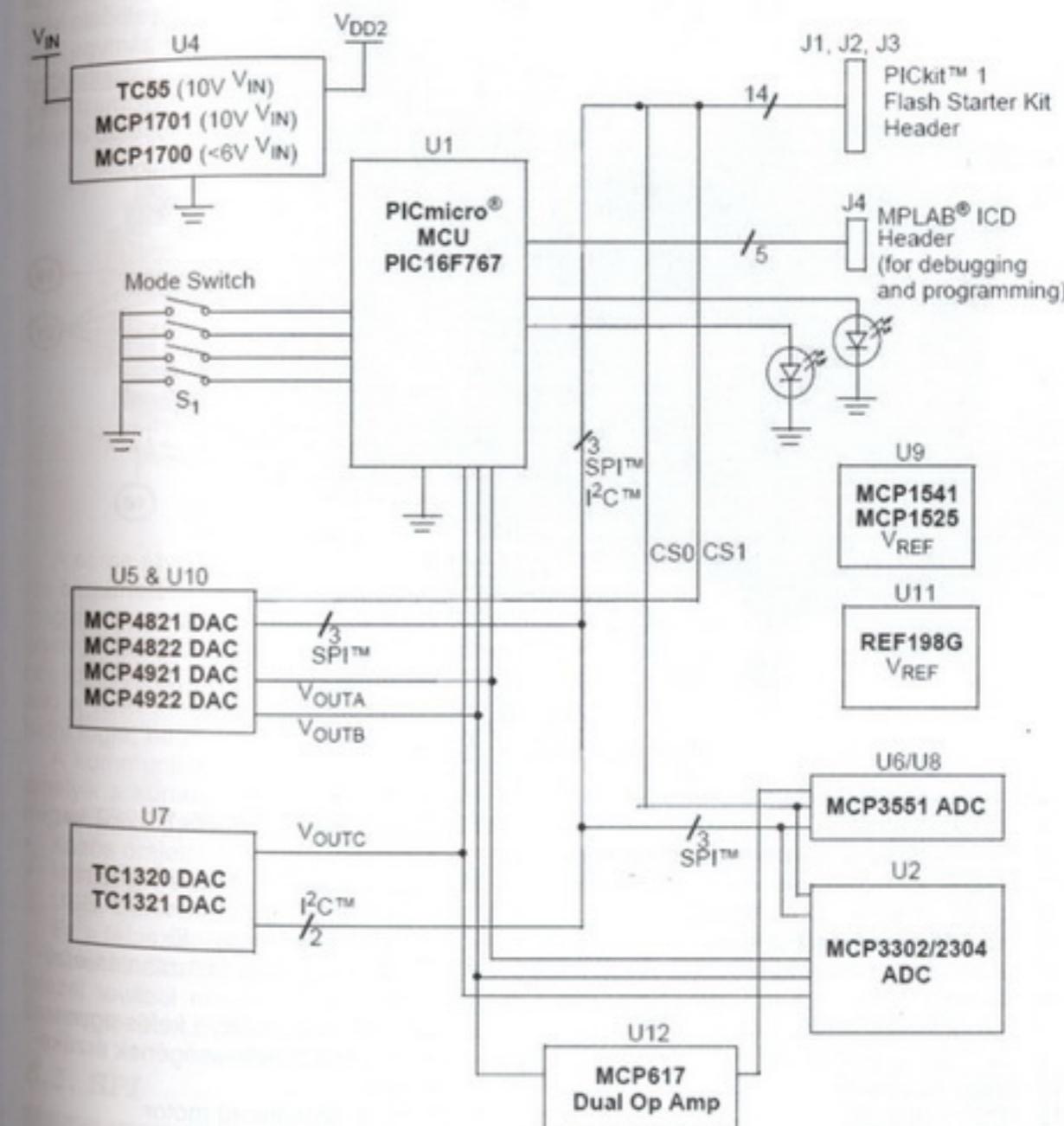


A mechatronikai megoldásoknak számos előnyük van: felhasználóbarátok, kezelőszervekkel, kijelzőkkel elláthatók, megnöveált képességekkel és általánosabb felhasználhatósággal rendelkeznek, finomabban vezérelhetők, ezért pontosabban és hatékonyabban működnek, könnyen áttervezhetők (újraprogramozhatók), kisebbek, könnyebbek, nagyobb megbízhatóságúak.

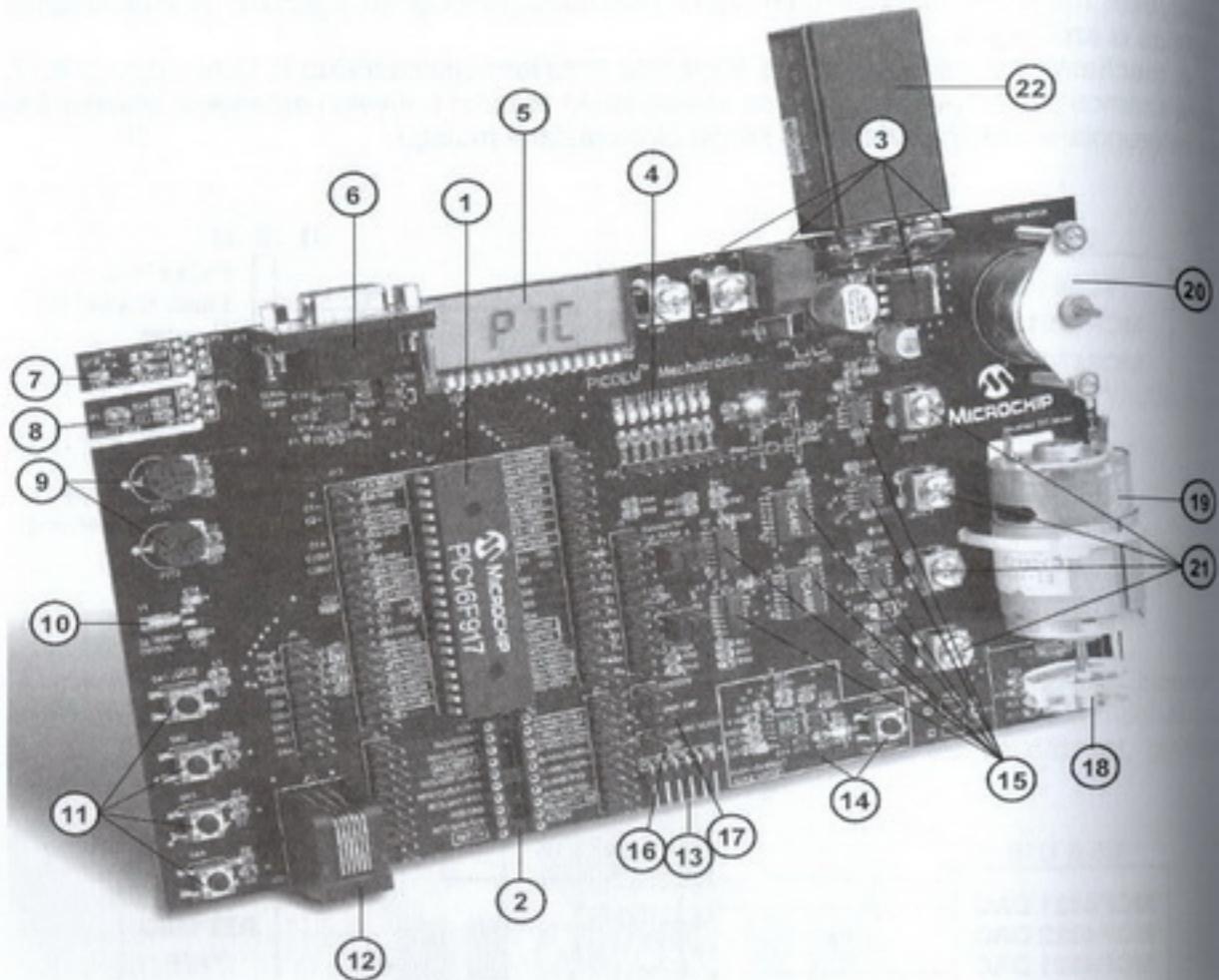
Nagyon sok esetben az előállításuk olcsóbb, mint a hasonló feladatokat ellátó mechanikus felépítésű szerkezeteké.

A mechatronikai kialakítás egy vezérlő/szabályozó rendszert eredményez. A jelbemenetek a mikrovezérlőhöz kapcsolódnak megfelelő jelátalakítás után. A mikrovezérlőben lévő programmal megvalósított szabályozási algoritmus segítségével állítja elő a kimeneteket, amelyeket megfelelő illesztés után kapcsolhatunk a mechanikai részhez. A működéshez táplálás is szükséges.

A mechatronikai fejlesztéseket a Microchip hardvermegoldásokkal is támogatja. A 4.17. ábra számos „kevert jelű” (digitális és analóg jelek) eszközt (műveleti erősítőket, átalakítókat és referencia-áramkört) tartalmazó kártya blokkvázlatát mutatja.



A 4.18. ábrán kijelző, kezelő, érzékelő és beavatkozó elemeket tartalmazó kártya fényképe látható, számmal jelölve a kártyán lévő egységek helyzetét; a táblázatban megtalálható az egyes egységek megnevezése is.

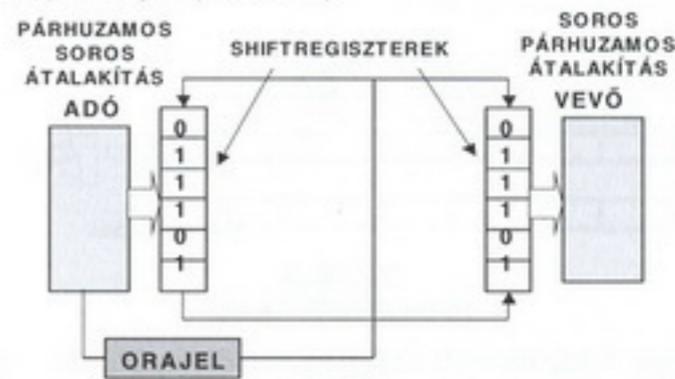


4.18. ábra
Mechatronics demóáramkör

- | | |
|--------------------------------------|---|
| 1 40 lábú foglalat | 13 ICSP csatlakozó |
| 2 20 lábú foglalat | 14 Túláramvédő áramkör RESET kapcsolóval |
| 3 Feszültségszabályozó és csatlakozó | 15 Négy MOSFET félhíd meghajtó áramkörökkel |
| 4 8 LED | 16 Kimenetiáram-érzékelő |
| 5 39 szegmenses LCD kijelző | 17 Elektromágneses visszahatás-érzékelő |
| 6 RS 232 foglalat + áramkör | 18 Optikai megszakító a kefés egyenáramú motor sebességének érzékeléséhez |
| 7 Hőméréklet-érzékelő | 19 Kefés egyenáramú motor |
| 8 Fényérzékelő | 20 Léptető motor |
| 9 Két potenciométer | 21 Erősáramú csavaros csatlakozó |
| 10 32 768 Hz-es kristály | 22 9 V-os telep |
| 11 Négy kapcsoló | |
| 12 ICD csatlakozó | |

5. KOMMUNIKÁCIÓS PERIFÉRIÁK

Ilyen perifériák felhasználásával kapcsolatot létesíthetünk és információt cserélhetünk más eszközökkel. A kommunikációt elméletileg leghatékonyabban több bit egyszerre, azaz párhuzamosan történő átvitelével tudjuk megvalósítani, azonban a **párhuzamos átvitel** fizikai nehézségei miatt – a jelek egymásra hatása, eltérő késleltetések – jelenleg már elsődlegesen a **soros átvitelt** használják. Ehhez meg kell oldani a bitcsoportok soros, egymás utáni bitekké történő átalakítását, ami a digitális technikában jól ismert léptetőregiszter (shiftregiszter) segítségével valósítható meg: az átalakítandó bitcsoportot párhuzamosan betöljtük egy léptetőregiszterbe, majd egy órajel felhasználásával egyenként kiléptetjük a bitcsoport bitjeit (5.1. ábra).



5.1. ábra
Soros átvitel

A soros átviteli csatorna másik végén szintén egy léptetőregiszter található, aminek soros bemenetére jutnak a bitek, és végiglépve a regiszteren, annak párhuzamos kimenetén megjelenik a párhuzamos bitcsoport. A kommunikációs csatorna két vonalat tartalmaz: az adatvonalat és az órajelet továbbító vonalat. Ha biztosítani tudjuk a két oldal közötti azonos időmérést, akkor az órajelvonal elhagyható. Ezt hívják szinkronizációknak. Ilyenkor az adó a vevő által is ismert időpontokban küldi a biteket, és a vevő a saját idejét figyelve tudni fogja, hogy mikor melyik bit érkezett meg.

A kommunikáció irányá szerint megkülönböztetünk **ADÓt** és **VEVŐt**. Azt az egységet, amelyik a kommunikációt vezéri, a szinkronitást meghatározza, **MESTERnek** nevezzük, míg a többi egység(ek) neve: **SLAVE (SZOLGA)**.

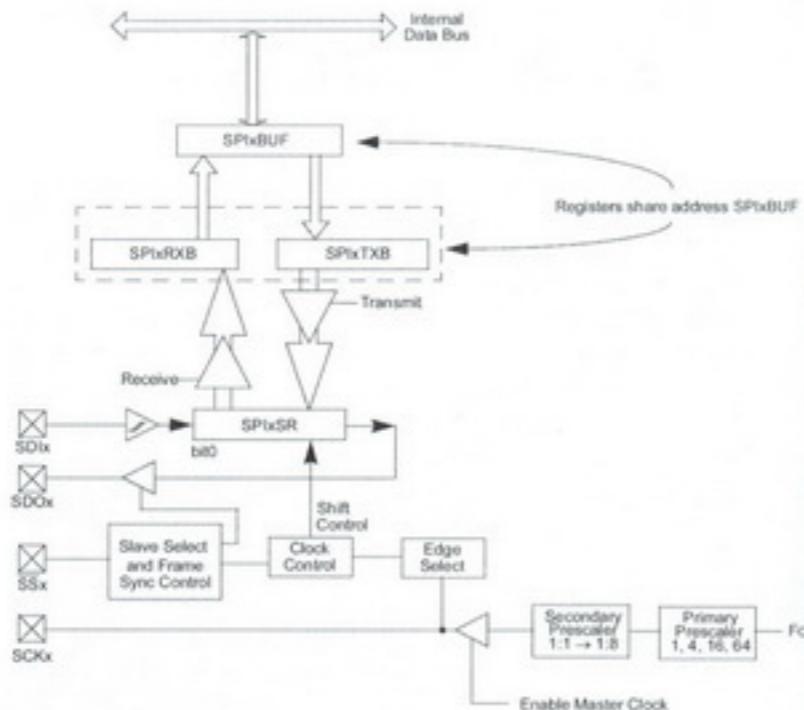
- Közös órajelet használó megoldások: az SPI, I₂C átvitel.
- Időszinkronizációt használó megoldások: **soros aszinkron, szinkron, LIN busz, CAN, USB, Ethernet, egyvezetékes** átvitelek.

Ez a felosztás meghatározza a fejezetben történő bemutatás sorrendiségét is.

Mivel az előző könyvben a kommunikációs perifériákat elég részletesen bemutattuk, ezért a fejezet további részében inkább a 16 bites mikrovezérlőkben bevezetett új megoldásokra összpontosítunk, nem az adatlapok, hanem a működés és a jelek megértésének a szintjén.

5.1. SPI

16/24 32/32 Az SPI interfész léptető regisztereiken alapul. A kommunikáció mester-szolga jellegű, ahol a mester vezéri a kommunikációt: szolgáltatja az órajelet, valamint egy külön vezetékkal engedélyezi a hozzákapcsolt szolga perifériákat.



5.2. ábra
16 bites SPIx modul

Ennek az a hátránya, hogy minden szolgához külön kiválasztó vezetéket kell használni, de mivel egyszerre csak egy periféria működhet, ezért a szükséges I/O lábak csökkenése érdekében használhatunk 2-ból 4, vagy 3-ból 8 dekódert. A kommunikáció egyszerre kétirányú – vagyis míg a léptetőregiszter bemenetén sorosan fogadja a biteket, ugyanakkor a regiszter másik végén kilépettük annak előző tartalmát. Ezért például 16 órajellel a mester és szolga 16 bites léptetőregisztereinek a tartalma helyet cserél. A PIC mikrovezérlőkben használt SPI modulok fontos tulajdonsága: az órajel élének és polaritásának megválasztása, valamint a választható mester vagy szolga mód.

A 16 bites mikrovezérlőknél – a típustól függően – egy vagy két egyforma SPI1, illetve SPI2 jelzésű modul található. A továbbiakban x-szel jelöljük a modulok sorszámát (1 vagy 2, felépítése az 5.2. ábrán látható).

A modulok működését három SFR regiszter határozza meg:

- SPIxBUF:** A küldött és a fogadott adatokat tárolja, olyan módon, hogy ugyanazt a címet használják a tényleges adatokat tároló **SPIxTXB** és **SPIxRXB** regiszterek.
- SPIxCON:** SPIx modul működését vezérli bitek.
- SPIxSTAT:** Az SPIx modul állapotát (státusát) tárolja.

A tényleges eltolást **SPIxSR** 16 bites shiftregiszter végzi. Nem SFR regiszter, címezni nem lehet, tehát programból nem elérhető. Kúlsőleg a modulhoz négy láb tartozik:

- SDIx:** serial data input – soros adatbemenet
- SDOx:** serial data output – soros adatkimenet
- SCKx:** shift clock órajelbemenet ha modul szolga. Kimenetként működik, ha a modul mesterként funkcionál.
- SSx:** (Slave Select) aktív alacsony szintjét a szolga kiválasztásához vagy a keretszinkronizáló I/O pulzushoz használjuk.

5.1.1. 16 bites SPI működési módok

Az SPI-nek három működési módja van:

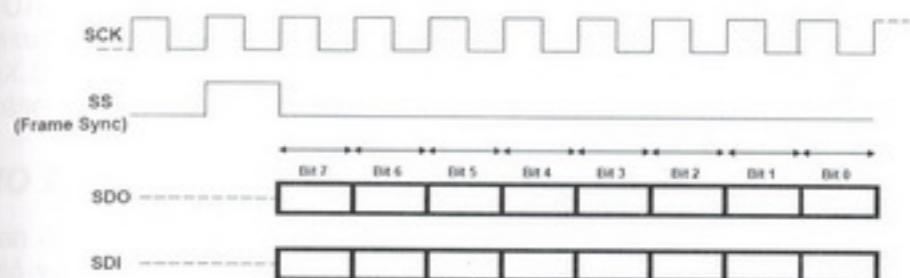
- 8 bit és 16 bit adatküldés/fogadás

- Mester és szolga módok

- Keretezett SPI módok

A **MODE16** (**SPIxCON<10>**) vezérlő bit határozza meg, hogy az átvitel 8 vagy 16 bites legyen. Működés közben ezt a bitet nem szabad állítani, mert ez alapállapotba állítja a modult. Nyolcbites módban a kiküldött bit a 7., illetve tizenhat bites módban a 15. pozícióból lép ki, míg a beolvasáskor az érték minden esetben a 0. bitre érkezik.

A **mester és szolga módok** megegyeznek a 8 bites SPI modul azonos működésmódjával. Speciális SPI működési mód a CODEC áramkörökkel történő kommunikációra a **keretezett SPI mód**. Ilyenkor az órajel folyamatos, nem korlátozódik az átvitel idejére. Használva az előzőekhez az SPI modul képes mesterként vagy szolgaként működni, ami meghatározza az órajel és a **FRAMESYNC** nevű, egy órajel hosszúságú, magas szintű keretezőimpulzus irányát.



5.3. ábra
Keretezett SPI mód

5.2. I2C

Először a Microchip olyan modult alakított ki, amelynek a neve: **SSP – Synchronous Serial Port** –, egy olyan áramkör, amely **vagy SPI, vagy I2C szolga** üzemmódban tudott működni, vagyis egyszerre nem lehetett a két perifériát használni.

A PIC18-asoknál megjelent az I2C mester módja is, de az SPI-vel való közösség megmaradt.

16/24 A 16 bites mikrovezérlőknél az I2C modult jelentősen továbbfejlesztették: önálló modullá alakították, és egyszerre képes egymástól független mesterként és szolgaként működni. Az üzemmódok:

- szolga I2C,
- mester I2C (ilyenkor a szolga is aktív lehet),
- mester/szolga I2C többmesteres kialakításként (figyelve az ütközést, a buszfoglalás megvalósítását).

Az I2C modul önálló I2C mester- és önálló I2C szolgaáramkört tartalmaz, saját megszakításkezeléssel. Ezért többmesteres környezetben a szoftver egyszerűen szétválasztható mester- és szolgakezelésre.

Ha az I2C mester működik, a szolga szintén aktív marad, figyelve az I2C busz állapotát, és képes a buszon folyó üzenetek vételére. A buszfoglalás során nincs üzenetvesztés többmesteres környezetben.

A modulnak önálló, 100 kHz és 400 kHz-es működést biztosító órajel-generátora van. Képes 7 és 10 bites eszközcímek kezelésére, az I2C protokollban definált általános hívási címeket is érzékelni.

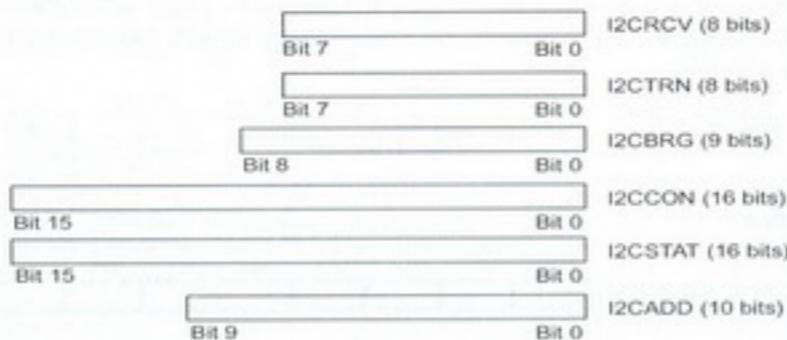
Képes az SCL órajel megnyújtására, hogy a processzor válaszolni tudjon a szolga adatkérésére.

Az I2C modulnak hat, a felhasználó által elérhető regisztere van, amelyek bájtosan vagy 16 bitesen érhetők el.

- Control Register (I2CCON):** I2C működését vezéri.

- Status Register (I2CSTAT):** A modul állapotát jelző biteket tartalmazza.
- Receive Buffer Register (I2CRCV):** Innen olvasható ki a vett adatbájt (csak olvasható).
- Transmit Register (I2CTRN):** Ide kell beírni az elküldendő adatbájtot (írható/olvasható).
- Address Register (I2CADD):** A szolgacímét tartalmazza.
- Baud Rate Generator Reload Register (I2CBRG):** A baud rate generator periódus idejét határozza meg a tartalma.

Az 5.4. ábrán foglaltuk össze, hogyan látja a regisztereket a programozó.

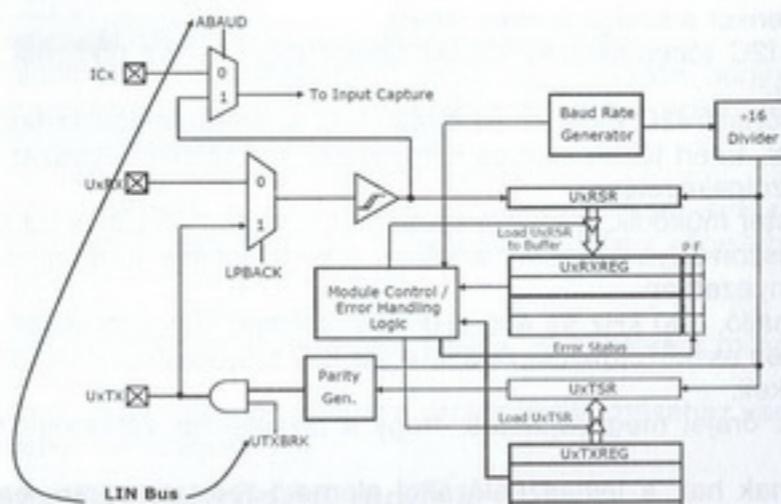


5.4. ábra
I2C programozói modell

5.3. ASZINKRON SOROS ÁTVITEL

Az előző kiadásban részletesen bemutatott *Universal Asynchronous Receiver Transmitter* (UART) modul működésén nem változtattak jelentősen. A modult négy területen használják:

- 1) Két állomás közötti RS232 pont–pont kapcsolatként: ezt szokták „soros port”-nak hívni, terminálok, modemek, számítógépek összekötésére.
- 2) RS485 (EIA-485) több pontos soros kapcsolatra, elsősorban az ipari berendezéseknél.
- 3) LIN busz: az előző kiadásban leírtuk a működését. Olyan UART-ot igényel, amely képes a kommunikációs sebesség automatikus meghatározására.
- 4) Infravörös vezeték nélküli kommunikációra. 38–40 kHz-es modulációt használ, és optikai jeladó-vevőket.

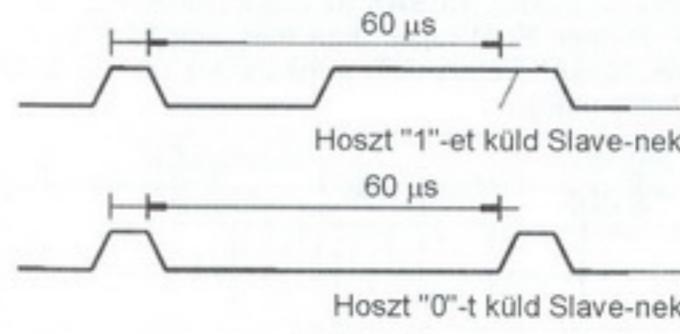


5.5. ábra
UART modul blokkvázlata

- A 16 bites mikrovezérlők UART moduljának legfontosabb jellemzői:
- Két UART modul is lehet egy mikrovezérlőben.
- Az adatformátumban lehet két STOP bitet definiálni.
- Önálló Baud Rate Generátor 16 bites előosztóval.
- A sebesség 29 bps-től 1,875 MB/s-ig (FCY = 30 MHz-nél) változhat.
- 4 tárolós First-In-First-Out (FIFO) adó puffer.
- 4 tárolós FIFO vevő puffer.
- Paritáshiba, keretezési hiba és puffertúlcordulási hiba jelzése.
- 9 bites átvitelnél címdetektálás (9. bit = 1).
- Adási és vételi megszakítás.
- Visszacsatolási (Loopback) mód diagnosztikai célokra. Ilyenkor az UART TX kimenete kijut az adólábra, de az UART vevő bemenete is rákapcsolódik. A VEVÖ lab leválasztódik az UART vevő bemenetéről.
- Automatikus baud-rate beállítás is lehetséges a vett karakterek alapján. Ilyenkor az UART RX bemenete egy kiválasztott bemeneti capture csatornára kapcsolódik a baud rate méréséhez. Ez a beállítás nem minden eszközön van megvalósítva.

5.4. UNIO BUSZ

Elsődlegesen a külső EEPROM memóriákkal való kapcsolatra fejlesztették ki az egyetlen kommunikáló vezetéket használó UNIO buszt. Ilyen megoldást használva az EEPROM-hoz való kapcsolat csupán három vezetéket igényel: tápfeszültség (Vcc), föld (Vss), és a kétirányú adatvonal (SCIO). Az egy vezetéket használó megoldás (1 wire bus) jól ismert a DALLAS, illetve ma már a MAXIM termékekben. Az átvitel alapja a pontos időzítés, ezért a biteket időszekletekben visszük át. minden szelet min. 60 µs hosszú, és ezeket min. 1 µs hosszú, H szintű szinkronjel választja el. minden időszeklet kezdetét a hoszt jelöli ki a vonal alacsony szintre húzásával. Az adott időszeklethez tartozó bit értékét az határozza meg, hogy a szinkronjel után 30–50 µs múlva a vonal alacsony vagy magas.



5.6. ábra
DALLAS 1 vezetékes kommunikáció

A Microchip UNI/O™ busz megoldásánál az Ethernet átvitel Manchester kódolását használják. A mester-szolga típusú kommunikációban a szinkronitást az biztosítja, hogy a szintváltások (az élek) hordozzák az információt.

- 1) A mester (mikrovezérlő) határozza meg az adatátviteli sebességet a start fejléccel.
- 2) A szolga UNI/O™ eszköz (az EEPROM) készenléti állapotba kerül, ha nem érzékel a várt jelet.
- 3) A szolga hozzászinkronizálódik a mester sebességéhez, ami 10 kHz – 100 kHz órajelfrekvenciát jelent.

Mindkét résztvevő lehet adó, illetve vevő, de a mester határozza meg, hogy mikor melyik mód aktív.



5.7. ábra
A lefutó él a „0”, míg a felfutó él az „1” bitet jelzi

UNI/O™ Busz Bit Periódusa

Ennek idejét (T_E) a mester határozza meg. Periódusonként egy adatbit van, és a periódus közepén lévő átmenet irányára határozza meg az adatbit értékét: lefutó él a „0”, míg a felfutó él az „1” bitet jelzi. A bit periódusidő 10 µs és 100 µs között van (5.7. ábra).

UNI/O™ EEPROM parancs-összefoglaló

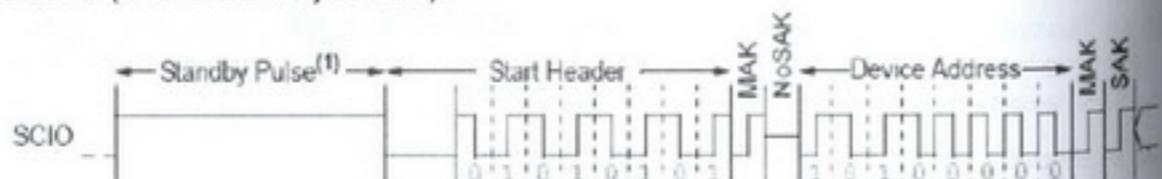
Standby pulzus: Egy hosszú pulzust használunk, hogy az eszköz standby állapotba jusson, és képes legyen venni a parancsokat. Ez szükséges POR/BOR után, vagy ha valami hiba történt. Nem kell, ha több egymást követő parancsot hajtunk végre ugyanazon az eszközzel.

Start fejléc (Header): Ezzel szinkronizálja az UNI/O™ eszközt a mester. A mesternek ugyanazt a bitperiódust kell tartania a következő fejlécig. Az UNI/O™ eszköz ugyanezzel a periódusidővel ad.

Nyugtázás (Acknowledge):

MAK (Master Ack): A mester által küldött minden bájt után következik. MAK: „1” állapotú bitként lesz küldve, míg a **NoMAK** „0”-ként. Ez utóbbit használjuk egy művelet befejezésekor és egy írási ciklus inicializálásakor írási parancsnál.

SAK (Slave Ack): Szolga jelzi a mesternek, hogy a kapott bájt vétele rendben volt. SAK „1” állapotú bitként lesz küldve. NoSAK az állapotváltás hiányát jelzi. SAK-ot minden küldjük, ha nincs hiba. Kivétel: Start fejléc után nem küldjük. NoSAK-ot hibakor, illetve Start fejléc után küldünk. NoSAK vétele után a mesternek feltétlenül kell Standby pulzust küldenie (kivéve Start fejléc után).

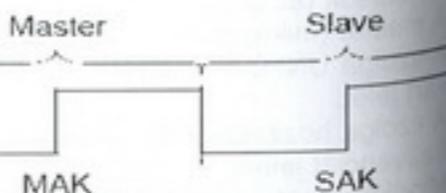


5.8. ábra
Standby pulzus és fejléc

A MAK/SAK sorozat biztosítja, hogy a mester és a szolga minden szinkronban marad, valamint csökkenti a hibás írás esélyét a mikrovezérlő hibás működése miatt. A mester 8 bitenként ellenörzi a szolgával való szinkronitást.

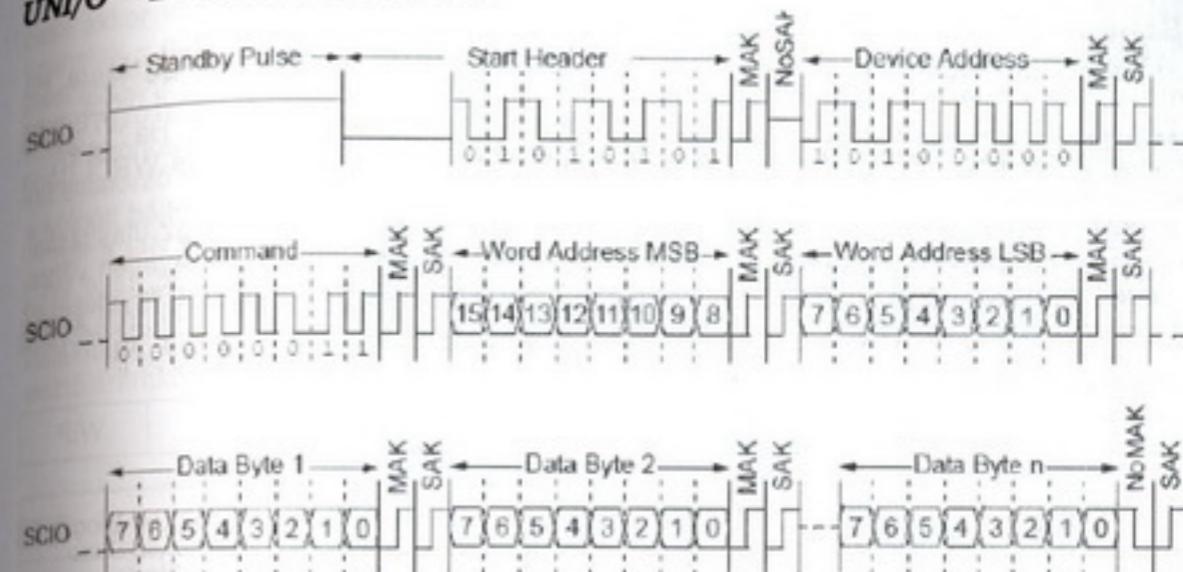
UNI/O™ eszközcím

UNI/O™ EEPROM-ból több eszköz is lehet ugyanazon a buszon. minden eszköztípusnak egy címe van, a már gyártásban lévő 11XXXX0 eszköz címe: 0xA0, az újabb eszközöknek más címük lesz. Az eszközcím minden parancsot megelőz.



5.9. ábra
MAK-SAK nyugtázás

UNI/O™ EEPROM olvasása



5.10. ábra
Olvasási parancs

Ha a bitperiódus és az eszközcím már adott, az EEPROM képes arra, hogy a sorban egymást követő bájtokat olvassa ki. A szolga 8 bitet küld az adott bitperiódus-idővel, és ezt a mester fogadja. Sorozatos olvasásnál a mester MAK-ot küld minden adatbáj után, a szolga erre SAK-kal válaszol, és küldi a következő 8 bitet. A címmutató automatikusan növekszik.

UNI/O™ EEPROM írása

Az első művelet az írásengedélyezés (Write Enable – WREN), ami 1-be állítja az írásengedélyezés bitet – Write Enable Latch (WEL) –, és ez szükséges minden tömb- és státusregiszter-írásnál.



5.11. ábra
Írási parancs

Bájt írása: Ha WEL = 1, akkor végrehajthatunk egy írási utasítást, amiben először sorban elküldjük a fejlécet, az eszközcímet, valamint cím MSB és LSB bájtját. Ha a bájtok nyugtázása megtörtént, a mester küldi a beírandó adatbájtot. Az írás végrehajtásához 5 ms szükséges.

Lap írása: Ilyenkor 16 bájtot írhatunk egy parancsal, aminek minden 16-tal osztható kezdőcímmel kell indulnia. Elsőként a WEL = 1 mellett ki kell adni a WREN parancsot is. Az írás végrehajtásához itt is 5 ms szükséges.

UNI/O™ EEPROM státusregiszter

Szerepe az írás és az adatvédelem biztosítása (az 5.12. ábrán látható).

Write-In-Process (WIP) bit jelzi, hogy az EEPROM kész-e az írás végrehajtására. Ha ennek a csak olvasható bitértéke 1, akkor írás van folyamatban, mikor 0, akkor nincs.

Write Enable Latch (WEL) bit jelzi az írásengedélyező flip-flop állapotát. Ha '1', akkor lehetséges az írás, ha '0', akkor nem. Ezt a bitet a WREN (1-be állítja) és WRDI (töríti) utasítások állítják.

A státusregiszter olvasása (RDSR): a státusregiszter érhető el ezzel az utasítással. A regiszter bármikor kiolvasható, még írási ciklus alatt is (ilyenkor az az egyetlen végrehajtható parancs). MAK küldésével ismételhető az olvasása, ami WIP bit figyeléséhez használható.

Status register

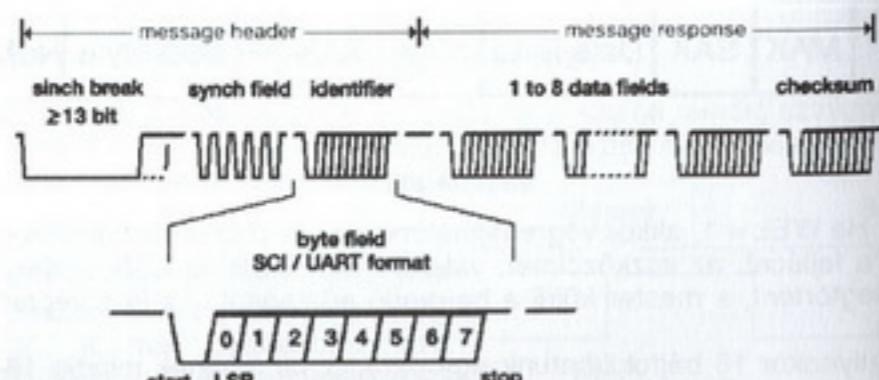
| - | - | - | - | BP1 | BP0 | WEL | WIP |
|-----------------------------|-----|------------------|---|-----|-----|-----|-----|
| CONTROL | | | | | | | |
| BP1, BP0 – Block Protection | | | | | | | |
| BP1 | BP0 | Protected Blocks | | | | | |
| 0 | 0 | None | | | | | |
| 0 | 1 | Upper ¼ of Array | | | | | |
| 1 | 0 | Upper ½ of Array | | | | | |
| 1 | 1 | All | | | | | |

5.12. ábra
Státusregiszter

Státusregiszter írása (WRSR): ezzel az utasítással tudunk, a BP1:BP0 biteket beállítva, védeni bizonyos memóriaterületeket írás ellen. Először WEL bitet 1-be kell állítani a WREN parancssal.

5.5. LIN

A LIN (Local Interconnect Network) busz az európai autóipar által kifejlesztett olcsó, rövid távolságokra (max. 40 m) használható alacsony sebességű hálózat. A kétféle kommunikáció a buszon egy vezetéken történik. A buszon lévő jelek időzítéséhez elegendő az olcsó RC oszcillátor által biztosított pontosság. Mivel az időzítés változhat, ezért a protokoll minden üzenetfogadáskor egy automatikus sebességmeghatározást végez. A bájtátviteli protokoll megegyezik az aszinkron soros átvitelnél megszokott megoldással.

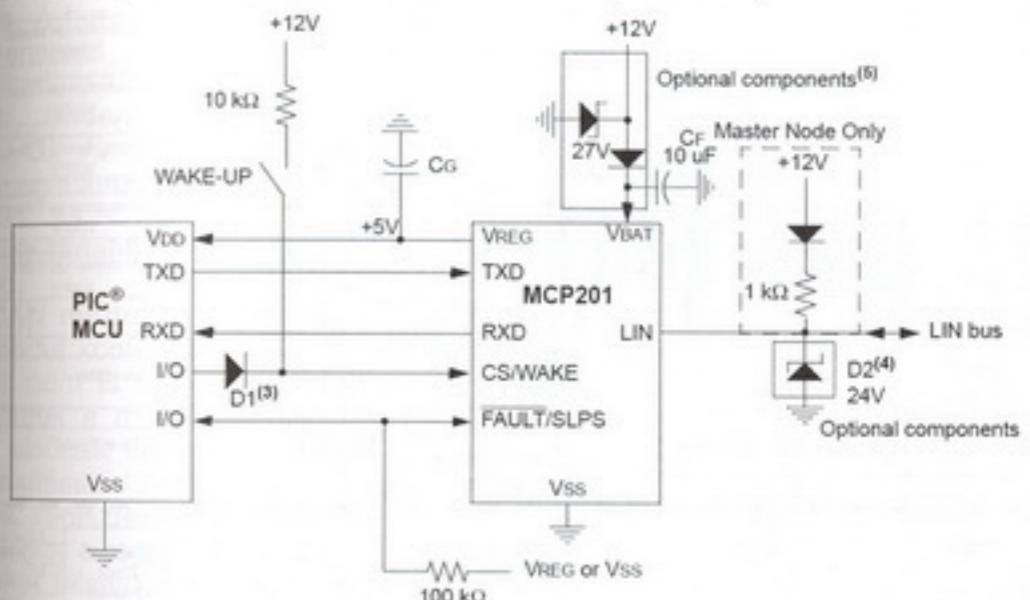


5.13. ábra
LIN protokoll

A mester a szolgák folyamatos lekérdezésével vezéri a kommunikációt, és egy szolga csak akkor válaszolhat, ha a mester megszólítja. A küldött és fogadott keretek – azaz üzenetek – egy keretkezdetet kijelölő 13 bitidőig alacsony szinten maradó „Sync break” jellel kezdődnek. Ezután következik a bitidő pontos megállapítását biztosító „Sync field” bájt: 01010101B = 55H értékkel.

Az azonosító bájt („Ident field”) hordozza a szolga címét, az üzenet hosszát és az ezelet ellenőrző paritásbitet. A készülék címe 4 bites, azaz 16 egység lehet összekapcsolva. A küldött bájtok száma 2, 4, 8 lehet.

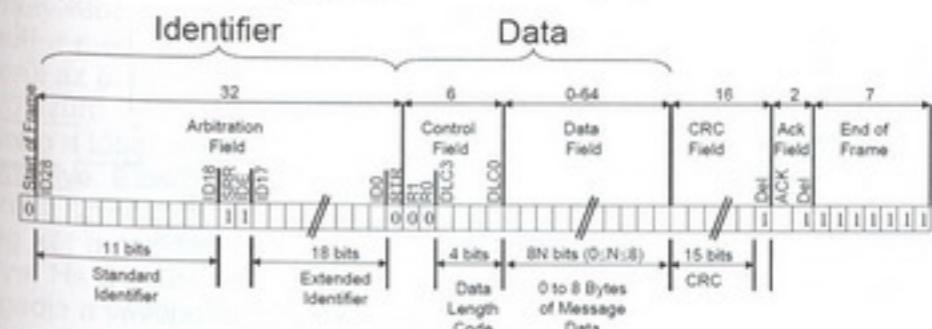
Bár a LIN protokoll nem kompatibilis a CAN protokollal, de képesek együttműködni. A CAN busz használható az autó teljes egészében a kommunikációra, amíg a LIN busz annak egy kis részét kezeli. A CAN buszon vannak a LIN csomópontok. Bár a LIN busz kialakítása diszkrét áramköri elemekből is lehetséges, a Microchip egy külön LIN buszillesztő áramkört (MCP201) fejlesztett ki erre a célra (5.14. ábra).



5.14. ábra
MCP 201 buszillesztő alkalmazása

5.6. CAN

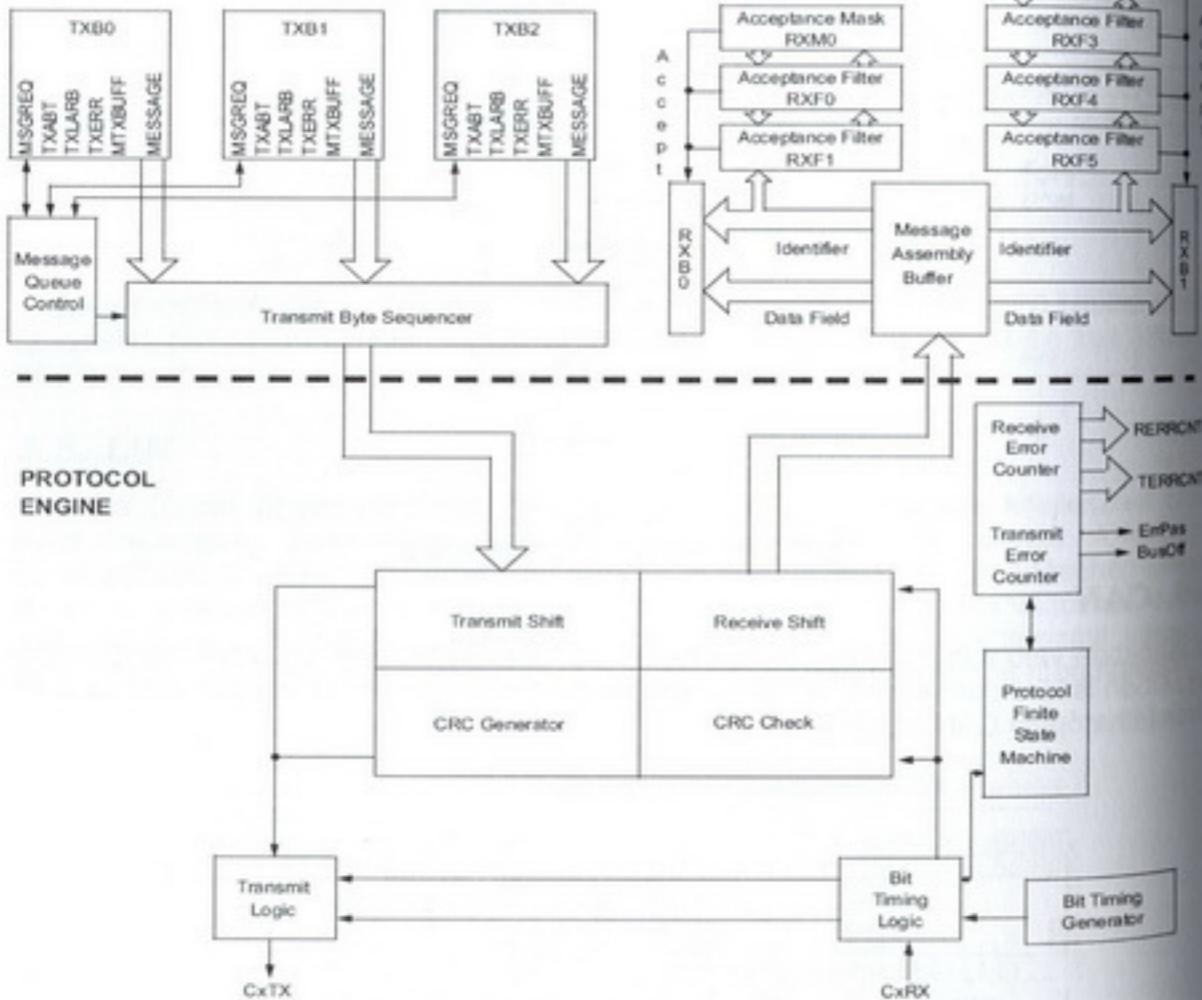
Az autókban való felhasználás során a fejlesztők felismerték, hogy a CAN busz ipari környezetben is kiválóan teljesít. Milyen tulajdonságai biztosítják ezt? Emlékeztetőül az 5.15. ábrán látható egy CAN keret felépítése.



5.15. ábra
CAN keret felépítése

- Minden adatkeret nyugtázva van Ezt az Ack mezőben lévő ACK bit biztosítja. Ugyanis az adó az ACK bitet recesszív állapotban hagyja, és a vevő feladata ilyenkor a bites domináns állapotba állítani.
- Az ipari kommunikációban az üzenetek rövidek, ezért a CAN keretben a maximálisan nyolc adatbájt elegendő.
- A kommunikációban a mester-szolga viszony mellett lehetséges, hogy egy CAN csomópont (node) olyan üzenetet küldjön egy másik node-nak, amiben felszólítja, hogy küldjön információt egy harmadik node-nak.
- Nincs címzés, hanem az üzenetek áramlanak a buszon, és a node-ok szűrői határoz zák meg, hogy értelmezi-e az üzenetet vagy nem.
- Nem kell külön beléptetni (adminisztrálni) a CAN buszon lévő node-okat, vagyis bármikor rácsatlakoztathatjuk a buszra, vagy leválaszthatjuk a buszról.
- Az üzenetek hordozzák a saját prioritásukat is, ami az esetleges ütközéskor értékelődik ki.

BUFFERS

5.16. ábra
PIC CAN blokkvázlat

A továbbiakban röviden összefoglaljuk a 16 bites PIC mikrovezérlőkben alkalmazott CAN periféria működését, aminek a blokkvázlata az 5.16. ábrán látható.

Bitidőzítés: A CAN busz sebességét a másodpercenként átvitt bitek számával adjuk meg. Például 1 MHz-es buszsebesség esetén 1 bit ideje (T_{BIT}) 1 μ s. A bitidőt a CAN protokoll négy részből állónak tekinti: *Synchronization Segment*, *Propagation Segment*, *Phase Segment 1* és *Phase Segment 2*.

A szegmensek hosszát a *Time Quanta* (TQ) elnevezéssel megadott elemi idők egész számú többszöröse adja. A bitidő 8–25 TQ lehet, és a felhasználó állíthatja be (5.17. ábra).

Miért van erre szükség? Mivel nincs külön órajel, minden csomópontnak egy digitális fázis-zárt hurkot kell használnia a buszhoz szinkronozott órajelének az előállítására. A hurok használja ezeket az időszegmenseket a pontos, szinkronizált órajelek előállítására. Részletesen erről az AN754: „*Understanding Microchip's CAN Module Bit Timing*” alkalmazási megjegyzésben olvashatunk. Ezeket az értékeket a C1CFG1 és C1CFG2 konfigurációs regiszterek tartalmának beállításával adhatjuk meg.

A CAN modul inicializálása: RESET után a CAN modul konfigurációs módban van, így nem küld és nem is fogad kereteket addig, amíg át nem állítjuk a modult normál működési módba. Ehhez elsőnek engedélyezni kell a CAN modul megszakításait. Utána a modul engedélyezhető a C1CTRL regiszter REQOP bitjeinek = 000 állításával. Mikor a regiszter OPMODE bitjei ugyanazt az értéket felveszik, a modul már kész az adás/vételre.

Üzenet vétele: A modul legfontosabb része a CAN protokollgenerátor (CAN Protocol Engine). Ez végzi az aktuális bit vételét és a hibaellenőrzést. A keret összes vett bitjét a MAB (Message Assembly Buffer) tárolóba helyezi, majd a vételi pufferek valamelyikébe küldi.

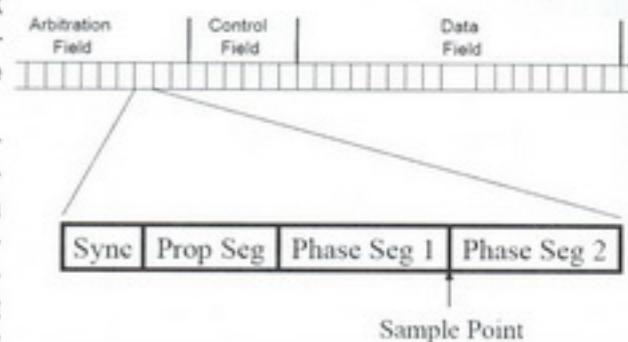
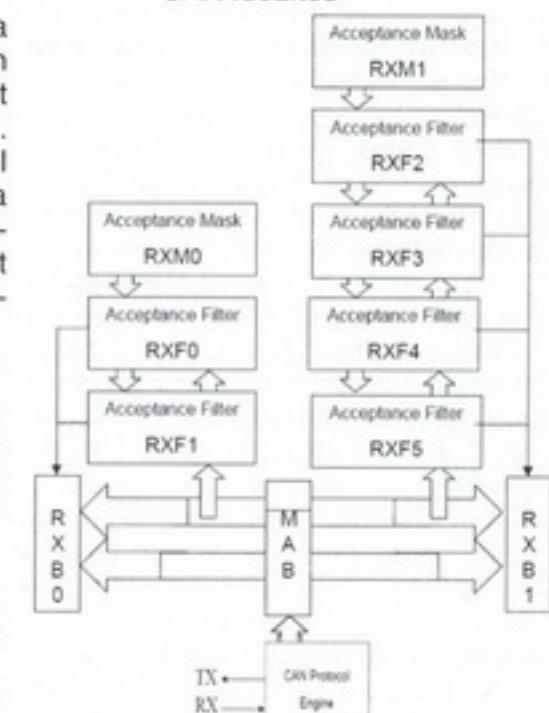
Kettő, RXB0 és RXB1 jelű vételi puffer van, ezek valamelyikébe kerül a beérkezett keret. Ha az egyik puffer még nem ürült ki, akkor a másik pufferbe kerül az érkezett keret.

Hat vevő szűrő van: RXF0 és RXF1 az RXB0 pufferhez, míg a többi négy: RXF2, RXF5, RXB1-hez van rendelve. Ezek után a CAN a keret azonosító részét a szűrőkhöz küldi kiértékelésre.

Van még két maszkregiszter: RXM0 az RXB0-hoz, az RXM1 jelű az RXB1 pufferhez van rendelve. Ha a szűrő és a maszk együttesen egyezik az üzenet azonosító bitjeivel, a modul elfogadja a vevőpufferben lévő üzenetet.

Mindkét vevőpuffer nyolc SFR regiszterből áll. Az első három 16 bites regiszter tartalmaza a vett üzenet azonosító mezőjének a bitjeit, míg a következő 4 szó magát az üzenetet. Az utolsó szóban a puffer vezérlő- és státusbitjei vannak. Mikor elfogadott üzenet

CAN Message

5.17. ábra
CAN időzítés5.18. ábra
CAN vétel

van a pufferben, a modul 1-be állítja az RXFUL (Receive Buffer Full) bitet, és generál egy megszakítást. A felhasználói program fogja törölni ezt a bitet, miután kiolvasta a puffer tartalmát. Az RXRTRRO státusbit jelzi, hogy a vett üzenet *Remote Transfer Request* típusú, amit egy másik node küldött, adatkérésként.

A C1RX1CON regiszter alsó három bitje státusbit, ami jelzi, hogy melyik szűrő okozta az üzenet elfogadását.

| FILTER/MASK TRUTH TABLE | | | |
|-------------------------|----------------|---------------------------|------------------------------------|
| Maszk bit n | Szűrő bit n | Üzenet azonosító bit n | Elfogadás vagy elutasítás bit n |
| 0 | X | X | Elfogadva |
| 1 | 0 | 0 | Elfogadva |
| 1 | 0 | 1 | Elutasítva |
| 1 | 1 | 0 | Elutasítva |
| 1 | 1 | 1 | Elfogadva |

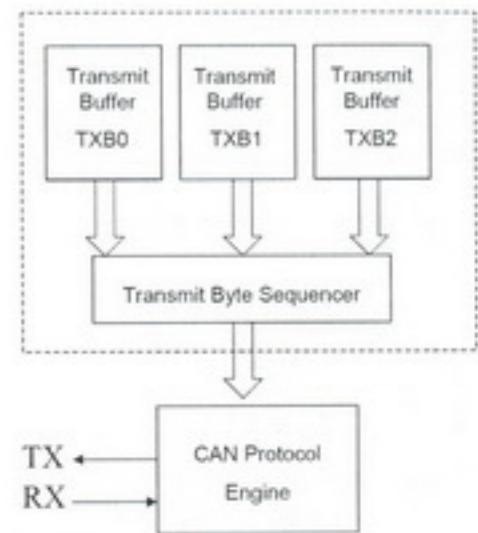
Üzenet elfogadása/elvetése (accept/reject): Vizsgáljuk meg a vételi szűrők és maszkk működését. A beérkezett keret azonosító mezőjét bitről bitre összehasonlítjuk a szűrő és a maszkregiszter azonos pozícióban lévő bitjeivel. Az előbb már leírtuk, a maszkregiszter az adott vevő pufferhez és több szűröhöz van rendelve. Az azonosító minden bitjénél az ugyanazon pozíciójú maszkbit határozza meg, hogy az a bit ellenőrzésre kerül-e. Ha a maszkbit nulla, a bitet nem vizsgáljuk. Vagyis ilyenkor a bit a szűrökön, értékeltő függetlenül, átjut. Ha a maszkbit értéke 1, akkor a bit ellenőrzésre kerül, vagyis a szűrőn való átjutáshoz az értékének egyeznie kell a szűrő megfelelő bitjével. Ha a vett keret azonosítójának minden bitje egyezik a szűrő bitjeivel (vagy a maszk miatt az értéke köztömbös), akkor az üzenet automatikusan a vevő pufferbe kerül.

A vételi szűrő három SFR regiszterből áll, amelyek felépítése hasonló a maszkregiszterek felépítéséhez. minden szűrő tartalmaz egy speciális vezérlő bitet: EXIDE – Extended Identifier Enable (kiterjesztett azonosító engedélyezése), ami azt jelzi, hogy az adott szűrő szabványos vagy kiterjesztett azonosítót fogad el. minden maszk tartalmaz egy MIDE nevű bitet. Ha MIDE bit nulla, akkor EXIDE bitet nem vesszük figyelembe, vagyis a szabványos és a kiterjesztett azonosító is szűrhető. Ha MIDE bit = 1, akkor az EXIDE bit határozza meg minden szűrőnél, hogy a szürés egyszerű vagy kiterjesztett azonosítóra történjen.

Üzenet adása: A CAN modulnak három adó pufferre van: TXB0..TXB2. Egyszerre mindig csak egy pufferból küldhetünk keretet. Adáskor a puffer bitjei sorban a CAN állapot-géphez kerülnek, ami kiküldi a buszra a biteket, hibaellenőrzés mellett. Az alkalmazás előállítja az üzenet azonosítóját, és az adó puffer első három 16 bites regiszterébe helyezi. Az adópuffer vezérlőregiszterének TXIDE bitjével jelezük, hogy szabványos vagy kiterjesztett azonosítót használunk. Az adópuffer további regisztereibe kerül a max. 8 bájtos küldendő adat. Az adatbájtok száma a keretet tartalmazó puffer DLC (Data Length Code) mezőjébe kerül.

Az adó küldhet egy speciális keretet, aminek a neve: RTP (Remote Transfer Request). Ez a keret egy másik node-ot szólít fel, hogy adatot küldjön vissza egy másik adatkeretben. Ez az üzenet TXRTR bit 1-be állításával küldhető el, és a keretben nincsenek adatbájtok.

Ha az üzenet össze van állítva az adópufferben, az alkalmazás a TXREQ (Transmit Request) bit 1-re állításával kérheti elküldését. Ez csak kérés, a tényleges átvitel a buszfoglaltság figyelembevételevel történik. Mikor a modul elküldte a keretet, a TXREQ bitet automatikusan törli, és egy CAN megszakítást generál. Ezután kezdeményezhető egy újabb adás. Mivel három adópuffer van, egyszerre akár három üzenetet is elő lehet készíteni adásra. Mikor a busz szabadá válik, dönten kell a küldési sorrendről, amit a felhasználó által megadott adópuffer-prioritás határoz meg. Az adó prioritás bitjei az adást vezérlő regiszterben vannak, és 4 prioritási szint választható. 11 bitpáros a legnagyobb, 00 a legalacsonyabb prioritást jelöli. Ha két pufferhez azonos prioritást rendelünk, akkor a nagyobb sorszámban lévőt küldjük először.



5.19. ábra
CAN adás

CAN megszakításkezelés: minden CAN modul csupán egyetlen megszakítást generál, és egyetlen megszakítási címe (vektora) van. Azonban számos esemény következménye lehet a megszakítás.

8 elsődleges esemény generálhat megszakítást. Ezek mindegyikének saját IE engedélyező és IR kérő bitje van. Ha bármelyik IR bit értéke 1, és a hozzá tartozó IE bitet 1-be állítottuk, akkor CAN megszakítás következik be. Hasonlóan más megszakításhoz, az IR bit akkor is 1-be állhat, ha a hozzá tartozó IE bit értéke 0. Melyek a megszakítást kiváltó események?

- A két vevő puffer bármelyikébe üzenet érkezett.
- Bármelyik adópufferből sikeres üzenetküldés történt.
- Ébresztő megszakítás jön létre, ha a processzor SLEEP állapotban van, és az engedélyezett CAN modul aktivitást érzékel a CAN buszon.
- Ha a vevő egy érvénytelen üzenetet vesz, akkor egy „érvénytelen üzenet” megszakítást generál.

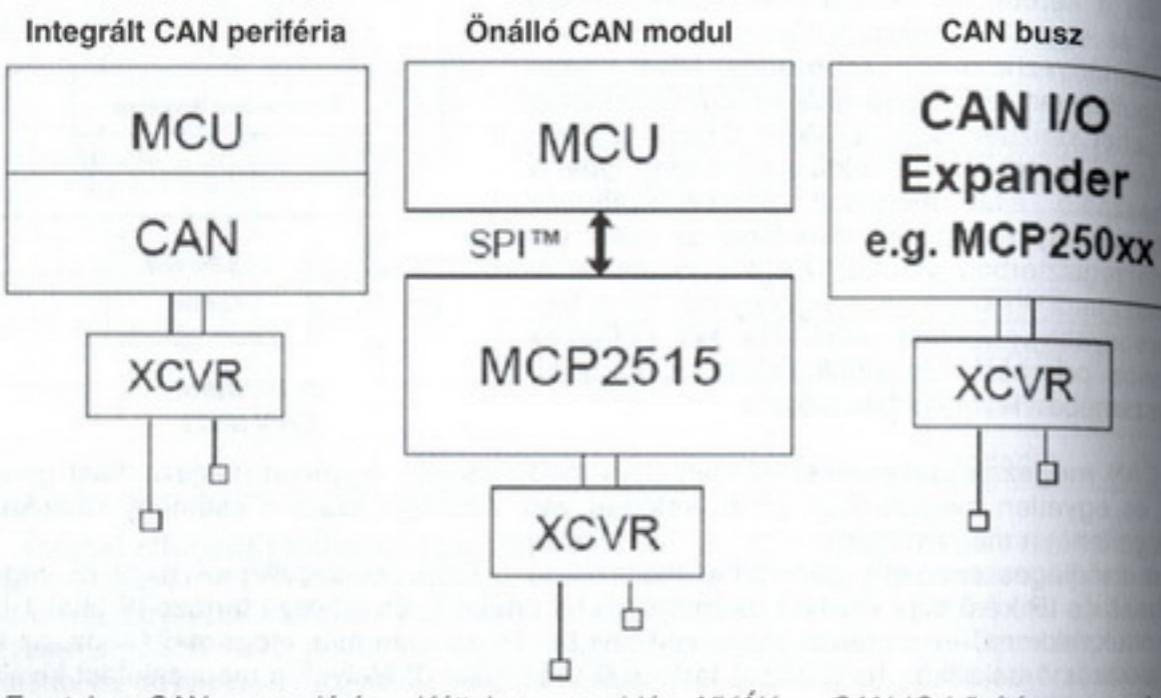
A végére maradt a hibamegszakítás. Ez a megszakítás generálódik, ha a nyolc lehetséges hibaok valamelyike létrejön. Azonosítását a hibaokokhoz rendelt státusbit 1 állapotának ellenőrzése teszi lehetővé. A nyolc lehetséges hiba státusbitjei a C1INTF megszakítást jelző regiszter felső bájtjában vannak. A hibabitek csak az adott hibafeltétel megszüntetése után törölhetők.

- Vevő puffer 0 túlcordult – RX0OVR
- Vevő puffer 1 túlcordult – RX1OVR
- Adó buszhiba figyelmeztetés – TXWAR
- Adó hiba passzív állapotban van – TXBP
- Adó kikapcsolt busz (bus off) állapotban van – TXBO
- Vevő buszhiba figyelmeztetés – RXWAR
- Vevő hiba passzív állapotban van – RXBP
- Adó vagy vevő buszhiba figyelmeztetés – EWARN

A CAN modul két számlálóban számolja a küldési és fogadási hibákat. Ha akár az adási, akár a vételi hibák száma eléri a 96-ot, a buszhiba figyelmeztetés bit 1-be áll. Ezután eléri a 128-at, akkor a hiba passzív bit 1-be áll. Ha eléri a 255-öt, akkor a bus off bit értéke is 1 lesz.

Az EWARN bitet az RXWAR és TXWAR bitek állítják.

Jelenleg a Microchip háromfajta megoldást kínál a CAN busz használatára: 8, 16 és 32 bites mikrovezérlőbe integrált perifériaként, önálló, a mikrovezérlővel SPI interfészen keresztül kommunikáló CAN modulként, illetve a CAN buszra csatlakoztatható CAN I/O modulként. Az alkalmazó döntheti el, hogy az igények, követelmények, előnyök és hátrányok figyelembevételével melyik megoldást választja.



Egy tok, a CAN gyors elérése, MCU nem biztos, hogy ideális a teljes alkalmazás-hoz, megbízhatósága nagy.

Kéttokos megoldás, NYÁK-méret növekszik, MCU ideálisra választható, rugalmasan bővíthető (utólag is).

CAN IO bővítésre, nem kell MCU-hoz illeszteni. Korlátozott, nem bővíthető képességek.

A 16 bites mikrovezérlök CAN modulját mutattuk be, ami természetesen hasonló a 8 bites mikrovezérlökben használt CAN modulokhoz, de ott 8 bites regisztereket kellett a működéshez használni. A következő táblázatban a Microchip CAN busszal kapcsolatos alkalmazási megjegyzéseit foglaltuk össze, ami természetesen folyamatosan bővül...

| | |
|-------|--|
| AN212 | Smart Sensor CAN Node Using the MCP2510 and PIC16F876 |
| AN215 | A Simple CAN Node Using the MCP2510 and PIC12C67X |
| AN225 | A CAN Physical Layer Discussion |
| AN247 | A CAN Bootloader for PIC18F CAN Microcontrollers |
| AN713 | An Introduction to the CAN Protocol |
| AN730 | ORC Generating and Checking |
| AN733 | Using the MCP2510 CAN Developer's Kit |
| AN734 | Using the PICmicro SSP for Slave I2C™ Communications |
| AN735 | Using the PICmicro MSSP Module for I2C™ Communications |
| AN736 | An I2C™ Network Protocol for Environmental Monitoring |
| AN737 | Using Digital Potentiometers to Design Low Pass Adjustable Filters |
| AN738 | PIC18C CAN Routines in C |
| AN739 | An In-Depth Look at the MCP2510 |
| AN754 | Understanding Microchip's CAN Module Bit Timing |
| AN816 | A CAN System Using Multiple MCP25025 I/O Expanders |

| | |
|-------|---|
| AN819 | Implementing Bootloader Firmware |
| AN853 | PIC18XXX8 CAN Driver with Prioritized Transmit Buffer |
| AN872 | Upgrading from the MCP2510 to the MCP2515 |
| AN873 | Using the MCP2515 CAN Developer's Kit |
| AN877 | DeviceNet™ Group 2 Slave Firmware for PIC18FXX8 |

5.7. USB

Az előző kiadásban már összefoglaltuk az USB kommunikáció alapjait. Megjelent az USB3 szabvány, amely már 4,8 Gbit/s sebességgel képes működni. Sok előnye van: egy interfész használható számos alkalmazás számára, a konfigurációja (címkiosztás, kommunikációs igények) automatikus, nem kell külső tápegység, bekapcsolt állapotban is csatlakoztatható, a felhasználónak nem kell semmit sem beállítania a csatlakozáskor.

A kommunikációt a PC alaplapján elhelyezett **USB hoszt** vezérli: kezeli és vezéri a buszműveleteket, minden adatcsomag átvitelét kezdeményezi, kezelve az adatforgalmat és a hibákat. A felcsatlakoztatott USB eszközöknek címeket foglal, valamint a megfelelő készülékosztály meghajtóprogramjához rendeli az eszközöket, észleli az eszközök fel-, illetve lecsatlakozását. Biztosítja a tápfeszültséget. Az ilyen rendszer tipikusan olyan összetett szoftveralkalmazás, amely az operációs rendszer és a mikrovezérlőben futó program együttműködésén alapul.

Az **USB perifériák** minden csak a hoszt kéréseire válaszolnak, nem tudják kezdeményezni az adatátvitelt. Érzékelik a hoszt felől indított kommunikációt, az érkező adatok hibáit, és saját fogyasztásuk szabályozására is képesek.

Az ezen felépítést használó megoldást fejlesztették tovább a merev mester-szolga működés feloldására.

Megjelent a hoszt- és perifériaműködést egyaránt tartalmazó **USB OTG** eszközcsalád. A legfontosabb a fogyasztás minimalizálása volt. Az adó-vevő meghajtó támogatja az új USB On-the-Go protokollokat: a *Host Negotiation Protocol*, illetve a *Session Request Protocol*.

Az ilyen periféria/hoszt eszközök sebessége 12 Mbit/s (*full speed*). Csak egyetlen Mini-AB aljzatot tartalmaz, ami képes akár Mini-A, akár Mini-B csatlakozót fogadni, és A típusú eszközöként (mesterként) képes legalább 8 mA-es áramot biztosítani.

A **PIC18F13K50** típusú mikrovezérlő számos előnyös tulajdonsággal rendelkezik. Kis lábszámú, olcsó, kiszolgálja a 12 Mbit/s-os USB 2.0 kapcsolatot, és a nanowatt technológiát hasznosítva a kommunikációhoz igazítja a fogyasztását és az órajelét. Telepes működtetéshez optimális.

Beépítették az mTouch (TM) érintésérzékelést megvalósító lehetőséget. 10 bites ADC, motorvezérlő PWM jelek, soros kommunikáció szerepel a perifériák között. A minden 20 lábú tok perifériáit az 5.20. ábrán foglaltuk össze.

| Pin | I/O | Analog | Comparator | Reference | ECCP | EUSART | MSSP | Timers | Interrupts | Pull-up | USB | Basic |
|-----|-----|--------|------------|-----------|------|--------|------|--------|------------|---------|-----|------------------------|
| 19 | RA0 | | | | | | | | IOC | D+ | PGD | |
| 18 | RA1 | | | | | | | | IOC | D- | PGD | |
| 4 | RA3 | | | | | | | | | | | MCLR / V _{PP} |
| 3 | RA4 | AN3 | | | | | | | IOC | Y | | OSC2/ CLKOUT |

| Pin | I/O | Analog | Comparator | Reference | ECCP | EUSART | MSSP | Timers | Interrupts | Pull-up | USB | Basic |
|-----|-----|--------|------------|-------------------|--------------|-----------|-------------------|------------------|------------|-----------------|-----|----------------|
| 2 | RA5 | | | | | | | | IOC | Y | | OSC1/ CLKIN |
| 13 | RB4 | AN10 | | | | | SDI/S DA | | IOC | Y | | |
| 12 | RB5 | AN11 | | | | RX/ DT | | | IOC | Y | | |
| 11 | RB6 | | | | | | SCL/S CK | | IOC | Y | | |
| 10 | RB7 | | | | | | TX/ CK | | IOC | Y | | |
| 16 | RC0 | AN4 | C12IN+ | V _{REF+} | | | | | INT0 | | | |
| 15 | RC1 | AN5 | C12IN1- | V _{REF-} | | | | | INT1 | | | |
| 14 | RC2 | AN6 | C12IN2- | CV _{REF} | P1D | | | | INT2 | | | |
| 7 | RC3 | AN7 | C12IN3- | | P1C | | | | | | | PGM |
| 6 | RC4 | | C12OUT | | P1B | | | | | | | SRQ |
| 5 | RC5 | | | | CCP1/ P1A | | T1CKI | | | | | |
| 8 | RC6 | AN8 | | | | SS | T13CKI/T 1OSCI | | | | | |
| 9 | RC7 | AN9 | | | | SDO | T1OSCO | | | | | |
| 17 | | | | | | | | V _{USB} | | | | |
| 1 | | | | | | | | | | V _{D0} | | |
| 20 | | | | | | | | | | V _{Gs} | | |

5.20. ábra
PIC 18F13K50 funkciók, lábak

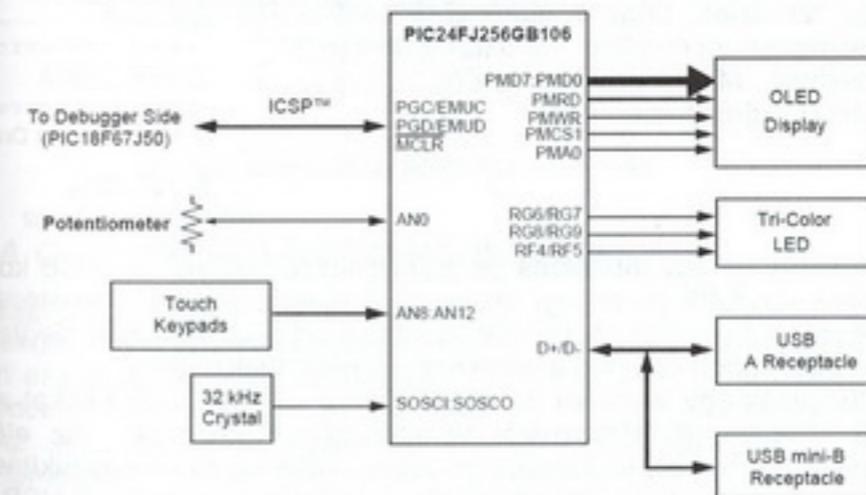
Kissé részletesebben:

- PIC18 CPU, 8 vagy 16 kb ájt programmemória, 512 vagy 768 bájtos adatmemória, ami tartalmazza az USB által is elérhető 256 bájtos duálport RAM-ot;
- Max. 131 mp hosszúságú WDT;
- Az USB 2.0-s 12 Mbit/s kapcsolata (tudja a kis sebességet is);
- Támogatja a vezérlő (control), megszakítás (interrupt), időkorlátos (isochronous) és nagy adatmennyiségek (bulk) USB adatátvitelket;
- Max. 16 végpont lehet (vagy max. 8 kétirányú);
- USB vezérlőhöz való fizikai kapcsolódás érzékelése;
- 256 bájt két irányból elérhető (Dual Access) RAM;
- Számos órajel-lehetőség, belsőleg előállítható az USB-hez szükséges 48 MHz;
- Kettős Capture/Compare/PWM periféria, max. 5 PWM kimenettel
 - Programozható holtidő, automatikus lekapcsolás/újraindítás;
- Mester I2C/SPI kommunikáció
 - Az SPI minden üzemmódját támogatja
 - I2C™ mester és szolga módjai, címmaszkolással
- Továbbfejlesztett címzhető USART modul
 - RS-485, RS-232, és LIN 2.0 támogatás

- 10 bites, 9 csatornás AD modul
- Automatikus mintavétel/konvertálás;
- SLEEP módban is működhet;
- mTouch (TM) kompatibilis analóg komparátorok SR tárolóval;
- Csúcstól csúcsig működés;
- Független bemeneti multiplexelés;
- Erős GPIO 25 mA nyelő/forrás áram;
- Programozható belső feszültségreferencia
- Komparátorral vagy önálló ADC-ként használható;
- Rugalmas működési körülmények
- 1,8–5,6 V tápfeszültség;
- Programban állítható Brown-out Reset;
- Négy fogyasztáskezelő működési mód;
- Kétsebességű indulási mód.

5.7.1. MPLAB Starter Kit PIC24-es mikrovezérlőkhöz

Ez az eszköz azért került az USB-vel foglalkozó fejezetbe, mert számos része demonstrálja ennek a perifériának a hasznosságát. A PIC24F Starter Kitet (DM240011) a PIC24F család gyakorlati megismerésére fejlesztették ki. Ez az olcsó eszköz beépítve tartalmaz egy hibavadász (debugger) és programozó áramkört, az USB-hosztfunkció megvalósítását, valamint perifériacsatlakozásokat: háromszínű LED-hez, kapacitív érintőgombokhoz és egy OLED kijelzőhöz. Blokkvázlata az 5.21. ábrán látható.



5.21. ábra
PIC24F Starter kit blokkvázlata

- Egy menüvezérelt demonstrációs szoftver segítségével több megvalósított feladatot lehet futtatni. Részletesebben:
- Párhuzamos mesterporttal működtetett menüt megjelenítő kijelzőt;
 - Kapacitív érintésérzékelőket a töltésiidő-mérő egységgel megvalósítva (CTMU);
 - Az idő és dátum kezelését biztosító RTCC egység használata;
 - Háromszínű (RGB) LED vezérlés három PWM jelkel, és a lábak – jelek – áthelyezhetőségét biztosító Peripheral Pin Select – PPS – megoldás;
 - USB pendrive-illesztést a beépített hoszt vezérlésével;
 - Valós idejű adatábrázolást az analóg-digitál átalakító (ADC) jelének felhasználásával, multitasking alkalmazásával;
 - Valós idejű multitaskinggal megvalósított adatgyűjtést az USB hoszt (mester) funkciójának felhasználásával a pendrive-ra.

A beépített hibavadász/programozó áramkör felhasználásával a kártyán saját fejlesztésekkel is futtathatunk a **PIC24FJ256GB106** programozásával. A PC-hez kapcsolódó USB-biztosítja a kártya áramellátását is. A Starter Kithez egy ingyenes C fordító is tartozik.

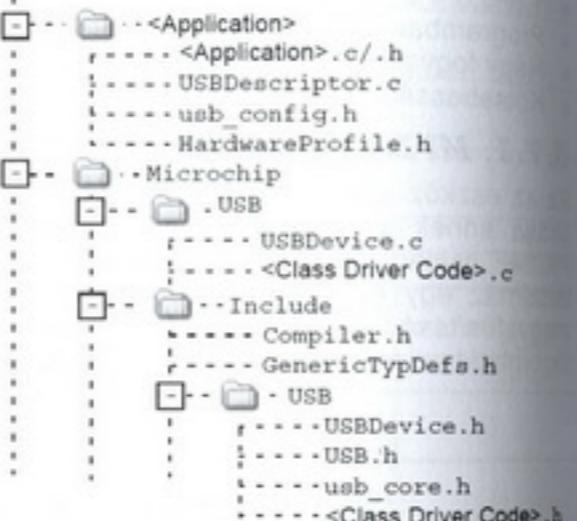
A PIC24F Starter kit „lelke” egy **PIC24F256GB106** típusú mikrovezérlő, és csak kevés járulékos hardverelemet kellett a panelen elhelyezni. minden demóprogram a tok flash memoriájában van, és a demók még tartalmazzák a Microchip szoftverkönyvtárak, a **Microchip USB Stack Library**, a **Microchip Memory Disk Drive File System** és a **Microchip Graphics Library** bizonyos részeit is.

5.7.2. Keretprogram az USB eszközök kezelésére

Az eredeti elnevezés, a **Microchip USB Device**

Firmware Framework egy könyvtárrendszer, amelynek segítségével új USB alkalmazás hozható létre. Úgy is felfogható, mint egy tervezési feladat megoldása, amely tartalmazza a szükséges programkódot, és lényegében kerete a felhasználói programkódnak. A teljes kódrendszer egyetlen gyökérkönyvtárban található, számos alkönyvtárba szervezve.

A keretprogram a Microchip fejlesztőszökei legutolsó verziójának felhasználásával készült, és úgy terveztek, hogy a fájlok a Microchip könyvtárban semmiféle változtatást nem igényelnek. minden módosítandó fájl az **<Application>** könyvtárban van.

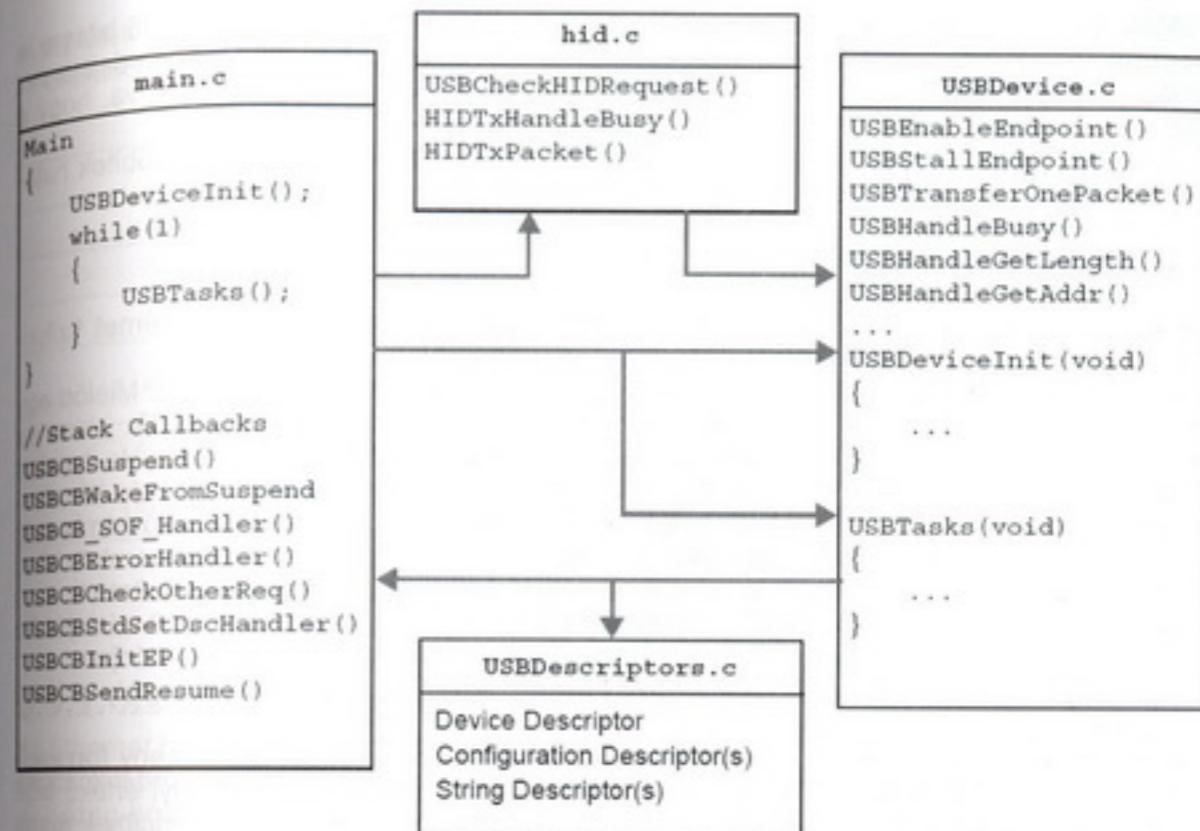


5.22. ábra
Könyvtárstruktúra

Az USB keretprogram egy moduláris programrendszert biztosít az USB kommunikáció megvalósításához. Az 5.23. ábrán egy tipikus USB programfelépítés látható. A teljes feladatot az időszeleteket használó kooperatív multitasking kívánalmainak megfelelően programozták, vagyis a program nem fog a várakozások miatt blokkolódni.

A **main()** függvény egy végtelen hurok, amiben a különféle feladatokat szolgáljuk ki. Ezek lehetnek USB vagy felhasználói tevékenységek (taszkok). Az előbbieket az **USBTasks()** rész kezeli, amely kiszolgálja az összes USB-hardvermegszakítást.

A keretprogram használatát leíró anyag részletesen tartalmazza az USB alkalmazás megvalósításához szükséges segítséget.



5.23.
Tipikus USB program felépítése

5.8. DATA CONVERTER INTERFACE (DCI)

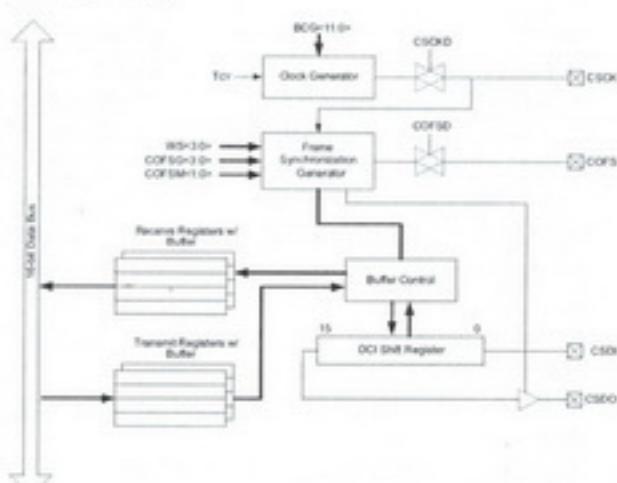
16/24 32/32 Ez a modul a digitális jel-feldolgozást segíti, mivel egyszerű illesztést biztosít olyan eszközökhöz, mint az audiódolók/dekódolók (kodekek), A/D és D/A átalakítók. Az itt használatos következő illesztéseket támogatja:

- Egy- vagy többcsatornás keretezett szinkron soros átvitel (*Framed Synchronous Serial Transfer*);
- Inter-IC Sound (I₂S) illesztés;
- AC-Link Compliant mód.

Sok audioalkalmazásban használatos kodek támogatja a mintavételei frekvenciát 8 kHz és 48 kHz között, és a fent felsorolt illesztések valamelyikét használja.

DCI automatikusan kezeli a fenti protokollok időzítéseit.

A CPU függetlenül működhet, amíg a megfelelő számú adat elküldésre vagy beolvasásra kerül.



5.24. ábra
DCI blokkvázlata

Max. négy adatszót lehet átvinni a CPU megszakításai között. Az adatszó 16 bitesre is programozható, ami dsPIC30F adatszélességének felel meg. Azonban sok kódék ennélfogva hosszabb adatokkal dolgozik, de a DCI ezt is kezeli, olyan módon programozva, hogy a hosszú adatokat 16 bites időrésekben viszi át.

A DCI az adatkeretben max. 16 időrést támogat, minden időrésben vezérlőbítek határozzák meg, hogy a DCI éppen küld/fogad az időrésben.

5.9. ETHERNET

A Microchip kommunikációs megoldásai között nagy jelentősége van az internet fizikai adatkapcsolati rétegét megvalósító Ethernet-alapú adatátvitelnek.

Az üzenetszóráson alapuló Ethernet közeghozzáférés a versenyen alapul. Mielőtt egy állomás adni akar, belehallgat a csatornába. Ha a kábelen adatforgalom zajlik, akkor az állomás addig vár, amíg az meg nem szűnik, máskülönben azonnal adni kezd. Ha ilyenkor két vagy több állomás várakozik, akkor ezek egyszerre kezdenek el adni, ütközés következik be. minden ütköző állomásnak be kell fejeznie az adását, majd véletlen ideig várnia kell, annak leteltével ismét próbálkoznia kell a keretküldéssel. minden Ethernet keret 8 bites oktetekből, lényegében bájtokból épül fel, amit az IEEE 802.3 szabványban rögzítettek.

- Minden keret egy **7 bájtos előtaggal** (Preamble) kezdődik, amely 10101010 mintájú. Ez lehetőséget biztosít a vevő belső órájának, hogy az adó órájához szinkronizálódjon.
- Ezután következik a **keretkezdet** (Start Frame Delimiter – SFD) bájt, amely a keret kezdetét jelöli ki az 10101011 mintával.
- A keret két címet tartalmaz, egy **célcímet** (Destination Address – DA) és egy **forráscímet** (Source Address – SA). Ezek hossza 6 bájt. A célcím legfelső helyi értékű bitje (I/G) közönséges címek esetén 0, csoportcímek esetén 1 értékű. A csoportcímek teszik lehetővé több állomás egyetlen címmel való megcímzését. Amikor egy keret csoportcím tartalmaz célcímként, akkor a keretet a csoport minden tagja veszi. Az állomások egy meghatározott csoportjának való keretküldést többes küldésnek (multicast) nevezik. A célcímben csupa 1-est tartalmazó keretet az összes állomás veszi. Ez az üzenetszórás (broadcast). A címzésnél érdekes a legmagasabb helyi értékű bit mellett 46. bit (UL) használata. Ez a bit a helyi és a globális címeket különbözteti meg. A helyi címeket a hálózatmenedzserek jelölik ki, és a helyi hálózaton kívül nincs jelentőségük. A globális címeket ellenben az IEEE jelöli ki azért, hogy ne fordulhasson elő két, a világban azonos globális cím. Mivel $48 - 2 = 46$ bit áll rendelkezésre, ezért megközelítőleg $7 \cdot 10^{13}$ globális cím létezik. Az alapgondolat az, hogy 2+46 bitet használva már a világ bármely két állomása megcímelheti egymást. Ezt a 6*8 bitet megegyezés szerint hexadecimális alakban, bájtonként kettőspontokkal elválasztva adják meg, például: 3A:12:17:0:56:34. A hatbájtos fizikai (MAC) cím felső három bájtját a XEROX cég osztja ki az Ethernet kártya és áramkörgyártók között, míg az alsó három bájtot az adott gyártó adja, az egyediség biztosítása miatt.
- Hossz/Típus (Length/Type):** Ha a mező értéke ≤ 1500 (decimálisan), akkor az adatmezőben (payload) található adatbájtok számát adja meg. Ha ≥ 1536 , akkor az adat típusát (milyen protokollcsomag) jelzi: IPv4 = 0800h, IPv6 = 86DDh, ARP = 0806h, RARP = 8035h.
- Adatmező (Payload):** Az érvényes keretek és a szemét megkülönböztetése érdekében a 802.3 szabvány szerint egy érvényes keretnek legalább 64 bájt hosszúnak kell lennie, a célcímtől kezdődően, az ellenőrző összeget is beleértve. Ha tehát egy keret adatréseze 46 bájtnál rövidebb, akkor kitöltő
- Töltelék (Pad)** mezőt kell használni a minimális kerethossz eléréséhez. A minimális kerethosszúság alkalmazásának másik oka az, hogy egy rövid keret küldését egy állomás még azelőtt befejezhetné, mielőtt a keret első bitje elérne a kábel legtávolabbi végét, ahol is az egy másik kerettel ütközhet.

5. fejezet: Kommunikációs perifériák

- Az utolsó mező az **ellenőrző összeg** (checksum). Az ellenőrző összeg algoritmus a ciklikus redundancia-ellenőrzésen (32 bites CRC) alapul.

10/100 IEEE 802.3™ Frame

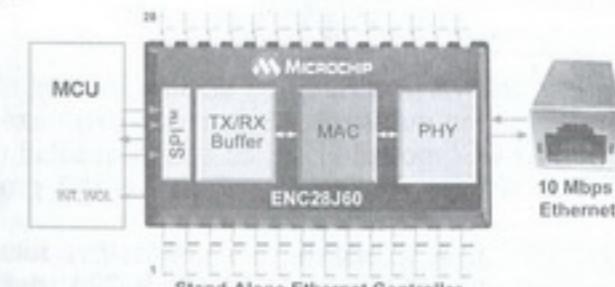
| | |
|--------------------------------|-----------------------------|
| 7 octets | Preamble |
| 1 octet | Start Frame Delimiter (SFD) |
| 6 octets | Destination Address (DA) |
| 6 octets | Source Address (SA) |
| 2 octets | Length (≤ 1500) |
| 46 octets to 1500 octets | Type (≥ 1536) |
| 4 octets | Client Data (Payload) |
| | Pad (if necessary) |
| | Frame Check Sequence (FCS) |

5.25. ábra
Ethernet keret felépítése

5.9.1. Az Ethernet sikere

Az Ethernet hálózatok hatalmas karrier futottak be: ma a világban legjobban elterjedt adatkapcsolati protokoll. Vajon mivel érte el a nagy sikerét? Ennek több oka van:

- Közeg-hozzáférési algoritmus a jól alkalmazkodik a hálózatot használó emberi tevékenységekhez: a felhasználók nem folyamatosan használják a hálózatot, hanem időben elosztva van hálózati forgalom. A számítógépes illesztőkártyái is olcsók.
- A jelek kódolásánál hatalmas előny, hogy minden bit önmagát szinkronizálja a jel fel- és lefutó eleivel (Manchester-kódolás).
- Technikai fejlődésével az adatátviteli sebességet folyamatosan lehetett növelni (1–10–100–1000... Mbit/s).
- A protokoll a viszonylag egyszerű felépítésű adatkeretek továbbítása során nem használ nyugtázást, de ez nem baj, mert a fizikai közegben való továbbítás során, az ütközésekben kívül nem nagyon keletkeznek hibák.
- Az Ethernet keretekkel kommunikáló gépet könnyen lehet a rendszerbe illeszteni: egyszerűen csupán csatlakoztatni kell a fizikai hálózathoz.
- A fizikai közeg kezdetben kialakított színtopológiája miatt az átvitel csak félduplex lehetett. A csavart érpáras, külön adó és vevő vonalat tartalmazó megoldások már teljes duplex átvitelt tettek lehetővé, megduplázva ezzel a rendszer áteresztőképességét.
- A rádiós hálózatok adatkapcsolati protokolljának szinte kínálkozik az üzenetszórást alkalmazó Ethernet.



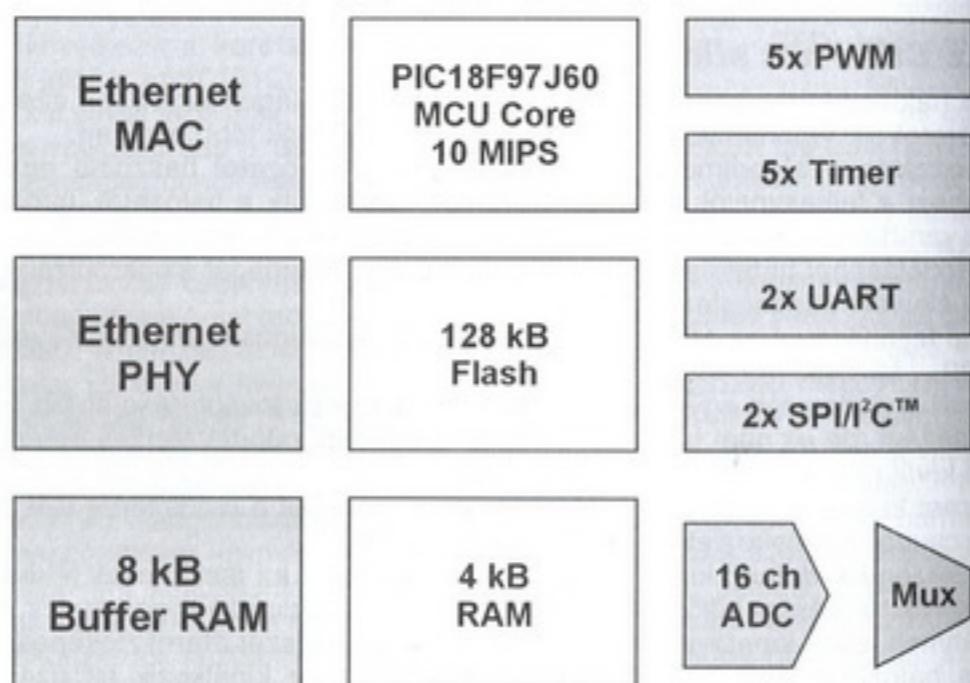
5.26. ábra
Ethernet tok

- Az integrált áramköri technika lehetővé tette, hogy a teljes keretkezelést egyetlen integrált áramköri lapka végezze (Ethernet chip). A mai hálózati kártyákon ez az egyetlen aktív elem, és ezért a PC-k alaplapjára is könnyen lehet integrálni.
- A hibájának tartott ütközések gyakoriságát lecsökkentették, és ezért a hálózat átviteli teljesítményét (*throughput*) megnövelték az Ethernet hálózatban elhelyezett intelligens kapcsolóeszközök (*bridge, switch*).
- Bár elvileg üzenetszórásos hálózaton működik, de képes pont–pont jellegű kapcsolat kiszolgálására is. A kábel- és ADSL modemek gyakran használnak a számítógéppel való adatcserére Ethernet kereteket. (Sebesség van, ütközés nincs!)

5.9.2. Microchip-megoldások

A Microchip első megoldása egy 10 Mbit/s sebességgel működő **ENC28J60** Ethernet chip kifejlesztése volt, ami a PC-s hálózati kártyák számára készített párhuzamos buszos PC-illesztés helyett csupán négy vonalat igénylő SPI buszos csatolást használ a mikrovezérlővel való kommunikációra.

Következő lépésként ez már mint beépített periféria jelent meg először a 8 bites PIC18-as termékvonalban **PIC18F97J60** néven. Ma már ez a periféria, különféle más perifériák társaságában, a 16 és 32 bit adatszélességű PIC mikrovezérlökben is megtalálható (5.27. ábra).

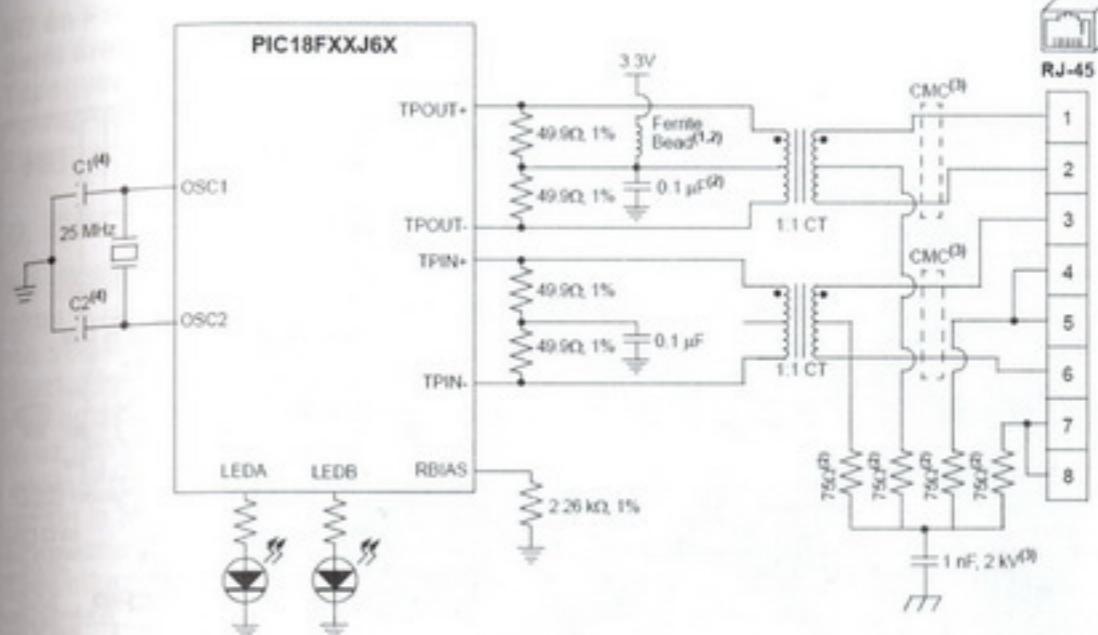


5.27. ábra
PIC18F97J60

Az Ethernet-kapcsolat – a szabványos RJ45-ös csatlakozó mellett – illesztő transzformátor használatát is igényli, amit ma már az RJ45-ös csatlakozóházába integrálnak.

Mivel az Ethernet csupán az OSI modell fizikai és adatkapcsolati réteget valósítja meg, meg kell valósítani a felsőbb rétegek funkcióit is. Ez a TCP/IP protokoll PIC-re történő megírását jelenti.

A Microchip ezt megvalósította, és szabadon hozzáférhetővé tette a TCP/IP protokollrendszer megvalósító programot, ami elérhető PIC18, PIC24, dsPIC és PIC32 mikrovezérlőkre. A program a TCP/IP rendszer protokolljait valósítja meg.

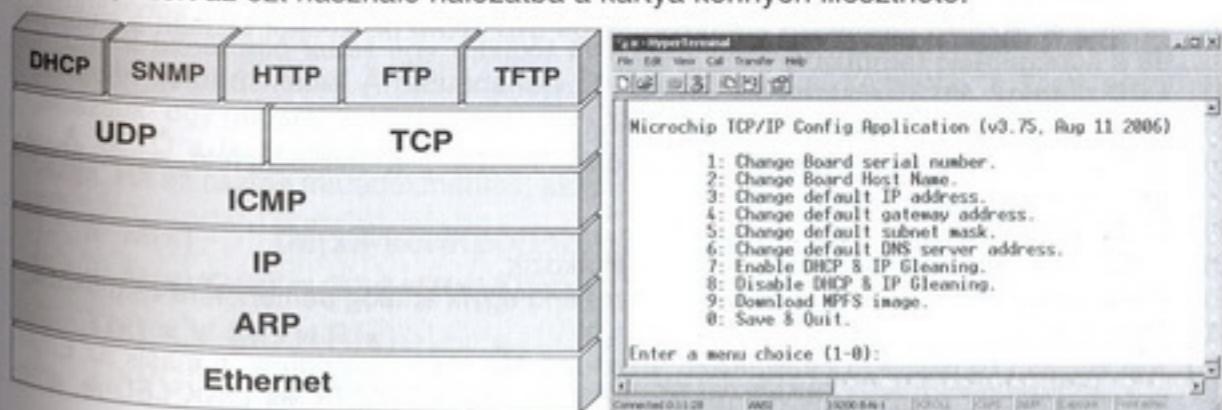


5.28. ábra
Ethernet áramköri kiegészítés

Megvalósított protokollok: **ARP, IP, ICMP, UDP, TCP, DHCP, SNMP, HTTP, FTP, TFTP**. Biztonsági réteg: *Secure Sockets Layer (SSL)*, *NetBIOS Name Service*, *DNS – Domain Name System*. A programrendszer C nyelven íródott, és a felhasználó által megadott paraméterekkel újra fordítva illeszthető az adott rendszerbe.

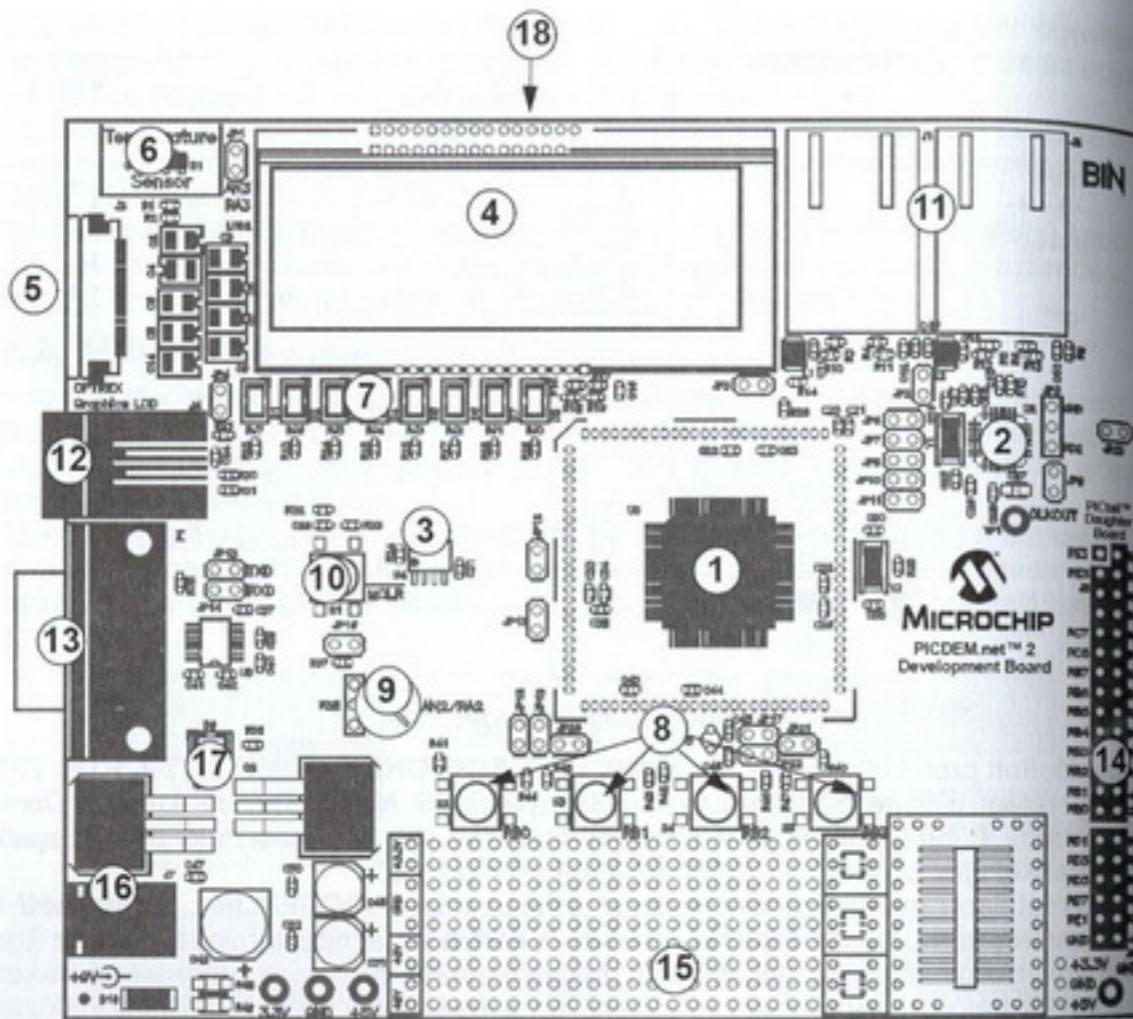
Az internetalapú mikrovezérlő-fejlesztést a Microchip a **PICDEM.net. 2 Internet/Ethernet Development Board** eszközzel segíti. A kártya tartalmazza mindeneket az áramköröket, amivel egy internetes alkalmazás megvalósítható. Képes az Ethernet csatlakozókon keresztül hálózati kapcsolatot létesíteni, és a beépített perifériák jeleit a hálózaton keresztül más helyre eljuttatni.

A kártya egy demóalkalmazással van felprogramozva, aminek legfontosabb jellemzői: az ingyenes Microchip TCP/IP Stack programot használja, ami tartalmaz: HTTP szervert, FTP szervert, DHCP klienst, IP Gleaninget. A felhasználó által konfigurálható, egy RS-232 soros vonalon keresztül. A kártyán lévő soros EEPROM-ban lehet tárolni a konfigurációs adatokat és a demó weboldalakat. minden gyárilag programozott kártya ismeri a DHCP protokollt, ezért az ezt használó hálózatba a kártya könnyen illeszthető.



5.29. ábra
MCHIP TCP/IP

5.30. ábra
Demóprogram indítási képernyője



5.31. ábra

PICDEM.net. 2 Internet/Ethernet Development Board

- 1) **Mikrovezérlő:** PIC18F97J60 mikrovezérlő. Órajele: 25 MHz. Előreprogramozva egy demó alkalmazással.
- 2) **Ethernet vezérlő:** A PIC18F97J60 mellett egy önálló, SPI-n kommunikáló ENC28J60 Ethernet tokot is tartalmaz a kártya.
- 3) **Memória:** Az SPI interfészű 25LC256 soros EEPROM 32 kbájtot biztosít a weblapok és a konfiguráció tárolására.
- 4) **LCD display:** 2*16 karakteres kijelző.
- 5) **Opcionális külső LCD csatlakozó**
- 6) **Analóg hőmérséklet-érzékelő:** Microchip TC1047. A mikrovezérlő egyik analóg bemenetére csatlakozik.
- 7) **8 db LED:** PORTJ-re csatlakozik.
- 8) **4 db nyomógomb:** PORTB<3:0>-ra csatlakozik.
- 9) **Potenciometerek:** 10 kilohmos. A mikrovezérlő egyik analóg bemenetére csatlakozik.
- 10) **Reset gomb**
- 11) **Két RJ-45 (10 Base-T) csatlakozó:** Egyik az AM mikrovezérlőhöz, a másik az Ethernet vezérlőhöz csatlakozik. Mindegyiknek van egy ACTIVITY és LINK LED-je, amelyeken a keretek adása/vétele látható.
- 12) **ICD2 csatlakozó**
- 13) **Soros RS232 PORT DB9 csatlakozó**

- 14) I/O és PICTail, daughter board csatlakozó
- 15) Saját áramkör kialakítására
- 16) Tápegység: 5 V és 3.3 V /500 mA
- 17) Power-on LED
- 18) Ethernet címek feliratozva

5.10. CRC ELLENŐRZŐ MODUL

16/24 32/32 Az adatátvitel során az adatokat célszerű összefüggő csoportokba – keretekbe – rendezve átvinni, amibe kiegészítő információként a keretben lévő adatokból számolt valamelyen ellenőrző információt is elhelyezünk. A vevő a keretekben lévő adatból ezt újraszámítja, és egyezés esetén az adatokat elfogadja. A legegyszerűbb megoldás az adatbájtokból számolt ellenőrző összeg használata, de több bit meghibásodása esetén ez látszólag helyes eredményt adhat.

A ciklikus redundancia-ellenőrzési (Cyclic Redundancy Check – CRC) eljárás ilyen hibák esetén is jól működik: egy keretnyi adatot az ADÓ és VEVŐ oldalon is ismert bitsorozattal „elosztunk”, és az osztás „maradékát” a keret részeként továbbítjuk. Az osztási elvből következik, hogy a bitsorozatnak tekinthető osztandó adatkeret bármelyik bitjének a megváltozása az osztás eredményét és a maradékot is befolyásolja. Az osztási művelet során minden osztandó keret bitsorozatát, minden közös osztó bitsorozatot 1 és 0 együtthatókból álló n-edfokú polinomnak tekintjük. Az osztó bitsorozatból képezett polinom a generátor polinom. A módszer a következő módon működik:

A küldő fél teendői:

- A küldendő adatokat egy hosszú bináris, szekvenciális adatfolyammá rendezzük (hosszú szám).
- Elvégezzük ennek az adatnak megfelelő polinom osztását egy választott osztóval (ezt nevezik **generátor polinomnak**).
- Elküldjük az eredeti adatokat és a polinomos osztás maradékát.

A vevőoldalon:

- A kapott adatokat egy hosszú bináris, szekvenciális adatfolyammá (hosszú szám) rendezzük.
- Elvégezzük e szám polinomos osztását a választott generátor polinom osztóval. Lesz egy vevőoldali maradék.
- Összehasonlítsuk a küldő oldalról kapott maradékot, és a vevőoldalon nyert eredményt. Ha a kettő egyezik, akkor az adat-sor helyes.

CRC – CYCLIC REDUNDANCY CHECK

Csoportos bithibák kezelésére alkalmas. A lényeg: az osztás eredményét több szomszédos számjegy határozza meg.

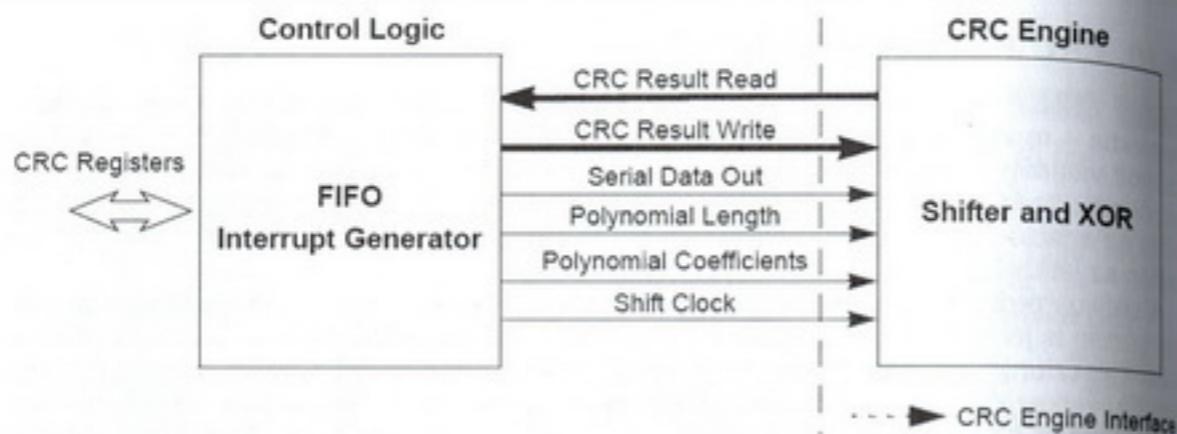
Egy k bitból álló keret egy r bitból álló redundanciával kiegészítve ($k+r$) bites blokkban kerül átvitelre. A redundancia polinom osztással képződik. A keret bitjeit „elosztjuk” egy generátor polinommal. Pl. $G(x) = x^4 + x + 1 \rightarrow 10011$

A VEVŐ a ($k+r$) bitból álló sorozatot ugyanavval a $G(x)$ generátor polinommal osztja. Ha az osztás maradékmentes, akkor feltételezhető, hogy nincs hiba.

1. $x^r M(x) \quad (M(x)=\text{keretpolinom})$
 2. $x^r M(x)/G(x) = Q(x)+R(x)/G(x) \quad R(x)=\text{maradékpolinom}$
 3. $T(x) = x^r M(x)+R(x)$
 4. $T(x)/G(x) = [x^r M(x)+R(x)]/G(x) = Q(x) + R(x)/G(x) + R(x)/G(x)$
- ALGORITMUS !**
- a „+” itt modulo 2 összeadás !!!

5.31. ábra
CRC módszer

A PIC CRC modul periféria ezt valósítja meg. A generátor polinom fokszámát és együtthatóját mi választjuk meg, mert minden polinomnak más és más a hibadetektáló képessége. A CRC meghatározás számításigényes feladat, szoftveresen táblázatok segítségével történik. A legfontosabb az a lehetőség, hogy az osztás kizáró-vagy kapukon keresztül visszacsatolt shiftregiszterek segítségével hardveresen is megvalósítható!



5.32. ábra
Programozható CRC generátor blokkvázlata

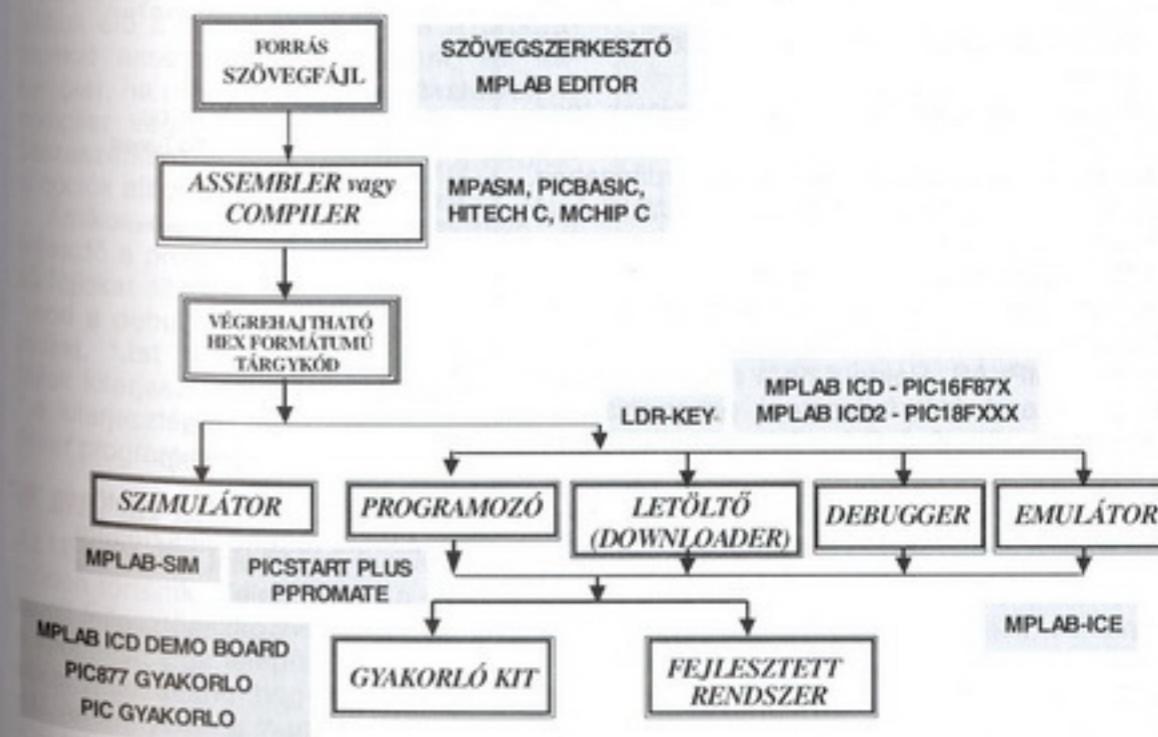
A modul legfontosabb tulajdonságai:

- a generátor polinom hossza és együtthatói változtathatók (maximum 16 bites lehet),
- megszakításos működés,
- 8 szintű 16 bites, illetve 16 szintű 8 bites FIFO puffertár az adatok tárolására.

6. PIC PROGRAMFEJLESZTŐ ESZKÖZÖK

A fejlesztés során a megírt és lefordított programot a kipróbálás céljából a mikrovezérlő programmemoriájába kell juttatni és ott futtatni. A programfejlesztésnek, amely a tesztelést is magában foglalja, több szintje lehetséges:

- A gépi kódot egy programozóval a programmemoriába írjuk, kipróbáljuk – és nem fut...
- A **programszimuláció** arra ad lehetőséget, hogy a programunk működését számítógépes környezetben – lépésenként vagy nagyobb blokkokban – ellenőrizhessük. A szimulátor mindenkor minden másik számítógépen futó program, amelyik „imitálja” a program utasításainak a végrehajtását. Az **ellenőrzés (tesztelés)** és a **hibakeresés (debugolás)** a gépi utasítások szintjén történik, a hardverjellegű, például időzítési hibák felderítése ilyen módon nem lehetséges. Előnye, hogy a hardver jelenléte nélkül is lehetséges ellenőrzés.
- **Valós idejű ellenőrzés In-Circuit-Debugger (ICD) segítségével.** Ilyenkor a mikrokontrollerekben elhelyezett töréspontkezelő áramkörrel bárhon meg tudjuk állítani a program futását és az aktuális állapotot, azaz a regiszterek tartalmát elemezve tudjuk az esetleges hibás működést felderíteni.
- **Valós idejű ellenőrzés REAL ICE emulátoron.** Itt az utasítások végrehajtása egy gyors áramkörökön felépített egységgel történik, az áramköri működés szintjén. Ez az egység egy fejen (POD) keresztül közvetlenül a fejlesztendő rendszer kontrollerének a helyére dugva működik (*In-Circuit-Emulator – ICE*).
- **Valós idejű ellenőrzés** a végleges hardveren, a gépi kódot letöltő (*bootloader*) és a program ellenőrzését biztosító **monitorprogram** segítségével.



6.1. ábra
Programfejlesztési lehetőségek

A Microchip több megoldást kínál fejlesztési célokra. A fejlesztő szoftverrendszer az ingyenes **MPLAB**. Ez képes működtetni a fejlesztést biztosító különböző eszközöket. A programjainkat assemblerben, BASIC-ben vagy C-ben megírva az MPLAB környezetben elő tudjuk állítani a futtatható tárgykódot. Ezt az **MPLAB-SIM** szimulátorral virtuálisan kipróbálhatjuk.

PICSTART programozóval a programot tokba írhatjuk. Az **ICD2**, **ICD3**, **PICKIT1**, **PICKIT2** hibakereső programozával vagy **REAL ICE** emulátor segítségével a programot akár lépésenként futtathatjuk, futás közben megállíthatjuk.

6.1. MPLAB

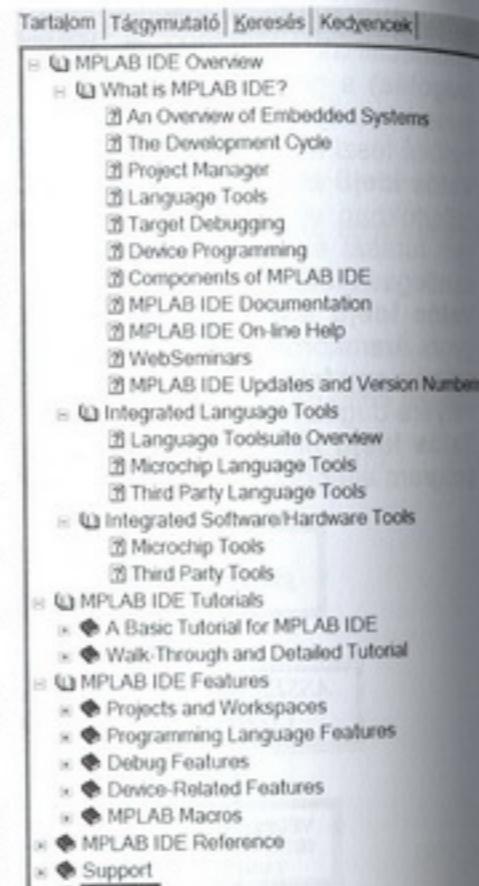
- 8/12 A Microchip által folyamatosan fejlesztett **MPLAB IDE** program 32 bites Windows operációs rendszer alatt fut, és lehetővé teszi a PIC mikrokontrollerek program-32/32 jainak számítógépes fejlesztését.

Az IDE az *Integrated Developing Environment* angol rövidítése, magyarul integrált fejlesztőkörnyezetnek vagy röviden **fejlesztőnek** nevezhetjük. Egy programba integráltak modulárisan minden, ami a PIC mikrokontrolleres fejlesztéshez szükséges.

- MPASM assemblert a 8/12-14-16 bites, MPLAB ASM30 assemblert a 16/24-bites MPLAB ASM32-t a 32/32 bites mikrovezérlőkhöz.
- Egy fejlett szövegszerkesztőt a programjaink írásához.
- MPLINK linker programot, a modulok összefüzetésére.
- MPLAB SIM szimulátort a programjaink teszteléséhez.
- Compilert programjaink gépi kódú fordításához.
- A Microchip összes programozókészüléknek kezelőprogramját.
- VDI vizuális inicializálóprogramot.

A felsoroltakon kívül számos eszköz illeszthető a fejlesztői környezetbe, és a későbbiekben bővíthető is az MPLAB. Például ha a magas szintű C nyelven szeretnénk fejleszteni programjainkat, utólag is telepíthetünk C fordítóprogramot, amely azt követően az MPLAB részévé válik.

A hozzáadható fejlesztőeszközök egy részét a Microchip gyártja, de lehetőség van különböző (third party) eszközeinek is az MPLAB környezetbe történő illesztésére.



6.2. ábra
MPLAB help

6. fejezet: PIC programfejlesztő eszközök

6.1.1. Fejlesztési lépések - projektek

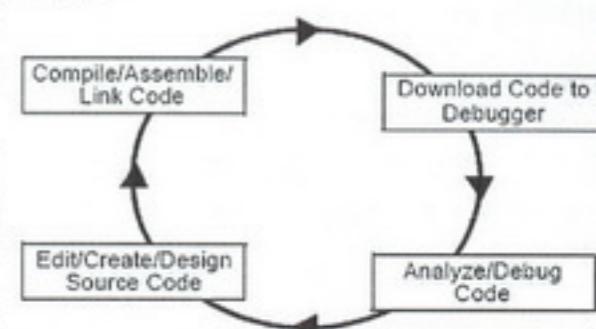
Ahogy azt a 6.3. ábrán láthatjuk, a fejlesztés tulajdonképpen egy körfolyamat, amelyet követve hibamentesen működő programot fejleszthetünk ki.

Minden forráskódot szövegszerkesztőben írunk meg (*Edit/ Create/ Design Source Code*). Majd következik a program fordítása és összefűzése (*Compile/ Assemble/ Link Code*). Ilyen módon létrehozhatunk egy szintaktikai hibáktól mentes programot, de ez még nem azt jelenti, hogy a programunk az eredetileg kitüzött célt valósítja meg. Ezért le kell tölteni egy hibakeresést (debugolást) végrehajtó eszközbe (*Download Code to Debugger*).

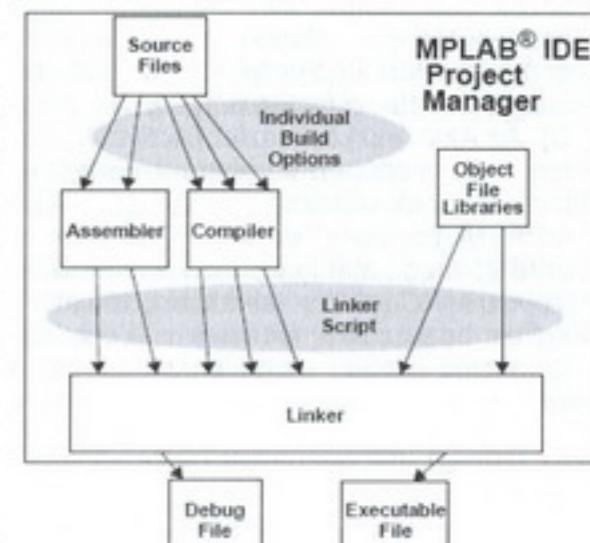
Amennyiben a program nem az elvártaknak megfelelően működik, elemezni kell a hiba okát (*Analyze/Debug Code*), majd a szerkesztőben elvégezni a módosításokat. Így viszszatérünk a ciklus elejére, amelyet egészen addig folytatunk, amíg a programunk tökéletesen nem működik.

Projektek kezelése: Mivel egy fejlesztés során több fájlt hoz létre a fejlesztő, ezért célszerű a fájlokat együttesen kezelni, nyilvántartani. A **projektmenedzser** segítségével, a szövegszerkesztőben megírt forráskódóból (*Source Files*) több lépésben állítjuk elő a futtatható fájlt. Ha a programunkat assembly nyelven írjuk, az assembler, ha pedig magas szintű nyelven, a compiler végezi el a gépi kódú program összeszerkesztését, létrehozását a fordítási opciók alapján.

Amikor a programunkat lefordítjuk, a fejlesztő a projektben különféle kiterjesztésű fájlokat állít elő: *.err a hibaüzeneteket, *.cod a debug információkat és szimbólumokat, *.lst a fordítási listát tartalmazó fájlok kiterjesztése. Ezenkívül létrehoz egy *.o kiterjesztésű object fájlt is, amelyet a linker program használ fel.



6.3. ábra
Fejlesztési ciklus



6.4. ábra
Projektmenedzser

6.1.2. MPLAB installálása, törlése

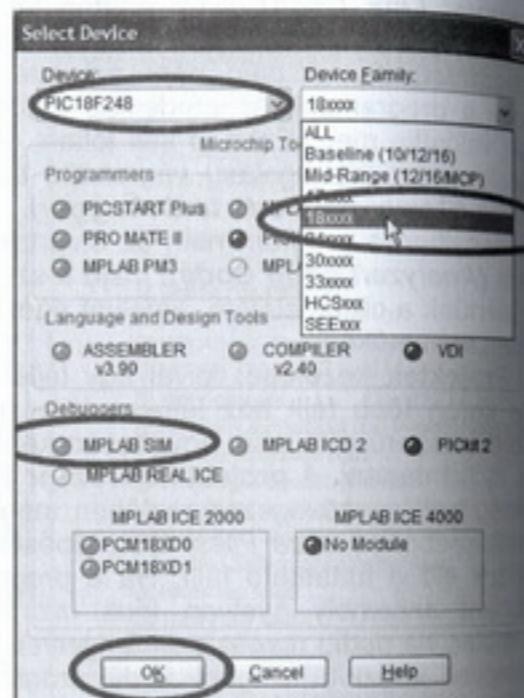
Az MPLAB IDE telepítése, illetve eltávolítása gépünkről a többi programknál megszokott módon történik. Telepítéskor a <http://www.microchip.com> oldalról letöltött, illetve a várolt fejlesztőeszközhöz adott CD-ről a telepítendő *.zip fájlt kicsomagoljuk, és elindítjuk a telepítést. Egy telepítési varázsló vezet végig a folyamatot, az alapértelmezett beállításokat változatlanul hagyva, a Tovább gombra kell kattintanunk a varázsló egyes lépéseihez. Amennyiben a magas szintű C nyelvet választjuk a programjaink írásához az MPLAB IDE telepítése után a megfelelő MPLAB alá illesztett programot is installálni kell. Eltávolításkor hajtsuk végre az alábbi lépéseket:

- válasszuk a Start menü → Vezérlőpult → Programok telepítése és törlése menüpontot;
- a megjelenő listából keressük meg és jelöljük ki az MPLAB Tools (verziószám) programot, majd klikkeljünk a Módosítás / Eltávolítás gombra;
- a telepítési varázsló ekkor három lehetőséget kínál fel, ezek közül a Remove parancsot kell választanunk a program eltávolításához.

6.1.3. Projekt létrehozása

Egy új projektet legegyszerűbben a Projektvarázsló (Project Wizard) segítségével hozhatunk létre. Válasszuk a Project menü Project Wizard menüpontját, majd kövessük a párbeszédpáncelban leírtakat lépésről lépésre.

- Elsőként** a Projektvarázsló üdvözlöképernyője jelentkezik be. A Tovább gombra kattintva,
- a következő** dialógusablakban a legördülő menüből elvégezhetjük az eszköz (a tok) kiválasztását. (A későbbiekben a jellemzőit a Configure → Select Device menüpontot választva nézhetjük meg – 6.5. ábra). Az OK gombra kattintva,
- a megjelenő párbeszédpáncel a nyelvi eszközök beállítására, illetve módosítására kínál lehetőséget. Alapból nyelvi eszközként az assemblert kínálja fel, de választhatjuk a magas szintű nyelveket is (pl. Basic, Pascal, C). Az Aktív eszközkészlet (Active Toolsuite) legördülő menüben az alapértelmezett beállítást hagyjuk változatlanul, így a „Microchip MPASM Toolsuite” eszközeit láthatjuk a legördülő menü alatti eszközkészlet tartalma a Toolsuite Contents ablakban látható. Az egyes eszközökre kattintva a Location információs sávban megjelennek az összetevők elérésének útvonalai.



6.5. ábra
Egy mikrovezérlő jellemzői

Az MPLAB IDE telepítése során az alábbi alapértelmezett könyvtárakba kerültek ezek a fájlok:

MPASM assembler: C:\Program Files\Microchip\MPASM Suite\MPAasmWin.exe
MPLINK linker: C:\Program Files\Microchip\MPASM Suite\MPLink.exe
MPLIB könyvtár: C:\Program Files\Microchip\MPASM Suite\MPLib.exe

- Harmadik lépésként az új dialógusablakban kattintsunk a **Tállózás (Browse)** gombra, jelöljük ki azt a meghajtót és mappát, ahol létrehozzuk a projektet, majd adjuk meg a tetszőleges nevét.
- A Projektvarázsló következő lépésein adhatunk különböző típusú fájlokat a projekt-hez, amelyek segíthetik a munkánkat. Ilyenek például a linkerscript-vezérlő szövegtáblák *.lkr kiterjesztéssel vagy az adott PIC mikrovezérlőre megírt, formailag kialakított és programkódot nem tartalmazó assembly sablon vagy más néven template forrásfájlok, amelyek *.asm kiterjesztéssel találhatók meg. A fájlokat a következő mappákban érhetjük el:

C:\Program Files\Microchip\MPASM Suite\Template\Object
C:\Program Files\Microchip\MPASM Suite\Template\Code
C:\Program Files\Microchip\MPASM Suite\LKR

6. fejezet: PIC programfejlesztő eszközök

Az MPLAB sablon fájljait a későbbiekben is hozzáfüzhetjük projektünkhez. Amennyiben a Projektvarázsló segítségével szeretnénk mindezt megtenni, hajtsuk végre a következő lépéseket:

A párbeszédpáncel bal oldali ablakában a fent leírt útvonalakat követve keressük meg a használni kívánt PIC típusának megfelelő template fájlt, majd kattintsunk az Add gombra.

A jobb oldali ablakban az így kiválasztott fájlt láthatjuk, amely előtt egy kiemelt „A” betű szerepel. Kattintsunk rá háromszor, amíg „C” betű nem jelenik meg a fájl neve előtt, ez teszi lehetővé, hogy a fájlt a projektkönyvtárunkba másoljuk, és az eredeti helyen változatlanul megmaradjon.

Az Object könyvtárban kell keresnünk azokat a mintafájlokat, amelyek az MPASM és MPLINK-kel használhatók. A Code könyvtár pedig azokat tartalmazza, amelyeket az MPLINK nélkül alkalmazhatunk.

Az Object könyvtárban lévő mintafájlok használatakor, assemblálás előtt a projekthez kell adni a linker script fájlt is.

- Használhatjuk a Template fájlokat, amelyeket megjegyzésekkel is elláttak; igaz, angol nyelven íródtak, így a nyelvet nem ismerőknek nem nyújtanak igazán segítséget.
- Második megoldásként átszerkeszthetünk egy kész programot, ha már kellő rutinnal rendelkezünk a programozás terén.
- A fentieken kívül elkezdhetünk írni egy teljesen új programot, szerkeszthetünk saját sablonfájlokat, majd ezeket elmentve alkalmazhatjuk az összes általunk írt programnál.

6.1.4. Projekt és munkakörnyezet

Az MPLAB IDE fejlesztőkörnyezetet használva gyakran találkozunk a projekt és a munkakörnyezet (workspace) fogalmakkal, fontos, hogy megismерjük jelentésükkel, illetve funkcióikkal.

A munkakörnyezet a kiválasztott mikrokontrollerről, a debug-, illetve programozószközökről, a megnyitott ablakokról, azok helyéről és egyéb IDE beállításokról hordoz információt.

A projekt kifejezés arra utal, hogy a fejlesztéskor létrehozott fájlokat egy projektfájlban (*.mcp kiterjesztéssel) rendszerezzük. Elég ezt az egy fájlt megnyitnunk, és a projekthez tartozó összes fájl azonnal a rendelkezésünkre áll. A munkakörnyezetet az *.mcw kiterjesztésű fájl tartalmazza.

Ilyen projektfájlt – ahogyan ezt már az előző alfejezetben láthattuk – a Projektvarázsló segítségével hozhatunk létre.

6.1.5. MPLAB – használat

Az MPLAB munkafelülete négy fő részre tagolódik:

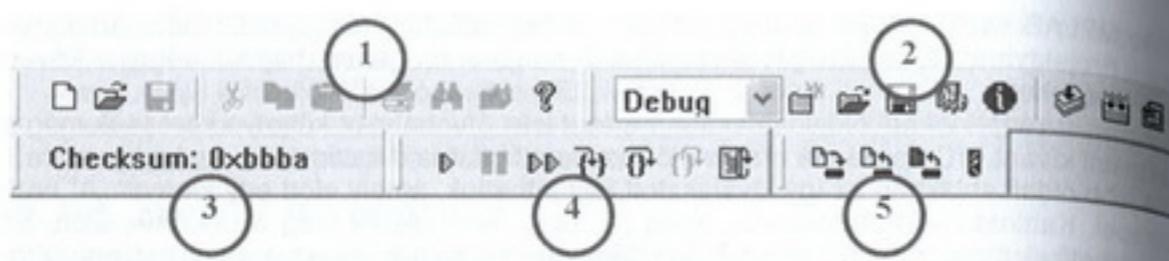
- Legördülő menüt tartalmazó menürendszer
- Szerszámkészlet (Toolbars)
- Az MPLAB munkaaszttal
- Információs vagy állapot-sáv (Status Bar)

A Toolbars és a Status Bar jelentősen megkönnyíti és gyorsítja munkánkat, használatukkal az alábbiakban ismerkedünk meg

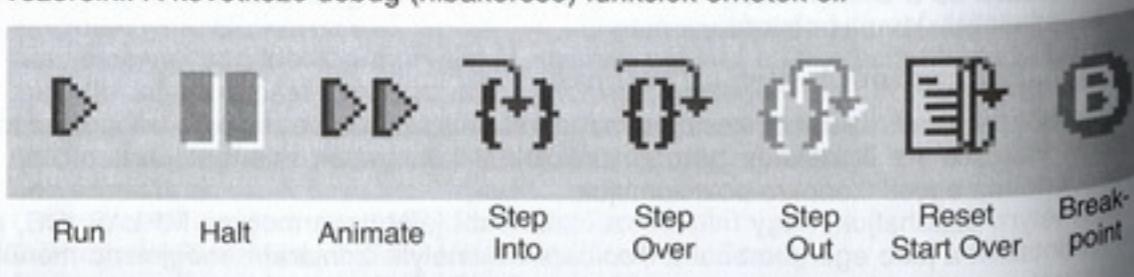
6.1.5.1. Szerzszámkészlet – Toolbars

A fejlesztőkörnyezet szerszámkészlete az adott feladat, illetve eszköz alkalmazásától függően változik. Az ikonokhoz tartozó funkciókról könnyedén informálódhatunk, ha az egérmutatót az egyes ikonokra pozicionáljuk.

Személyre szabhatjuk, hogy milyen szerszámokat jelenítsen meg az MPLAB IDE, ehhez kattintsunk a jobb egérgombbal a Toolbars valamelyik ikonjára, a megjelenő menüből válasszunk az alábbiak szerint:

6.6. ábra
Toolbars

- 1) **Standard:** fájl- és szerkesztési műveleteket érhetünk el könnyedén segítségével, valamint a Help menüt.
 - New File – Új fájlablak megnyitása
 - Open File – Egy már létező fájl megnyitása
 - Save File – A megnyitott fájl változásainak mentése
 - Cut – A kijelölt szövegrész kivágása és vágólapra másolása
 - Copy – A kiválasztott programrész átmásolása a vágólapra
 - Paste – A vágólapról a szöveg bemásolása az aktuális kurzorpozícióra
 - Print – Az aktuális ablakban lévő fájl nyomtatása
 - Find – Keresés a szövegben
 - Find in Files – Keresés fájlból
 - Help – Súgó menü megnyitása
- 2) **Project Manager:** a projekt és munkakörnyezet, valamint a fájlok feldolgozásakor használt ikonokat tartalmazza.
 - Build Configuration – A feldolgozó konfigurálása végezhető el (Debug, illetve Release)
 - New Project – Új projekt létrehozása, név és elérési útvonal megadásával
 - Open Project – Létező projekt megnyitása
 - Save Workspace – Az aktuális projekt és munkaterület mentése
 - Build Options – A projekt fordítási opcionális ellenőrzése, módosítása
 - Toolsuite Info – Információ a nyelvi eszközökről
 - Make – Az aktuális projektben csak azokat a fájlokat dolgozza fel, amelyek megváltoztak
 - Build All – Az aktív projektben az összes fájlt feldolgozza
 - Export Makefile – Makefile és NMakefile nevű szöveges fájlt generál, amelyeket számítógépünkön a projektmappában tárol. A fájlokban követhetjük nyomon, hogy a fordító milyen fájlokat hozott létre, és azokat melyik elérési útvonalon tárolja.
- 3) **Checksum:** egy adott adatsor elemeinek összeadásával képzett összeg, amely az adatsorban bekövetkező változások és sérülések detektálásához használható.
- 4) **Debug:** ez a leggyakrabban használt eszköz, mivel ezzel tudjuk a programunk futását vezérelni. A következő debug (hibakereső) funkciók érhetők el:



- Run – Programunk futtatása
 - Halt – A program végrehajtásának megállítása
 - Animate – Program folyamatos futtatása animálva, így követhető az utasítások végrehajtásának sorrendje
 - Step Into – Mindig a következő utasítást hajtja végre lépésenként
 - Step Over – A szubrutint egy lépésben végrehajtja a CALL utasításnál
 - Step Out – Kilépés szubrutinból
 - Reset – MCLR resetet hajt végre
 - Breakpoint – Töréspont elhelyezése
- 5) **PICkit 2 Debug Toolbar:** amennyiben ezt az eszközt használjuk, a Toolbars az alábbi ikonokkal bővül:
- Program the target device – A programot a tokba tölti
 - Read target device memories – A kontroller memóriáinak kiolvasása
 - Read the target EEDATA memory – EEDATA memória kiolvasása
 - Re-establish PICkit2 connection, re-check device ID, power, etc. – újra kapcsolódik az eszközhöz, és újra ellenőrzi.

6.1.5.2. Állapotsáv – Status Bar

A Status Bar információt nyújt a fejlesztőkörnyezet aktuális állapotáról. Amennyiben futtatjuk a programunkat, az állapotsáv a *Running...* felirattal jelzi a végrehajtást. Egyébként pedig láthatjuk

- a Debugger menüben kiválasztott fejlesztőeszköz nevét,
- a PIC mikrovezérlő típusát,
- az utasításszámláló (PC) aktuális értékét; duplán rákattintva módosítható az érték a szimulátorban,
- a W regiszter aktuális értékét,
- a STATUS regiszter bitjeit; a nagybetűs bit logikai 1, a kisbetűs pedig logikai 0 értékű,
- a processzor órajel-frekvenciáját,
- az adatmemória aktuális Bank-információját,
- annak a sorak és oszlopnak a számát, ahol a kurzor áll,
- az Insert funkció használatát (átírja az adott karaktert),
- a fájl írási és olvasási információt (WR – írható, RO – csak olvasható).

6.1.5.3. A dokkolás (Dockable) funkció használata

A programírás elkezdése előtt érdemes a folyamatosan használt ablakokat kiválasztani és a képernyön úgy eligazítani, hogy a későbbi munkánk során könnyen átláthatóak legyenek. Az aktuális ablakbeállításokat az alábbiak szerint el is menthetjük, elkerülve azt a kellemetlen jelenséget, amikor egy-egy újabb ablakot megnyitva az éppen használt „eltűnik”.

Kattintsunk a View menüre, és jelöljük ki azokat a nézeteket, amelyeket a munkaterületen látni szeretnénk. Válasszuk ki a legfontosabbakat, a Project, Output és Watch stb. nézeteket, majd az ablakokat állítsuk be úgy az MPLAB munkaasztalon, hogy minél több információhoz juthassunk egyszerre a képernyőn. Ezután az összes ablaknál használjuk a dokkolás (Dockable) funkciót, ami rögzíti az ablakot, azaz minden látható lesz a képernyőn. Az ablak beállítását segítő menüt elérhetjük, ha a projektnév mellett balra található fehér téglalapra a bal egérgombbal kattintunk.

Az MPLAB IDE lehetővé teszi, hogy az ablakbeállításainkat elmentessük. A funkció azért előnyös, mert egy projekt megnyitásakor az ablakok nem a megszokott módon helyezkednek el. Ekkor elég meghívunk a saját elmentett beállításainkat: kattintsunk a Window menüpontra, majd válasszuk a Create Window Set parancsot. Ezután adjunk meg egy tetszőleges nevet, amelyen menti a beállításokat a program, majd klikkeljünk az OK gombra.

Az elmentett beállítást a Window menüpontra kattintva ismét alkalmazhatjuk, ha a Window Set parancsot választjuk.

6.1.5.4. Special Function Registers – SFR regiszterek

A rendszerregisztereket és a perifériaregisztereket összefoglaló néven speciális funkciójú regisztereknek vagy röviden SFR-eknek nevezzük. Programjaink írásakor nagyon gyakran találkozunk ilyen regiszterekkel. Segítségükkel állíthatjuk be például a mikrovezérlő különböző részeit, perifériáit. Az egyes regiszterek bitjeinek funkciójáról az adott típusú PIC adatlapjáról részletesen tájékozódhatunk.

Kattintsunk a View → Special Function Registers menüpontra, ekkor az oszlopokba rendezve megjeleníti az adatokat. Beállítástól függően (az oszlopakra a jobb egér-gombbal kattintva) tekinthetjük meg az SFR regiszterek neveit, a címeit és különböző formátumokban az értékeit. Felépítésében nagyon hasonló a Watch ablakhoz. Az ablak tartalmazza az adott PIC-típushoz használható összes SFR regisztert, így a program futása közben ezeket külön figyelhetjük.

Az SFR regisztereket a Watch ablakhoz is hozzá lehet adni, ekkor az előbbi ablakot nem is kell használnunk.

6.1.5.5. Disassembly listing

Ha programunkat a generált gépi kóddal együtt szeretnénk látni, akkor válasszuk a View → Disassembly Listing parancsát. Ekkor az ablak jobb oldalán a programunkat, balra pedig egy nagyon érdekes kódlistát láthatunk, amelyet aztán hibakeresésre vagy nyomon követésre használhatunk. A disassembly magyarul visszafordítást jelent, tehát a program a gépi kódóból visszafordított assemblylistát jeleníti meg. Lényegében – szimbolikusan – a programunk memóriában történő elhelyezkedését láthatjuk.

Az egyes oszlopok a következő információkat tartalmazzák (balról jobbra haladva). Vizsgáljuk a 6.7. ábrán látható első sort:

6.7. ábra
Visszafejtett lista

- a programmemória címe (0000), ez a programszámláló értéke (pc: 0x00);
- az utasítás hexa formában (0E72), amely az utasítást és az operandus értékét foglalja magában;
- az utasítás neve (MOVLW = Move Literal to Wreg – egy konstanst tölt W regiszterbe);
- a literál értéke (0x72);
- a visszafejtett listában lévő sor száma (ezzel lehet adott sorra hivatkozni, ez azonosítja, ha valamelyik sorban hiba van);
- az assemblerben szereplő eredeti sor megjegyzéssel együtt.

Az utasítások típusáról, felépítéséről, valamint a programmemóriában történő elhelyezkedésről az utasításkészlet leírásában tájékozódhatunk részletesen.

6.1.5.6. Hardware Stack – Verem használata

Amikor a programunkban megszakítást használunk, a megszakított program programszámlálójának értékét elmentjük a Stackbe, közismertebb nevén a verembe. Miután a megszakítást kissolgáló rutin végrehajtása befejeződik, a megszakított program programszámlálója a veremből töltödik vissza.

A Hardware Stack szintén a View menüből hívható meg. Használatának a prioritásos, illetve a többszintű megszakításokat kezelő programoknál van jelentősége. A PIC 18-as család már támogatja a prioritásos megszakításkezelést.

Amíg nincs megszakítás vagy szubrutinhívás, a veremmutató a 0-s szinten áll. Ha lét-rejón valamelyik esemény, akkor a visszatérési cím kerül a verembe, és a veremszint mutatója 1-re vált. Az ábrán látható, hogy egy szubrutin vagy megszakítás hajtódik végre, és ha befejeződik (RETIE, RETURN), akkor az utasításszámlálóba ismét a 0x2A íródik, vagyis a megszakított program onnan folytatódik.

| Hardware Stack | | | |
|----------------|-------------|----------------|----------|
| TOS | Stack Level | Return Address | Location |
| | 0 | Empty | |
| ⇒ | 1 | 00002A | |
| | 2 | 000000 | |
| | 3 | 000000 | |
| | 4 | 000000 | |

6.1.5.7. A program működésének ellenőrzése, hibakeresés (debug)

A programellenőrző-hibakereső eszköz, a DEBUGGER az MPLAB IDE fontos része, és ha erre a céla akár az MPLAB SIM, MPLAB ICE, MPLAB ICD 2, MPLAB PICKIT1, MPLAB PICKIT2 eszközöket használjuk, a legtöbb művelet hasonló, a kezelőfelület pedig azonos. Vagyis programfejlesztéskor a szimulátor használhatjuk, és ha kész a hardver, használhatjuk a hardverdebuggert, valós környezetben folytatva a tesztelést. Az alapvető debug funkciók a következők:

- Reset:** a fejlesztett program (eszköz) alapállapotba állítása, hogy a programot újra tudjuk indítani;
- Execute:** A programkód végrehajtása;
- Halt:** A programfutás megállítása adott ponton (breakpoint). Amikor megállítottuk, a memória és a változók vizsgálata, módosítása;
- Single Step:** Lépésenkénti futtatáskor egyenként hajtjuk végre a gépi kódú utasításokat (vagy a C program utasításait), figyelem közben a változók, regiszterek, jelzöbök változását.

A debugger további lehetőséget is biztosít a használatának könnyítése érdekében:

- Watch ablak:** kiválasztott változók, memóriahelyek megjelenítése különféle formában.
- Nyomkövetés tárolása (Trace buffers)** amivel a végrehajtott utasítássorozatot szöveges formában eltárolhatjuk, mellette a kijelölt változók értékeit is eltárolva.
- Stopwatch:** egy adott kód részlet végrehajtási idejét tudjuk megmérni.
- Összetett töréspontok:** nem csupán az utasításszámláló adott értékénél lehet megállni, hanem mellette még egyéb feltételeknek is teljesülnie kell, például egy regiszternek adott értéknek kell lenni, vagy a törésponton történő adott számú áthaladás után.

A következőkben az itt általánosan leírtakat részletezzük.

6.1.5.8. Breakpoint – töréspont

Amikor programunkat szimulátorban futtatjuk, esetleges hibakeresés vagy a változók, memóriák, regiszterek nyomon követése esetén nagyon sokszor szükséges, hogy egy adott utasításnál megállítsuk a működést. Ezt a célit szolgálja a **breakpoint** vagy magával a **töréspont**. Törésponton való megálláskor a regiszterekben az éppen aktuális értékek vannak, ennek elemzésével előírható, hogy a töréspontig történt programvégrehajtás helyes volt-e. Ezért célszerű a programunkat logikailag jó elhatárolt részekre osztani a töréspontok segítségével, így könnyen kiszűrhető, hogy melyik programrész hibás.

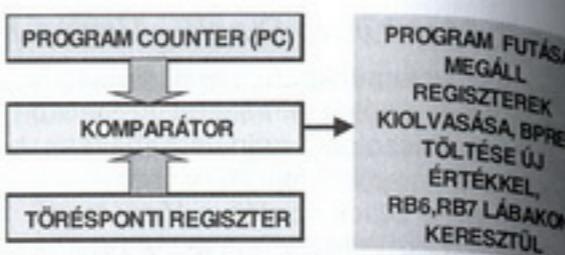
Töréspontokat helyezhetünk el a szerkesztőablakban, de a *Program Memory* és a *Disassembly* ablakokban is lehetséges. Több lehetőség is kínálkozik, hogy elhelyezzük, illetve eltávolítsuk a töréspontokat:

- az adott programsorra duplán kattintva minden két művelet végrehajtható;
- abban az ablakban, ahol a töréspontot elhelyezzük, jobb egérgombbal kattintva a megjelenő menüben válasszuk a *Set Breakpoint* parancsot (akkor a töréspont a kurzorral kijelölt sorba kerül). Amennyiben törlni vagy tiltani szeretnénk a töréspontot, ebben a menüben válasszuk a *Remove Breakpoint*, illetve a *Disable Breakpoint* menüpontokat (több töréspont esetén is elérhetők ezek a funkciók, és alkalmazhatók akár az összes töréspontra).
- kattintsunk a *Debugger* → *Breakpoints* parancsra, és a megjelenő ablakban elvégezzük a töréspontokkal kapcsolatos összes beállítást.

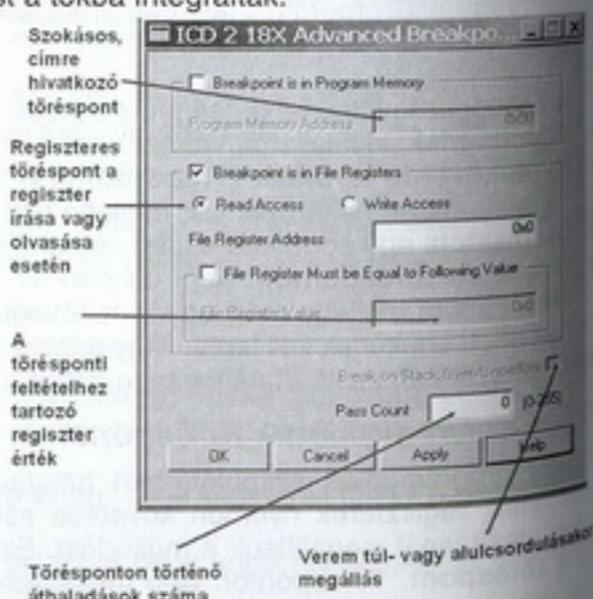
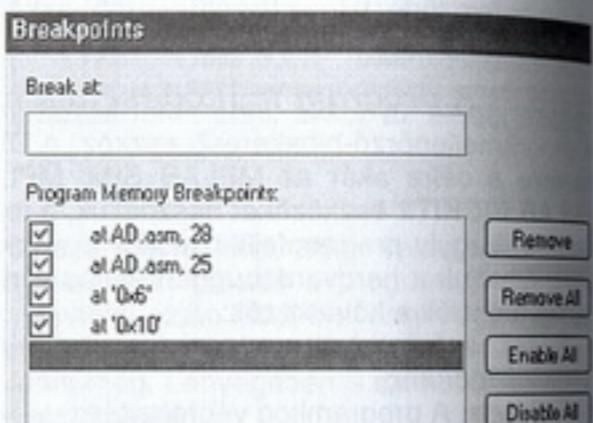
A *Debugger* → *Breakpoints* parancsra megjelenő ablakban összefoglalón láthatjuk, hogy melyik programrészben, hányadik sorban helyeztük el a töréspontot. Itt is elvégezzük a fentiekben már említett funkciókat (törles, engedélyezés, letiltás). A törésponton való megállás feltétele az, hogy a törésponti cím megegyezzen az utasításszámláló (PC) tartalmával. A töréspont megvalósítására kezdetben csak szoftveres lehetőség volt, a szimulátor használatakor tetszőleges számú töréspont helyezhetünk el a programban. A PIC16F87x család megjelenésével a töréspontkezelést a tokba integrálták.

A törésponti működés úgy valósítható meg, hogy egy több bites digitális komparátor áramkör egyik bemeneti oldalára az utasításszámláló bitjeit, míg a másik bemeneti oldalára egy, a debug programból feltöltött ún. törésponti regiszter bitjeit kapcsoljuk. A két oldal egyezésekor teljesül a törésponti feltétel, és a komparátor ilyenkor megjelenő kimemeti jelét felhasználva meg lehet állítani a program futását.

A program törésponton történő megállása után indul el a tok programmemoriájának utolsó 256 szavában elhelyezkedő **debug program**, ami lehetővé teszi új törésponti cím beírását, illetve lehetőség van az adatmemória elérésére, vagyis olvasására/írására is. Mindez az MPLAB kezelőfelületén keresztül valósul meg soros kommunikáció felhasználásával.



6.8. ábra
Hardveres töréspont



6.9. ábra
Fejlett töréspontkezelés

A törésponti hardver működési elvából következik, hogy minden törésponti cím elérhető meg, ott megállva adható meg a következő. A 8 bites PIC18-as, valamint a 16 és 32 bites mikrovezérlőknél a töréspontkezelés jelentősen kibővült (*Advanced Breakpoint*). Az új lehetőségeket a 6.9. ábrán követhetjük nyomon.

6.1.5.9. Watch ablak

A View legördülő menüből a *Watch* feliratra kattintva aktivizálhatjuk. Az ablak alkalmas a program regisztereinek, változóinak, szimbólumainak futás közbeni megfigyelésére. Egy projekthez egyszerre **négy watch** ablak is választható. Azokat az SFR regisztereit, illetve szimbólumokat, amelyekre kíváncsiak vagyunk, legördülő menükben választhatjuk ki. Ha egy változót szeretnénk nyomon követni, keressük meg a programunkban, majd az egérrrel egyszerűen húzzuk be a *watch* ablakba.

Az ablakban megjelenő oszlopokat személyre szabhatjuk. Klikkeljünk a jobb egérgombbal az oszlopok nevére, és a menüben pipáljuk ki azokat a nézeteket, amelyeket szeretnénk megjeleníteni (*Hex*, *Decimal*, *Binary*, *Char*). A beállításokat megtehetjük akkor is, ha a *Watch* ablakra kattintunk a jobb egérgombbal, és a *Properties* parancsot választjuk. A megjelenő menüben elmenthetjük, átnevezhetjük, törlhetjük a *watch* ablakokat, vagy betölthetünk egy már korábban mentett verziót is.

Gyorsabban kereshetünk a legördülő menükben, ha a kívánt regiszter vagy szimbólum kezdőbetűjét adjuk meg. Az *Add SFR* és az *Add Symbol* gombra kattintva jeleníthetjük meg a tartalmukat a *watch* ablakban.

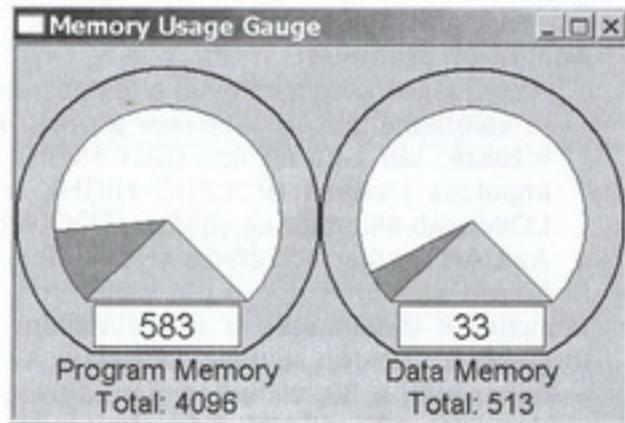
Lehetőségünk van arra is, hogy a *Watch* ablakban lévő regisztereknek, változóknak kezdőértéket adjunk. Például ha egy 8 bites időzítő működését figyeljük, és programunk az időzítő túlcordulásakor lép tovább, meggyorsíthatjuk ezt a folyamatot. Ilyenkor a TMR1 regiszter tartalmát az aktuális értékre duplán kattintva módosíthatjuk, a szimuláció az új értékről folytatódik. Abban az esetben, ha olyan módosítást akarunk végrehajtani, amely ütközik a program korábban végrehajtott parancsaival, a szimulátor nem engedi átmi az értéket.

6.1.5.10. Felhasznált program- és adatmemória

A program- és az adatmemoriában lefoglalt területek nagyságáról a *View* → *Memory Usage Gauge* menüpontra kattintva kapunk információt.

Az ábrán láthatjuk a teljes program- és adatmemória-területeket, számértékkal, valamint diagramos formában megjelenítve a felhasznált és a szabad területeket.

| Watch | | | | | |
|---------|-------------|----------|---------|------------|-----------|
| | | Add SFR | ADCON0 | Add Symbol | P_BSRTEMP |
| Address | Symbol Name | Hex | Decimal | Char | |
| FF6 | TBLPTR | 0x000000 | 0 | ... | |
| FF5 | TABLAT | 0x00 | 0 | .. | |
| FE8 | WREG | 0x00 | 0 | .. | |
| FE1 | FSR1 | 0x00 | 0 | .. | |
| 00E | R_VALT | 0x41 | 65 | A | |
| 012 | R_KOD | 0x00 | 0 | .. | |
| FC2 | ADCON0 | 0x00 | 0 | .. | |



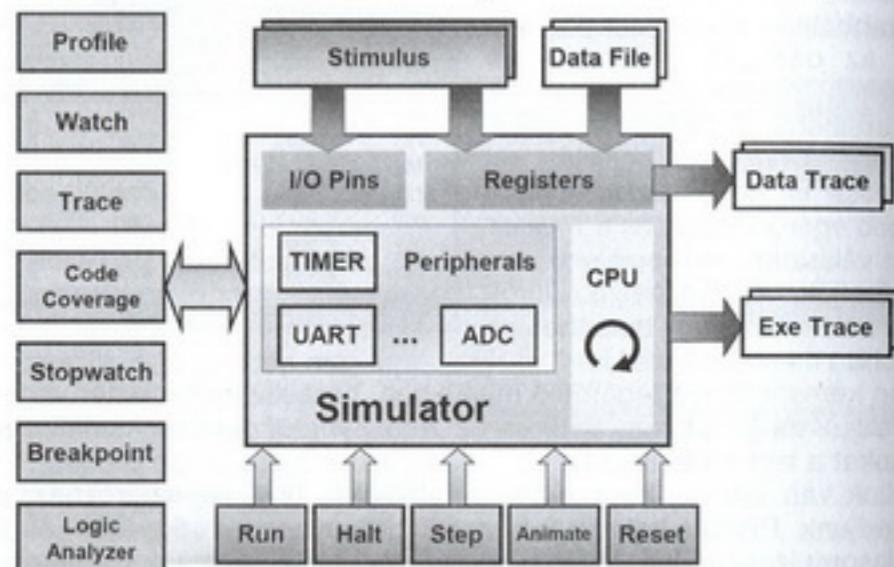
6.1.6. Szimulátor használata és nyomkövetés

A PIC mikrovezérlökre írt programjaink tesztelésére az MPLAB IDE fejlesztői szimulátorprogramot integráltak a fejlesztői környezetbe. Aktivizálásához a *Debugger → Select Tool → MPLAB SIM* menüpontot kell kiválasztanunk. A szoftverszimulátor egy program, ami a gerjesztésekre pontosan úgy reagál, mint maga mikrovezérlő.

A szimulátort felfoghatjuk úgy, mint egy olyan interaktív programot, amely utasításokonként hajtja végre a műveleteket.

Stimulusnak nevezzük azon szimulált jelek összességét, amivel a programunk külső környezetét tudjuk figyelembe venni, a be- és kimeneti portokra vagy perifériaregiszterekbe juttatott tartalom segítségével.

A legtöbb periféria működését képes regiszter- (és nem áramköri!) szinten szimulálni. A szimulátor működését a futtatást vezérlő gombokra kattintással tudjuk irányítani.



6.10. ábra
Szimulátor bokvázlata

A szimulátorral nemcsak a program végrehajtását ellenőrizhetjük, illetve követhetjük nyomon, hanem a külső környezetből jövő adatokat egy fájlban elhelyezve lehetséges a program szimulálása során ezeket beolvasni, illetve a kiküldött adatokat is eltárolhatjuk egy fájlban. A stimulusoknak két fajtája van:

- **Aszinkron stimulus:**

- o Az I/O lábak állapotváltását a felhasználó indíthatja, a szimulátor megfelelő gombjára való kattintással, **bármikor a program futása közben**. A választható állapotváltozások: lab 1-be állítása (SET HIGH), lab 0-be állítása (SET LOW), magas szintű impulzus kiadása (PULZUS HIGH), alacsony szintű impulzus kiadása (PULZUS LOW), lab állapotának váltása (TOGGLE).

- o Az UART pufferregiszterbe (RCREG) egyenként egy karaktersorozat juttatása.

• **Szinkron stimulus:** Előre programozzuk, hogy mikor következzen be, I/O lábakra és regiszterek tartalmának a változtatására is használható. Milyen események esetén történhet a szinkron stimulus aktivizálása? Például:

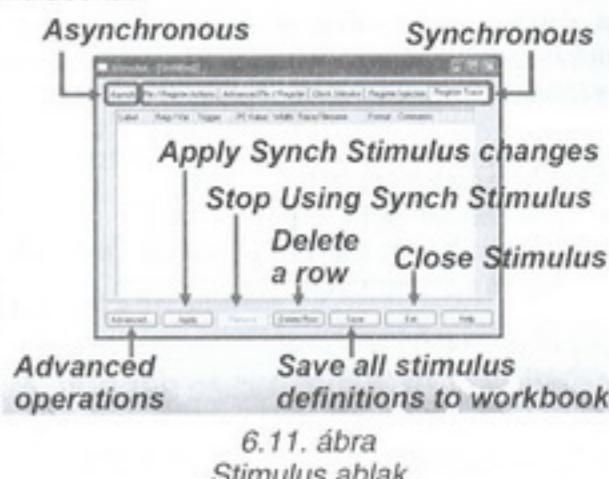
- o mikor eléri a 30. ciklust, a programvégrehajtást, vagy a program futásának a kezdete után 120 ms-mal;
- o mikor az utasításszámláló értéke eléri a 0xCO4 címet;
- o mikor az ADCBUF tartalmát kolvassuk;
- o mikor az RA4 lab 1-be vált.

A szimulátor gerjesztéseit (a stimulusokat) a Workbookban tároljuk. Ez egy fájl, aminek nevet adunk. Az állomány kezelése a szokásos New, Open, Save, Save Workbook As, illetve Close parancsokkal lehetséges.

Hogyan adjuk meg, illetve használjuk a stimulusokat?

- 1) Megnyitjuk a Stimulus ablakot (6.11. ábra): *Debugger → Stimulus → New Workbook*. A fülek jelölik az egyes gerjesztési lehetőségeket. Az első fül az aszinkron stimulusok megadására szolgál (Asynch), míg a többi fül a szinkron stimulusok megadásához hasznáható.
- 2) Megadjuk a stimulusokat.
- 3) Ha szükséges, ezután elmentjük a stimulusokat: *Debugger → Stimulus → Save Workbook*.
- 4) Rákattintunk az *Apply* gombra. (Alkalmazzuk a beállított stimulusokat.)
- 5) A stimulus ablak nyitva marad.
- 6) Elkezdjük a szimulációt.

A projekt betöltését követően le kell fordítanunk a programot (*Project → Build All*), majd elkezdhetjük a szimulálást. Amennyiben a fordító nem talál hibát programunkban, létrejön a *.hex kiterjesztésű fájl. Abban az esetben, ha szintaktikai vagy szemantikai hibát ejtettünk a program írásában, a fordításkor a hibára utaló üzenet jelenik meg az Output ablakban. Az üzenetre duplán kattintva egy mutató a hibát okozó programsorra ugrik.



6.11. ábra
Stimulus ablak

6.1.7. Stimulus használata

Programunkat szimulátorban elindíthatjuk, leállíthatjuk, akár lépésenként (*Step Into*) vagy animálva (*Animate*) is futathatjuk. Míg az első üzemmód mindenkor csak egy programsort hajt végre, az animálás folyamatos programfuttatást jelent, amely közben vizsgálhatjuk a regisztereket vagy a tártartalmat is. A második esetben a végrehajtás sebességét a *Debugger → Settings* menüpontban az *Animation → Realtime Updates* fülön állíthatjuk be. Ha animációs üzemmódban figyelni szeretnénk a változók és regiszterek aktuális helyzetét, pipáljuk ki az *Enable Realtime watch updates* jelölőnégyzetet.

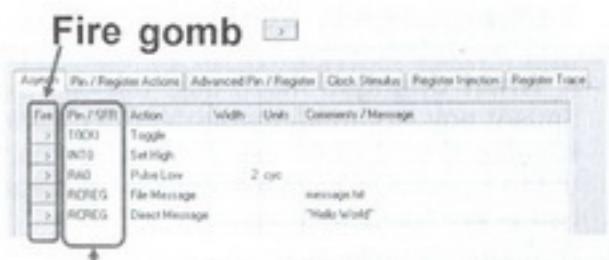
Az adatok megadásánál a következőket vegyük figyelembe: az időegységek minden decimálisak, PC és a regiszterek értéke minden hexadecimális, a lábak megadása 0 és 1 lehet.

FIGYELEM! A következőkben ismertetett számos menüpont csak akkor jelenik meg, ha eszközként az MPLAB SIM-et választjuk a Tools legördülő menüben!

6.1.8. Aszinkron stimulus

A Pin/SFR menüre kattintva választhatjuk ki a jelforrásat. **Action:** a gerjesztés mikéntjét: Set High/Low Pulse High/Low, Toggle (állapot az ellentétre).

RCREG esetén: a soros bemenetre vagy egy fájlból: File Message, vagy előzetesen begépelve: Direct Message érkeznek a karakterek. A „>” gombra kattintva lehet a gerjesztést aktivizálni.



Cél : I/O lab v. RCREG

6.1.9. Szinkron stimulus

6.1.9.1. Pin / Register Actions

A **Click here to Add Signals** gombra kattintva egy legördülő listából választhatjuk ki a lábakat, regisztereket, illetve azok bitjeit.

Az időt meg lehet adni:

- utasításciklusban (**cyc**),
- időben: **h:mm:ss** (óra:perc:mp) vagy **ms**, **μs**, **ns** számértékkel.

RESET után 10 ciklus múlva RB0 értéke 1, PORTA = 0x1E, WREG = 0x80 SSBPUF bitjei: 00000001 legyen.

RESET után 12 ciklussal: SSBPUF bitjei 01010100-ra változzanak stb.

A **Repeat** mezőben adhatjuk meg, hogy mennyi várakozás után ismételje meg a beállított gerjesztéseket.

A **Restart** mezőben adhatjuk meg, hogy ne az elejéről, hanem valamelyik gerjesztéstől kezdve ismételjen.

6.1.9.2. Advanced Pin / Register

Láb-, illetve regisztergerjesztés létrehozása, valamilyen feltételtől függően.

Feltétel:

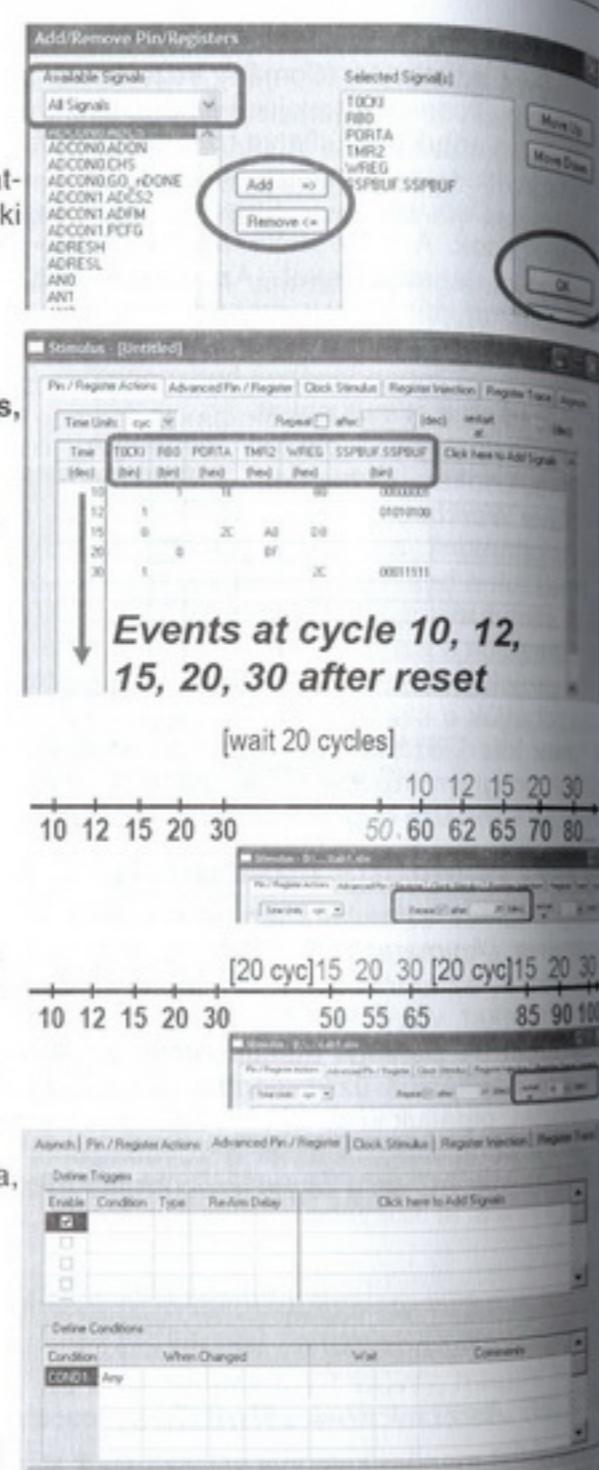
- pl. ha TMR0 tartalma 0xE0,
- vagy INT1IF = 1.

Gerjesztés:

- egyszer,
- mindaddig, amíg a feltétel fennáll.

Elsőként a **Define Conditions** (feltétel megadása) táblát használjuk.

- A **When Changed** mező első oszlopára kattintva megadhatjuk a változás forrásának típusát egy legördülő menüből: SFR, Bitfield, Pin, All (bármelyik az előző háromból). A következő oszlopóból már az előző alapján felkínált legördülő listából választhatunk. Az utána következő cella a feltétel megadása egy legördülő menüből (<, <=, =, !=, >, >). Utána kell beírni az értéket.
- A **Wait** mezőben adjuk meg azt, hogy még hány időegységet várakozzunk a feltétel teljesülése után, hogy a szimulátor végrehajtsa a gerjesztést. A számérték megadása után történik az időegység választása egy legördülő menüből.



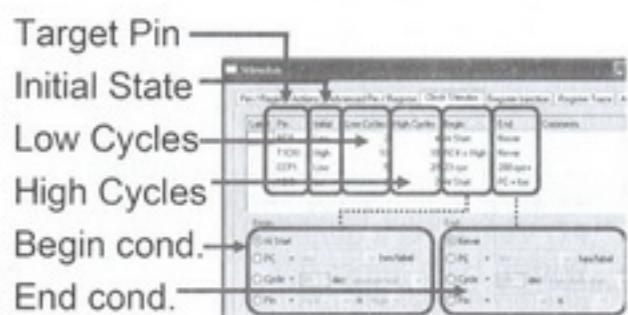
Egy sor megadása után automatikusan létrejön a COND# elnevezésű feltétel. A következő lépés a felső, **Define Triggers** (gerjesztés indítása) táblázat kitöltése.

- Az **Enable** oszlopban kipipáljuk, ha engedélyezzük ezt a triggert.
- A **Condition** mezőben szerepelhetetjük a megfelelő COND# feltételt.
- **Type:** Itt adjuk meg a trigger típusát: egyszer történjen vagy ismétlődően.
- **Re-Arm Delay:** Ha az ismétlődő triggerelést választottuk, akkor itt adjuk meg azt a késleltetést, aminek elteltével a feltételezett újra megvizsgáljuk. (Késleltetés megadása: számérték és utána időegység választása egy legördülő menüből.)
- **Click Here to Add Signals:** Itt kell kiválasztani azt a lábat, regisztert vagy más jeleket, amire a gerjesztést alkalmazzuk.

6.1.9.3. Clock Stimulus

Egy kiválasztott lábon 0 és 1 értékekből álló hullámformát hoz létre. Meg kell adni a lábat, a láb kezdeti állapotát, az alacsony és magas állapotok hosszát, valamint a generálás kezdetét és a végét, amik feltételektől függnek.

- **Label:** az órastimulus elnevezése.
- **Pin:** az a láb, amelyikre az óragerjesztést alkalmazzuk.
- **Initial State:** az órajelgerjesztés indulása előtt mi legyen az indulási állapota.
- **Low Cycles:** alacsony szintű állapotok száma az órajelben.
- **High Cycles:** magas szintű állapotok száma az órajelben.
- **Begin cond.:** négy opció választható:
 - **At Start:** (ez az alapértelmezett) – a program futásának kezdetétől működik az órajel-generálás,
 - **PC:** amikor a PC eléri az itt megadott értéket, attól **kezdve lesz** órajel-generálás.
 - **Cycle:** akkor **kezdődik** az órajel-generálás, amikor a program által végrehajtott ciklusok száma eléri az itt megadott értéket.
⇒ **Absolute time** – a szimuláció kezdetétől számítjuk;
⇒ **After last clock** – az utolsó órajeltől.
 - **Pin:** akkor kezdődik az órajel-generálás, amikor az itt megadott lábon a szintén megadott **Low** vagy **High** állapot létrejön.
- **End:** itt is a Beginnél leírt hasonló négy opció választható, csak nem elindulást, hanem megállást jelez.
 - **Never:** a program a megállásáig működik.
 - **PC:** amikor a PC eléri az itt megadott értéket, attól **kezdve nem lesz** órajel-generálás.
 - **Cycle:** akkor **fejeződik be** az órajel-generálás, mikor a program által végrehajtott ciklusok száma eléri az itt megadott értéket.
⇒ **Absolute time:** a szimuláció kezdetétől számítjuk.
⇒ **From clock start:** a Begin szekciójában adott megkezdett időtől számítjuk.
 - **Pin:** akkor **fejeződik be** az órajel-generálás, amikor az itt megadott lábon a szintén megadott **Low** vagy **High** állapot létrejön.
- **Comment:** Megjegyzés.



6.1.9.4. Regisztergerjesztés (Register Injection)

Segítségével a regiszterekbe egy fájlban tárolt értékeket tudunk juttatni. Például egy üzenetet az UART-ba, vagy egy felhasználói regiszterbe tudunk értéket beírni a gerjesztéskor.

A regisztergerjesztés megadása:

- **Label**: Megadható a regisztergerjesztés neve (opcionális).
- **Reg/Var**: Egy listából a célregiszter nevét kell megadni, ahova a gerjesztés tartalma kerül. Ez lehet SFR vagy felhasználói regiszter.
- **Trigger**: A gerjesztési feltétel kiválasztása:
 - kérésre (**On Demand**), vagyis amikor a program olvassa a regiszter/változó értékét;
 - amikor a PC tartalma egyenlő egy adott értékkel (**PC=**);
 - Üzenet – ez csak UART esetén használható (RCREG-be kerülnek az üzenet bajai). Soros szimulációt lásd később.
- Ha perifériát használunk, on demand aktivizálható. Ha periféria nincs, akkor csak PC értékkel aktivizálható. Pl. ha ADON bitet 1-be állítjuk, és ADRESL regiszter elérhető, az érték ADRESL-be kerül.
- **PC Value**: Ha a gerjesztés (Trigger): PC=, akkor PC értékét itt kell beírni.
- **Width**: Ha a gerjesztés (Trigger): PC=, akkor itt kell beírni a bevitt bajtok számát. SFR esetén 8 bitesknél értéke:1, 16 bitesknél értéke: 2. GPR esetén a felhasználó adja meg a hosszt.
- **Data Filename**: A gerjesztést tartalmazó fájl neve.
- **Wrap**: **Yes**: Ha a fájlból szereplő összes értéket felhasználtuk, akkor a gerjesztés a fájl kezdetétől folytatódik. **No**: Ha a fájlból szereplő összes értéket felhasználtuk, a gerjesztés csupán az utolsó értékkel folytatódik.
- **Format**: A regisztergerjesztő fájl formátumának a kiválasztása.
 - Hex: ASCII hexadecimális értékeket tartalmaz.
 - Raw: Bináris értékek vannak a fájlból.
 - SCL: SCL formátum. Lásd az SCL fájlformátum leírását.
 - Dec: ASCII decimális értékeket tartalmaz.
- **Comments**: A gerjesztést leíró információ megadása.

6.1.9.5. Regiszter-nyomkövetés (Register trace)

A program futtatásakor a kiválasztott regiszter tartalmát egy fájlba küldjük. Felépítése nagyon hasonló a Register Injection leírásában szereplő táblázathoz. FSR/GPR egyaránt használható. Nincs WRAP oszlop.

A nyomkövetési feltételek megadása a következő:

- **Label**: A regiszter-nyomkövetés neve adható meg (opcionális).
- **Reg/Var**: A nyomon követett regiszter nevét kell itt megadni egy legördülő listából. Lehet SFR vagy felhasználó által definiált regiszter.
- **Trigger**:
 - kérésre (**On Demand**), vagyis amikor a program megváltoztatja a regiszter/változó értékét.
 - minden esetben, amikor a PC tartalma egyenlő egy adott értékkel (**PC=**). Ezt lehet egy címkevel is azonosítani.

| Pin / Register Actions Advanced Pin / Register Clock Status Register Injection Register Trace Asyn | | | | | | | | | |
|--|-----------|---------|----------|-------|-----------------|--------|----------|------|---------|
| Label | Reg / Var | Trigger | PC Value | Width | Trace Filename | Format | Comments | Asyn | Actions |
| (optional) PORTA | PC = | Isa | | 1 | porta_output.b6 | Hex | optional | | |
| (optional) ADRESL | Demand | | | 1 | adc_input.b6 | No | Dec | | |
| (optional) RCREG | Message | | | 1 | usd_input.b6 | Yes | Plz | | |

| Pin / Register Actions Advanced Pin / Register Clock Status Register Injection Register Trace Asyn | | | | | | | | | |
|--|-----------|---------|----------|-------|----------------|--------|----------|------|---------|
| Label | Reg / Var | Trigger | PC Value | Width | Trace Filename | Format | Comments | Asyn | Actions |
| (optional) WREG | Demand | | | 1 | wreg_output.b6 | Hex | optional | | |
| (optional) PC = | Isa | | | 1 | wreg_input.b6 | Hex | optional | | |

Ha perifériát használunk, az csak kérésre (on demand) aktivizálható. Ha periféria nincs, akkor csak PC értékkel aktivizálható. Pl. ha ADON bitet 1-be állítjuk, és ADRESL regiszter elérhető, az ADRESL értéke kerül a fájlba.

- **PC Value**: Ha a nyomkövetés: PC=, akkor PC értékét itt kell beírni.
- **Trace Filename**: A nyomkövetés eredményét tartalmazó fájl megadása.
- **Format**: A regisztergerjesztő fájl formátumának a kiválasztása.
 - Hex: ASCII hexadecimális értékeket tartalmaz.
 - Raw: Bináris értékek vannak a fájlból.
 - SCL: SCL formátum. Lásd az SCL fájlformátum leírását.
 - Dec: ASCII decimális értékeket tartalmaz.
- **Comments**: A nyomkövetéssel kapcsolatos információ megadása.

6.1.9.6. Soros vonali szimuláció

A Register Injection stimulus és a Register Trace nagy segítségünkre lehet például egy soros vonal kezelését megvalósító program szimulálásában. Ilyen programokban általában részfeladat, hogy valamelyen adatot küldjünk be a soros vonalon, és azt az átalakítás után ismét a soros vonalat használva küldjük vissza.

Első lépésként két *.txt kiterjesztésű fájlt kell létrehoznunk, és abban a mappában elmentenünk, amelyben a projekt is található. A két fájlt tetszőlegesen elnevezhetjük, lehet például **soros_bemenet.txt** és **soros_kimenet.txt**. A másodikkal nincs semmi dolunk, ebben a fájlból fognak megjelenni azok az adatok, amelyeket a soros vonalon küldünk.

```
// Soros vonalon egyesével beküldött adatok
0x36 0x35 0x35 0x33 0x35 0x2D
// Beállított várakozási idő
wait 100 ms
// Újabb adatok beküldése a soros vonalon
0x33 0x35 0x2D
```

Miután mindezt megadtuk, a képernyön látható mezőket kell kitöltenünk (a Stimulus ablakban), a Data Filename mezőbe a soros_bemenet.txt, a Trace Filename mezőbe pedig a soros_kimenet.txt fájl kerül.

Ha azt szeretnénk, hogy a programunk szimulátorban történő futtatásakor automatikusan alkalmazza a stimulusban beállított műveleteket, a Stimulus ablakban az Advanced parancsra kell kattintanunk, hogy elérjük a speciális funkciókat. Itt generálunk egy *.scl kiterjesztésű fájlt (**Generate SCL file**), és tetszőleges névvel mentünk el, majd az Attach gombra kattintva fűzzük hozzá a stimulushoz.

Ezek után programunk futtatásakor a soros_bemenet.txt fájlból található adatokat küldi be a soros vonalon, és a soros_kimenet.txt fájlból jelenik meg az eredmény.

6.1.10. Vizsgálóeszközök a szimulátorban

A szimulátorprogram vezérelt végrehajtása több vizsgálóeszköz létrehozását és használatát teszi lehetővé. Az alábbiakban ezeket tekintjük át, közülük kettőt, a Watch ablakot és a töréspont használatát már az előzőekben áttekintettük, de a teljesség kedvéért itt is megemlíttük.

6.1.10.1. Watch és töréspont

A töréspont használata a szimulátorban sokkal többet nyújt, mint hardveres hibakereső eszköz esetén. Ugyanis a programmal megvalósított szimulációkor elméletileg tetszőleges számú töréspontot elhelyezhetünk a programba, akár egyszerre is, mert a szimulátor ezt tudja kezelní. A Watch ablakot az eredeti funkciója szerint használhatjuk.

6.1.10.2. Profile

Megjelenítése: a *Debugger* → *Profile* menüre kattintva. Itt két cselekvést kínál fel:

- **Reset Profile** – minden számértéket lenulláz,
- **Display Profile** – megjeleníti az eredményablakot.

Az összes utasítást felsoroló listában megadja, hogy melyik utasítást hányszor hajtottuk végre, és a végén megadja az indulástól a megállásig eltelt időt, a végrehajtott utasítások számát és a számítási teljesítményt MIPS-ben (millió utasítás/másodperc).

6.1.10.3. Simulator Trace

A szimuláció során a nyomonkövetési funkciókat a *Simulator Trace* használatával valósíthatjuk meg. Használatához először futtassuk a programunkat, majd válasszuk a *View* → *Simulator Trace* menüpontot.

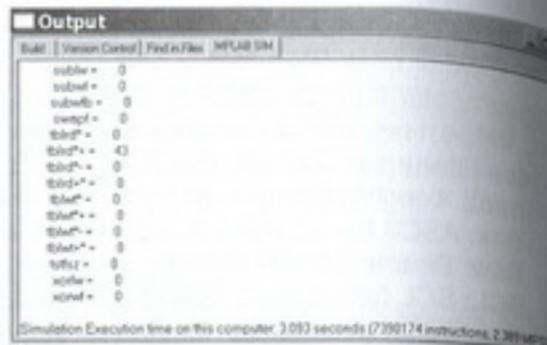
A Trace ablakban megjelenő oszlopokra a jobb egérgombbal kattintva beállíthatjuk, hogy milyen információkat jelenítsen meg a program.

Alapértelmezésként a következőket láthatjuk:

- **Line (Line Number)**: a nyomonkövetés kezdetétől a sorszámok.
- **Addr (Address)**: az utasításhoz tartozó programmemória-cím.
- **Op (Opcode)**: az utasítás numerikus kódja.
- **Label**: a programban használt címkék.
- **Instruction**: disassembly információk, az utasítás visszafordított mnemonikja.
- **SA (Source Data Address)**: a forrásadat címe vagy szimbóluma.
- **SD (Source Data Value)**: a forrásadat értéke.
- **DA (Destination Data Address)**: a céladat címe vagy szimbóluma.
- **DD (Destination Data Value)**: a céladat értéke.
- **Cycles**: a nyomon követés ciklusainak száma.

Az ablak felső felében a jobb egérgombot használva még számos lehetőség közül választhatunk. (Például beállíthatjuk, hogy a program milyen formátumban jelenítse meg a ciklusszámot, vagy elugorhatunk a nyomonkövetési lista adott sorára stb.)

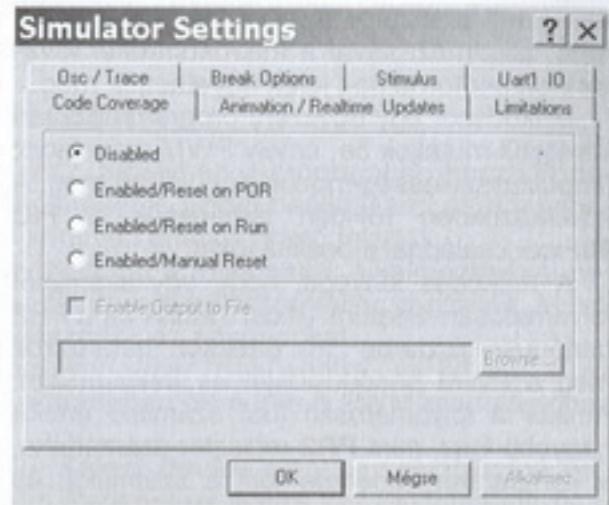
Egy-egy sorra kikkelve az ablak alsó részében az aktuális sorhoz tartozó programrészlet jelenik meg. Lépésenként követhetjük a futás során használt regisztereket, így megkönythető az esetleges hibakeresés (láthatjuk például, hogy a programunk mikor hajt végre NOP utasítást).



6.1.10.4. Code coverage

Ez a tulajdonság az mutatja, hogy éppen melyik kódrészt hajtjuk végre. Ez a Programmemória ablakban van megjelölve. Ez nem ugyanaz, mint a programkövetés (trace):

- **Trace** mutatja a kódot, amit végrehajtunk, és megmondja, hogy mikor hajtottuk azt végre.
- **Code coverage** csak megjelöli a már végrehajtott kódot, de azt nem jelzi ki, hogy mikor történt. Ez úgy működik, hogy tárolja a végrehajtott utasítások címeit, és a következő megállásnál (Step, töréspont, Halt parancs) megjelöli a Programmemória ablakban. Kiválasztása: *Debugger* → *Settings* és kattintás a Code Coverage fülön.

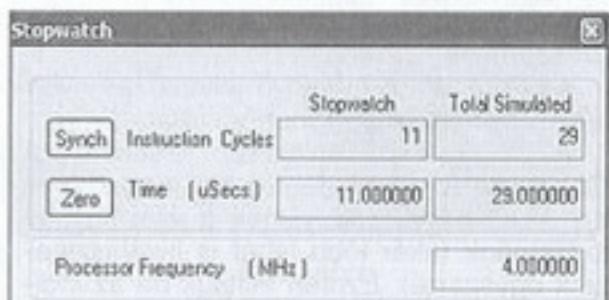


6.1.10.5. Stopwatch – Stopperóra

A stopperóra a szimulátor nagyon hasznos eszköze, használatával programrészletek végrehajtásának idejét, illetve ciklusok számát mérhetjük le. Elsőként a szimulált proceszszor órajelének frekvenciáját kell megadnunk MHz-ben, a beállításhoz töltük ki a *Debugger* → *Settings* → *Osc/Trace* fülön a *Processor Frequency* mezőt. Ezután töréspontok segítségével határozzuk meg azt a programrészletet, amelyet vizsgálunk. Programunkat futtassuk le az első töréspontig, majd a **Zero** gombbal nullázzuk a stopperót, végül futtassuk ismét a programot a következő megállási pontig.

Ekkor az *Instruction Cycles* mezőben kijelzi a nullázás óta végrehajtott ciklusok számát, a *Time* mezőben pedig az eltelt időtartamot (a mértékegységet automatikusan változtatja).

Használata előnyös, ha például egy megszakítás vagy egy késleltetés idejét szeretnénk pontosan beállítani, ez utóbbira láthatunk példát az ábrán.



```

SETUPDELAY
    MOVLW      .3          ; CIKLUSSZÁM = 3
    MOVWF      TEMP
    DECFSZ    TEMP, F     ; KÉSLELTETŐ HUROK
    GOTO      SD
    RETURN
    END       ; A PROGRAM VÉGE
  
```

Ebben a kis programrészletben 3 ciklusos késleltetést valósítottunk meg. Belső órajelet használunk, 4 MHz frekvenciájú működésre konfiguráltuk az oszcillátort. Rövid számítással meghatározható, hogy egy ciklus végrehajtásának ideje 1 µs. A programrészletben a GOTO ellenőrizhetjük, hogy a ciklusmag valóban 11 µs időtartamú. Ha figyelembe vesszük, hogy a késleltetést egy szubrutinhívás (CALL utasítás, 2 ciklus) előzi meg, és a RETURN (2 ciklus) zárja le, összesen 15 µs időtartamot kapunk.

6.1.10.6. Simulator logic analyzer

A logikai analizátor a View menüben található, alkalmazásával a mikrokontroller kiválasztott csatornáinak jeleit figyelhetjük meg.

Működését egy egyszerű programrészleten keresztül mutatjuk be, amely PWM modulációt (impulzusszélesség-modulációt) valósít meg. A következőkben röviden ismertetjük a PIC 18Fxxx családnál a beállításokat.

A működés lényege, hogy egy számlálót folyamatosan futtatunk (most **TIMER 2**), a PR2 periódusregiszterbe írt értékkel határozzuk meg a PWM periódusidejét és frekvenciáját. Amikor a folyamatosan futó számláló értéke nagyobb lesz, mint PR2 regiszter számértéke, a komparátor kimenete törli a számlálót, és magas szintre állítja az RS tárolót.

A CCPR2L regiszterben kell megadnunk a kitöltési tényezővel arányos számértéket. A komparátor előző vezérlő jele ezt az értéket a CCPR2H regiszterbe írja. Ezután az ismét folyamatosan futó számláló értékét már a másik komparátor a CCPR2H értékével hasonlítja össze, kimeneti jele reseteli a tárolót.

Szoftverben a PWM modul használatához a CCPCON regiszter CCP2M3 és CCP2M2 bitjeit kell 1 szintre állítanunk (részletes információ az adatlapokon található).

A jelet az RC1 lábon figyeljük, így a TRISC regisztert kimenetként állítjuk be. A TIMER 2 számlálót a T2CON regiszter segítségével 8 bitesre konfiguráljuk, kinullázzuk, majd elindítjuk a folyamatos számlálást.

A PWM jel periódusidejét a $[(PR2)+1]*4*TOSC * (TMR2 előosztó értéke)$ képlet alapján számíthatjuk ki.

Az RC1 lábon létrejövő jelváltás nyomon követéséhez klikkeljünk a View → Simulator Logic Analyzer menüpontra.

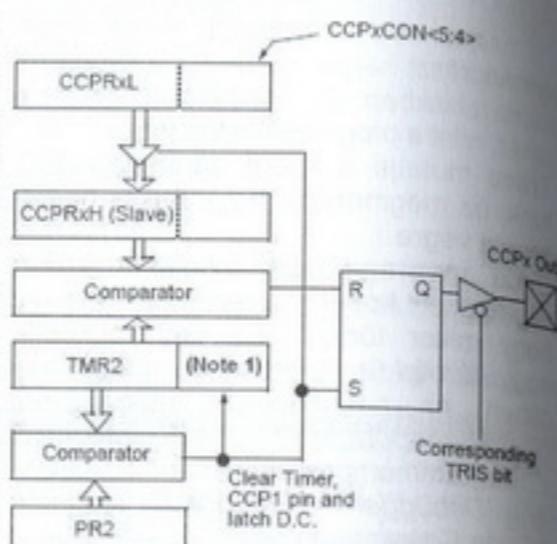
A megjelenő ablakban elsőként a **Channels** gombra kattintva kell megadnunk azt a csatornát, amelyiknek a jelét figyelni szeretnénk (akkor több lábat is kiválaszthatunk egyszerre). Ezután állítsuk be az indítójel pozícióját (*Trigger Position*), amelyet hozzárendelhetünk az utasításszámláló egy értékéhez is (*Trigger PC=*).

A legjobban úgy figyelhetjük meg a változásokat, ha a példaprogramot lépésenként (*Step Into*) vagy animálva (*Animate*) futtatjuk.

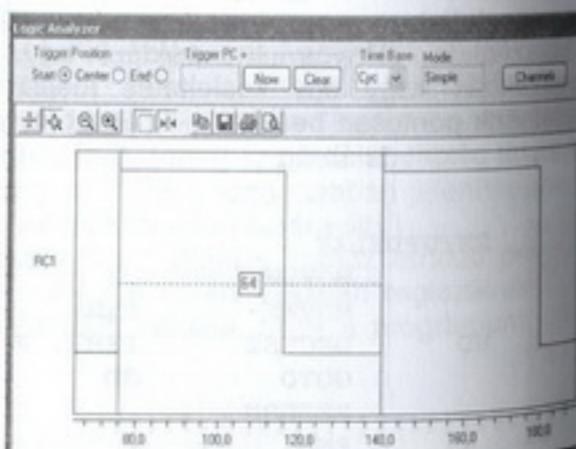
Ha mindezt sikeresen végrehajtottuk, a 6.12. ábrán látható jelalakot kapjuk. A koordinátarendszer x tengelyén az utasításciklusok száma változik.

A logikai analizátor szerszámkészletén a jelre kattintva lemérhetjük a vizsgált jelalak periódusidejét (ciklusok száma).

A szerszámkészlet még számos lehetőséggel segíti munkánkat (nagyítás, a jelalak mentése, nyomtatása stb.).



6.12. ábra
PWM jelalak



6.12. ábra
PWM jelalak

A diagramterületen a jobb egérgombbal kattintva állíthatjuk többek között a diagram tulajdonságait (*Properties*), elmenthetjük a munkánkat (*Export Table*), illetve megnyithatunk egy már korábban mentett jelalakot (*Import Table*).

6.2. VDI – VISUAL DEVICE INITIALIZER

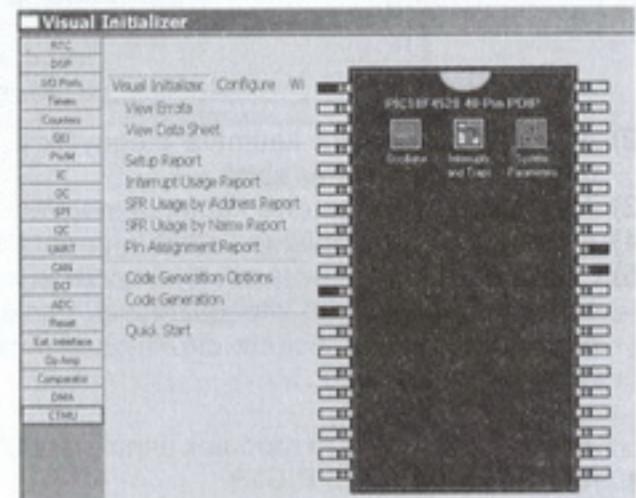
Minden programban szerepel egy inicializáló rész, amelyben a programban használt perifériák regisztereit állítjuk be. A regiszterek felépítéséről, illetve beállításáról az adatlapokon tájékozódhatunk, ami elég időt rabló és könnyen eltéveszthető feladat.

Visual Device Initializer eszköz megkönnyítheti ezt a folyamatot, alkalmazásával néhány párbeszédablak felhasználásával konfigurálhatjuk a mikrokontroller perifériáit. Minden eset megtehetjük anélkül, hogy hosszan bongézsnénk az adatlapokat.

Használatához kattintsunk a Tools → Visual Initializer menüpontra, és kövessük az alábbiakban leírtakat. Példaként végezzük el a korábban bemutatott PWM modulációhoz szükséges inicializálásokat.

A VDI-t elindítva megjelenik a **Configure → Select Device** menüben kiválasztott PIC mikrokontroller ábrája. Első lépésként a **System Parameters** ikonra kell kattintanunk és a **Package Type** legördülő menüben beállítani a kontroller pontos jellemzőit (lábszám, TQFP, QFN vagy PDIP).

Ha változtatunk az alapértelmezett beállításokon a program egy figyelmeztető üzenetet küld, melyben közli, hogy a konfigurálást újra kell kezdenünk. Miután nyugtazzuk az OK gombbal, a 6.13. ábrán látható képernyő jelenik meg. Az ablak két oldalán találhatóak azok az eszközök, amelyek konfigurálását elvégezhetjük.



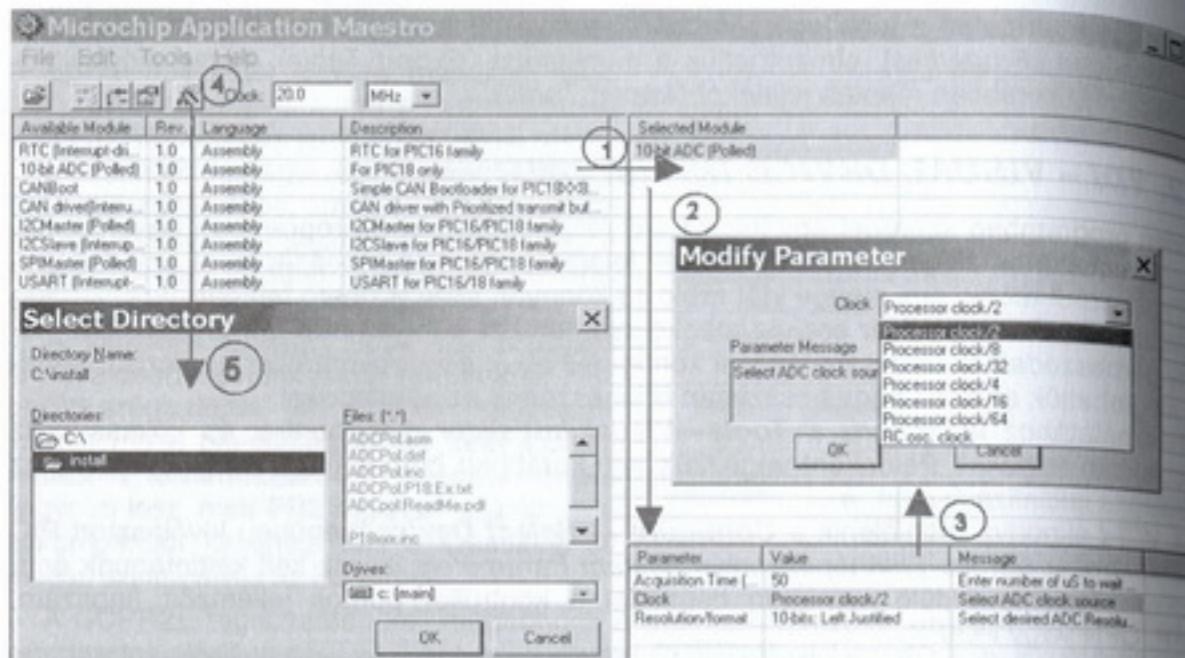
6.13. ábra
VDI grafikus felület

6.3. APPLICATION MAESTRO

A Microchip Application Maestro Software önálló program, amelynek segítségével konfigurálás után a felhasználók be tudják illeszteni a kész, megírt kezelőprogramokat a saját alkalmazásaiiba. A program legfontosabb része a Microchip által a PIC mikrovezérlőkre kifejlesztett modulgyűjtemény. Egy grafikus felületen keresztül a felhasználó kiválasztja a számára fontos modulokat, azután beállítja a paramétereiket.

Ezután az Application Maestro Software a paraméterek felhasználásával generálja a megfelelő kódot. Fontos tudni, hogy ez a program nem az MPLAB IDE része, attól teljesen független. A program az installálás után a Program Files → MpAM könyvtárból indítható az MpAM.exe fárra történő kattintással.

1) Az Available Module ablakban lévő modulokból választhatunk, a megfelelő modulra rákattintva, és lenyomott egérgombbal áthúzva a Selected Module ablakba (akkor több modult is).



6.14. ábra
Application Maestro használata

- 2) Az áthúzott modulra kattintva megnyílik a hozzá tartozó, változtatható paramétereket tartalmazó *Parameter* ablak.
- 3) A megfelelő paraméterre kattintva megváltoztathatjuk a paraméter értékét (*Value*).
- 4) A Generate Code ablakra kattintva generálhatjuk le a beillesztendő kódot,...
- 5) ... ami az általunk kiválasztott könyvtárba kerül. Itt lesznek a modulhoz tartozó információs fájlok .txt és .pdf kiterjesztéssel. Ezekben vannak leírva a hívható makrók és rutinok. Az egymásra include direktívával hivatkozó .asm, .def, .inc fájlok tartalmazzák a felhasználható kódot.

Jelenleg a következő modulok állnak rendelkezésre:

- I2C Slave for PIC16/PIC18
- Simple CAN Bootloader for PIC18
- RTC for PIC16 family
- 10-bit ADC polled for PIC18
- 10-bit ADC interrupt for PIC18
- Can driver with prioritized transmit buffer
- I2C Master for PIC16/PIC18
- SPIMaster for PIC16/PIC18
- USART for PIC16/PIC18
- Simple SRAM Dynamic Memory Allocation
- EUSART based for PIC18
- ECAN routines for PIC18+ECAN
- DeviceNet Group 2 Slave for PIC18
- CAN from PIC18Fxx8
- LCD C routines for PIC18
- LCD routines for PIC16/PIC18
- SPISlave for PIC16/PIC18
- Oversampling module for PIC16/PIC18

6.4. KÓDMODUL KÖNYVTÁR (CODE MODULE LIBRARY)

A kódmodulkönyvtár, *Code Module Library* (röviden CML) olyan alkalmazói program, amivel kód részleteket és egyéb szövegeket kezelhetünk. A több helyen is használható kód részleteket könyvtárakba rendezzük, hogy gyorsan és könnyen ki tudjuk választani a megfelelőt.

A könnyű használatot az is biztosítja, hogy bármely Windows-alapú szövegszerkesztővel használható, ami akár az MPLAB IDE szerkesztője is lehet. A kiválasztott kód részletet a vágólapra másoljuk, és utána beillesztjük az alkalmazásba.

CML-be már számos Microchip-kód részlet készén megtalálható. Ezek bármikor módosíthatók, és újabbak is beilleszthetők. Két fórum is segíti a fejlesztést: az egyikben a kódokat cserélgetik egymással a programozók, míg a másik fórum a Microchiphez érkező visszajelzések kezelésére szolgál. Ezek a fórumok a <http://forum.microchip.com> oldalon találhatók. A gyakorlottabbak további lehetőségeket használhatnak: változókat lehet deklarálni a kód részletekben, és a „Find and Replace” funkcióval könnyen kicserélhető az alkalmazásunkban használt névre.



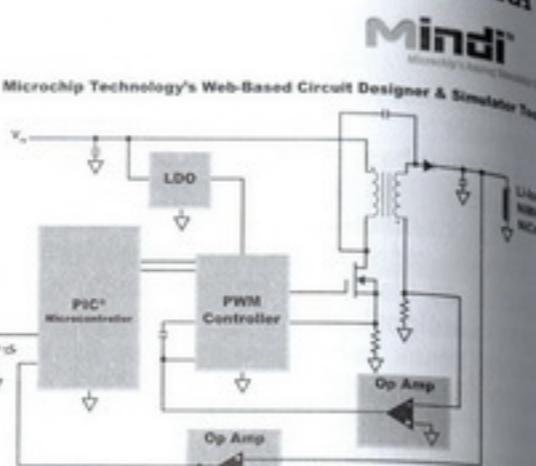
6.15. ábra
A CML képernyőképe

A kódmodul-könyvtárral kapcsolatos további ismeretek a DVD-melléklet /cml könyvtárában, illetve a www.microchip.com/cml oldalon találhatók.

6.5. MINDI – A MICROCHIP WEBALAPÚ ANALÓG SZIMULÁTORA

Microchip's Mindi™ szimulátora segít a tokok fogyasztását kezelő vagy különböző analóg áramkörök tervezésében és analízálásában. Segítségével gyorsan lehet áramköri rajzokat generálni, az áramköröket szimulálni, és ezek alapján az áramkörök passzív elemeit értékre és teljesítményre meghatározni. A Mindi™-vel fejlesztett áramkörök ezek után számítógépre letölthetők, és akár közvetlenül az áramköri terv részévé válhatnak.

Az eszköz használatához regisztrálni kell a www.microchip.com/mindi oldalon, ami után letölthető az eszköz. 2008 végén már öt témakörhöz léteztek segítő megoldások.



6.16. ábra
MINDI

A MINDI fejlesztőeszközzel kapcsolatos további ismeretek a DVD-melléklet /mindi könyvtárában, illetve a www.microchip.com/mindi oldalon találhatók.

6.6. MAPS – MICROCHIP ADVANCED PRODUCT SELECTOR

A Microchip által gyártott termékek nyomtatott adatlapjai vagy katalógusok helyett ezek elektronikus formában letölthetők az internetről. A 2.0 verziótól kezdődően a program lehetővé teszi, hogy a felhasználó kiválaszthassa a neki legjobban megfelelő Microchip-gyártmányt.

A program nem csupán egy elektronikus adatbázis, hanem interaktív grafikus felhasználói felületet is biztosít a parametrikus kereséshez, amivel a termékek egymás mellettől összehasonlíthatók. Mivel a programhoz tartozó nagy mennyiségű adat gyorsan változik, ezért a Microchip lehetővé tette az interneten keresztül történő használatát.



- Analog
 - Op-amps, sensors, controllers
 - Prefix: MCPxxxx, Txxxx
- Memory
 - Serial and Parallel EEPROMs
 - Prefix: 24xxxx, 25xxxx, 93xxxx
- Microcontrollers
 - 8-bit & 16-bit Microcontrollers (MCU) and Digital Signal Controllers (DSC)
 - MCU Prefix: PICxxxx
 - DSC Prefix: dsPICxxxx
 - Wireless Prefix: rfxxxx, MRFxxxx
- Global Part Search
 - Part lookup of MCP and competitive database
 - Enter any of the product prefix's or part number

6.17. ábra
MAPS

7. HARDVER FEJLESZTŐ-ESZKÖZÖK PIC MIKROVEZÉRLÖKHÖZ

A mikroprocesszoros fejlesztéshez feltétlenül szükség van két hardvereszközre:

- Az egyik egy programozó-hibakereső (debugger) eszköz, amivel a programkódot a fejlesztendő eszköz programmemoriájába tudjuk juttatni, és ott a kódot futtathatjuk, és felderíthetjük az esetleges hibákat. Ezek a programozók, a debugerek és az emulátorok.

- A másik a fejlesztendő termék funkcióinak kipróbálását lehetővé tévő összerakott áramkör, a „deszkamodell”.

Ez utóbbit természetesen a fejlesztő is kialakíthatja, és az áramköri tervezés után megrajzolhatja a nyomtatott áramkört, azt elkészítheti, utána beültetheti, az áramkört felélesztheti. Utána kezdődhet a tényleges fejlesztőmunka. Azonban ez elég költséges megoldás, még ha csupán a deszkamodell elkészítéséhez szükséges mérnöki munka órabérét nézzük is.

A Microchip ezt felismerve számos előre elkészített fejlesztői panelt kínál: ezek alkalmasára nyilván költségkimelőbb megoldás, mint az egyedi elkészítés. Ezek a panelek – összehasonlítva az egyedi fejlesztés költségeivel – jóval olcsóbbak. Mivel a témával már régóta foglalkozó fejlesztőmérnökök tervezik, ezért valószínűleg színvonalasabbak, mint ha a tématerülettel ismerkedő kezdő teszi ezt.

A következőkben ezeket a hardveres fejlesztői eszközöket tekintjük át.

7.1. ICD2

Mivel az emuláció tulajdonképpen ellenőrzött lépésekkel programvégrehajtásnak tekintető, akár minden tokba beleintegrálhatjuk ezt a nem túl bonyolult áramkört, amelynek segítségével a tok lényegében képes a legfontosabb emulációs tevékenység végrehajtására.

A Microchip esetén ezt a megoldást elsőként a PIC16C87X tokokba integrálták bele, az ezt felhasználó fejlesztőeszköz az ICD (*In-Circuit Debugger*). Mivel egyszerűségükönél fogva ezek az eszközök számítógépről történő vezérlést és felhasználói felületet igényelnek, teljesen kézenfekvő volt a Microchip megoldása, hogy az eszközt az MPLAB környezetbe integrálta.

7.1.1. A debugger működése

Hogyan tudjuk egy programunk helyes működését ellenőrizni? Úgy, hogy bármely utasításánál meg tudjuk állítani a program futását (töréspont), és ki tudjuk olvasni az őszes változóinak értékét, regisztereinek a tartalmát.

A törésponti működés úgy valósítható meg, hogy egy több-bites digitális komparátor-áramkör egyik bemeneti oldalára az utasításszámító bitjeit, míg a másik bemeneti oldalára egy, a debug programból feltöltött ún. törésponti regiszter bitjeit kapcsoljuk. A két oldal egyezésekor teljesül a törésponti feltétel, és a komparátor ilyenkor megjelenő kimeneti jelét felhasználva meg lehet állítani a program futását.

A program törésponton történő megállása után indul el a tok programmemoriájának utolsó 256 szavában elhelyezkedő debug program, ami lehetővé teszi új törésponti cím beírását, illetve lehetőség van az adatmemória elérésére, vagyis olvasására/írására is. Mindez az MPLAB kezelőfelületén keresztül valósul meg soros kommunikáció felhasználásával.

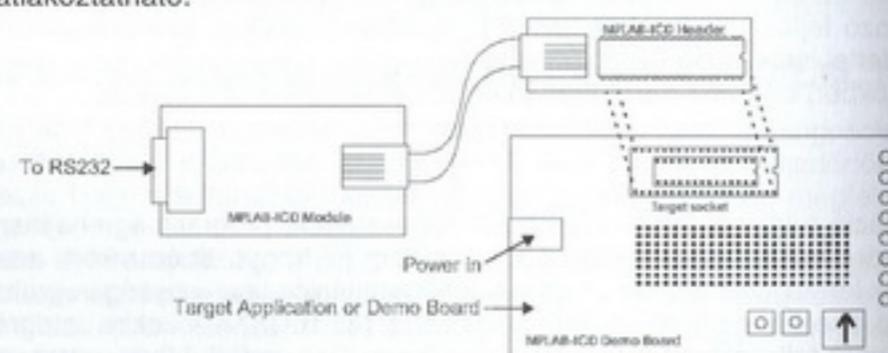
Nagy ötlet, hogy mind a tok programozásához, mind a debug információk átviteléhez az RB6, RB7 lábakat használjuk fel!

A röviden vázolt működés két regiszter, az **ICKBUG** (cím: 18EH) és **BIGBUG** (cím: 18FH) felhasználásával valósul meg. A <BKA12:BKA0> elnevezésű 13 bites törésponti cím alsó 8 bitjét a BIGBUG regiszter, a felső 5 bitjét az ICKBUG regiszter alsó 5 bitje tárolja. A belső debugger áramkör vezérlése a regiszter felső három bitje segítségével történik. Ezek elnevezése és működése a 6. bittel kezdve:

- **INBUG:** a debug program végrehajtása alatt az értéke 1.
- **FREEZ:** ha az értéke 1, akkor a TMR0, TMR1 és TMR2 számlálók és előosztóik értéke „befagy”, és ugyanez történik az SSP, A/D és az USART belső állapotával is. Ezért törésponti megálláskor ezek adott pillanatnyi értéke is kiolvasható.
- **SSTEP:** a lépésenkénti működés végrehajtását vezéri. A normál programvégrehajtás-hoz való visszatéréskor (INBUG = 0), ha SSTEP = 1, akkor egy programutasítás végrehajtását teszi lehetővé.

7.1.2. Az ICD használata

Az ICD modul a PIC-hez kapcsolva működőképes. A PIC-kel való kommunikáció egy 6 pólusú telefoncsatlakozón keresztül történik, amelynek a kivezetései a tok **VDD**, **GND**, **MCLR**, **RB3**, **RB6**, **RB7** elnevezésű lábaihoz kapcsolódnak. Egyik megoldásként a fejlesztett rendszerben kell egy 6 pólusú telefoncsatlakozót elhelyezni, és a PIC lábat oda kökötni. Másik megoldásként kapható a fejlesztendő áramkörbe csatlakozást biztosító ICD-fej. A PC soros vonala az ICD modulhoz csatlakozik, és a modul a fejjel (és a rajta lévő PIC16F87X vagy PIC18FXXX tokkal) az áramkörben kialakított 28 vagy 40 lábú DIP foglalatba csatlakoztatható.

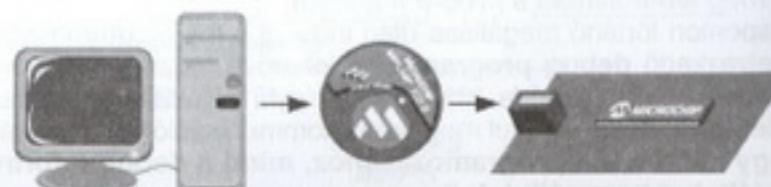


7.1.3. ICD2 debugger

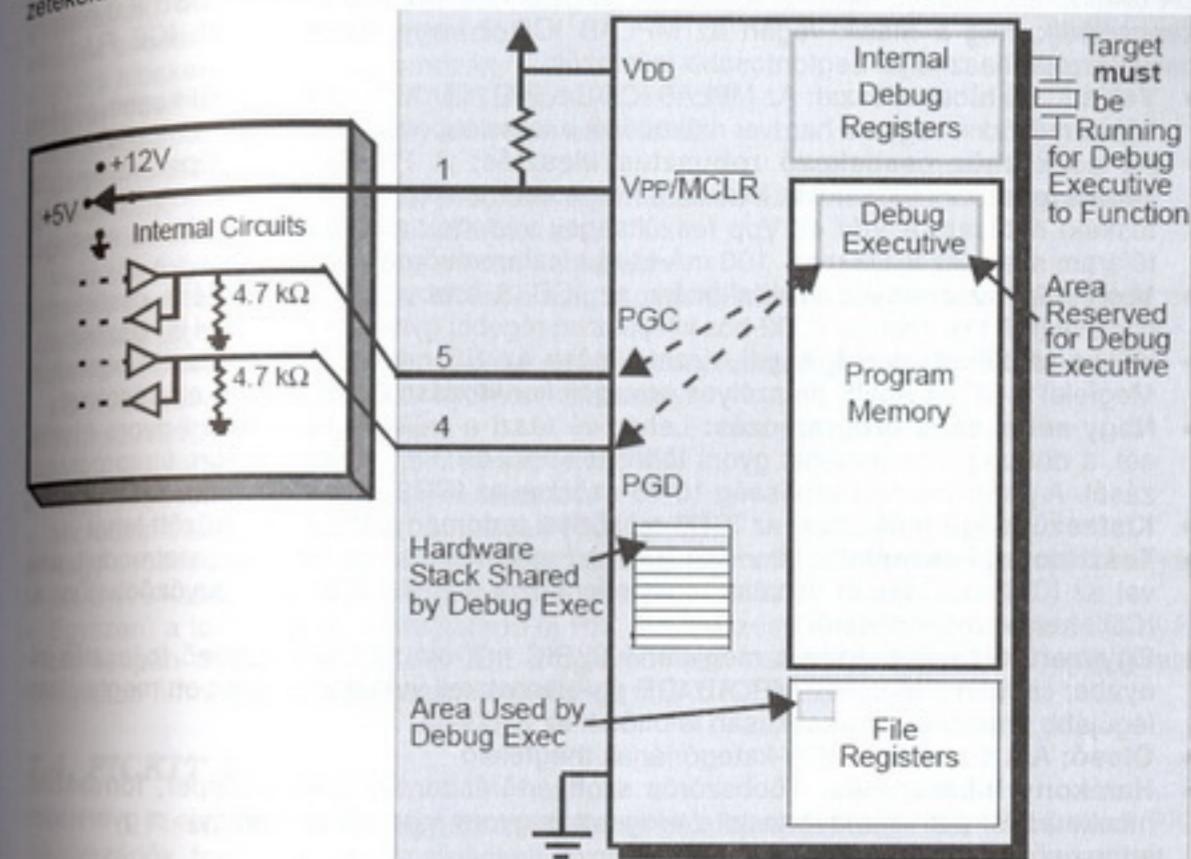
Először az ICD-t a PIC16F87X tokokkal való együttműködésre alakították ki. A 18-as család megjelenésével lépett színre az ICD-t felváltó **ICD2 fejlesztő**, amely képes az ICD eredeti funkcióinak megtartása mellett a PIC18FXXX család fejlesztésére is.

Az érdekesen formatervezett, korong alakú egységet elsődlegesen a PIC18FXXX típusok programfejlesztésére és programozására alkották meg, amely nemcsak soros porton, hanem nagy sebességet biztosító **USB** (2 Mbit/s) csatornán is tud kommunikálni a számítógéppel. A működtető programjának új verziót képes önmagában frissíteni. Képes 2,0–6,0 V közötti működésre. Hárrom LED található az egységen:

- **Power** – a tápfeszültség meglétét jelzi,
- **Busy** – működés közben világít,
- **Error** – hibát jelez.



A DIP tokos PIC típusok felprogramozására kapható az ICD2-höz illeszkedő univerzális programozómodul (AC162049). A programozáshoz használt hét kivezetést átkötő vezetékekkel lehet a 40 lábú programozófoglalat megfelelő kivezetéseihez csatlakoztatni.



7.1. ábra

Az ICD modulok működéséhez nélkülözhetetlen a PIC-be letöltendő debug program, aminek működéséhez debug regiszterek és veremterület is szükséges!

7.1.4. Korlátozások az ICD használatakor

Az ICD a működéséhez adatmemóriát használ fel a változónak, illetve programmemóriát a DEBUG programmodul elhelyezésére. Ezért az ICD használatakor néhány, nem igazán zavaró megszorítást be kell tartanunk:

- a 0-s (RESET) címen NOP utasításnak kell lennie,
- 1, illetve 2 veremszintet felhasznál,
- használja a programmemória utolsó területét,
- 6–14 bájt adatmemória-helyet,
- továbbá az RB6 és RB7 lábakat, ezért csak a PORTB 0–5 bitje használható.

Az ICD2 debuggerrel kapcsolatos további ismeretek a DVD-melléklet /ICD2 könyvtárában, illetve a www.microchip.com/ICD2 oldalon találhatók.

7.2. ICD3

Megjelent az ICD2 utódja, a kinézetre hasonló, nagymértékben továbbfejlesztett eszköz, a DV164035 azonosítójú **MPLAB ICD 3**. Az eszköz a PC nagy sebességű USB 2.0 portjára kapcsolódik, míg a másik végén az MPLAB ICD 2 vagy MPLAB REAL ICE RJ-11-es csatlakozóját használja. Legfontosabb jellemzői:

- Valós idejű hibavadászat:** Az MPLAB ICD 3 a PIC mikrovezérlök maximális sebességével képes működni, vagyis a hardver működését a teljes sebességén tudjuk nyomon követni.
- Az eszközhöz csatlakozó robusztus illesztés:** A PIC-hez csatlakozó áramkörök védelemmel vannak ellátva, kiküszöbölte a céláramkör felől jövő zavarok (feszültség-tükék) hatásait. A Vdd és Vpp feszültségek védettek a túlfeszültség és az esetleges túláram ellen. Az ICD3 max. 100 mA-t tud a céláramkörnek biztosítani.
- Microchip szabványos csatlakozás:** az ICD 3 szabványos hibavadász csatlakozást használ (RJ-11), ezért az ICD2-höz kifejlesztett régebbi gyakorlópanelekkel is használható.
- Külön táplálást nem igényel,** áramellátása az SB csatlakozón keresztül biztosított. Megfelel a CE és RoHS (veszélyes anyagok korlátozása a termékben) előírásoknak.
- Nagy sebességű programozás:** Lehetővé teszi a működtető firmware gyors frissítését, a debug programmodul gyors töltését a PIC-be, illetve annak gyors újraprogramozását. A programozási sebesség 10-15-szöröse az ICD2 sebességének.
- Kifeszültségű működés:** az ICD3 működési tartománya 2,0–5,5 V között lehet.
- Tesztmodul használata:** minden ICD3-hoz mellékelnek egy hardver tesztmodult, amivel az ICD3 csatlakozó vonalait lehet ellenőrizni, vagyis minden meggyőződhetünk az ICD3 helyes működéséről.
- Egyszerű a továbblépés** a megjelenő új PIC mikrovezérlőkkel történő fejlesztés irányába: csupán a legújabb MPLAB IDE programot kell installálni, és az ott megtalálható legújabb firmware automatikusan letöltödik az ICD3-ba.
- Olcsó:** Ára a szokásos ICD-kategóriának megfelelő.
- Hatókony hibakeresés:** többszörös szoftvertöréspont-kezelés, stopper, forrásszintű hibakeresés, ami lehetővé teszi a programok gyors módosítását, vagyis a gyors hibakeresést.
- Támogatott PIC mikrovezérlök:** Az ICD3 támogatja a legtöbb Flash memóriájú PIC, illetve dsPIC mikrovezérlő fejlesztését.

Az ICD3 debuggerrel kapcsolatos további ismeretek a DVD-melléklet /ICD3 könyvtárában, illetve a www.microchip.com/ICD3 oldalon találhatók.

7.3. REAL ICE EMULÁTOR

Az emulátorok képesek az egyes utasításokat egyenként, egy belső, beépített áramkör segítségével végrehajtani. Ilyenkor az emulátor IC tok lábkiosztást formázó csatlakozójai (ezt POD-nak szokták hívni) helyezzük a fejlesztendő áramkörben a processzor foglalatába. Ezek után az emulátorba letöltött program már ellenőrzött körülmenyek között futatható, hasonlóan az MPLAB szoftver szimulátorához.

Az MPLAB **REAL ICE** In-Circuit Emulator System eszközt a Microchip Flash DSC (Digital Signal Controller) és MCU (Micro Controller Unit) mikrovezérlőinek fejlesztéseinek használhatjuk. Hibát lehet keresni a segítségével, illetve programozni az MPLAB IDE grafikus környezete felhasználásával.

Az MPLAB **REAL ICE** a PC-hez nagy sebességű USB 2.0 illesztésen keresztül kapcsolódik, vagy az ICD2-nél bevezetett 6 pólusú RJ11-es csatlakozón, vagy egy új, nagyobb sebességet biztosító, zajtűrő differenciális jelet használó CAT5-ös csatlakozáson keresztül.

Az MPLAB REAL ICE belső programja folyamatosan frissül az új mikrovezérlök megjelenésével, és az MPLAB minden újabb verziója tartalmazza ezt a frissítést.

Milyen előnyei vannak a REAL ICE használatának?

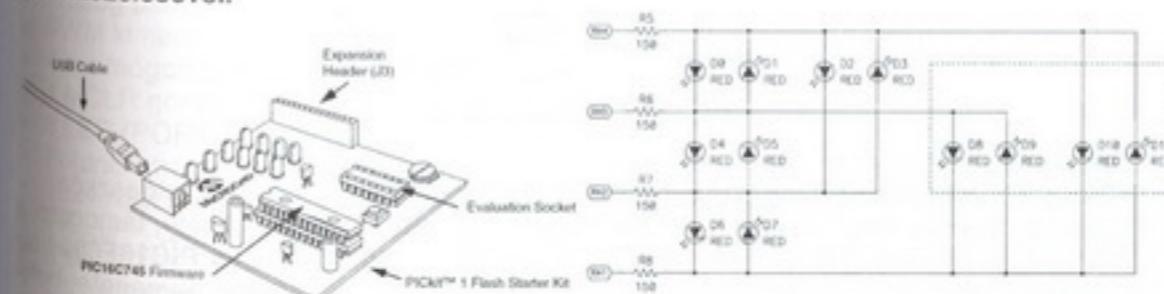
- kisméretű, MPLAB IDE környezetbe integrált és olcsó eszköz;
- teljes sebességgel történő emuláció;
- gyors hibakeresés és programozás;
- zavarvédett csatlakozás a REALICE és a PIC között;
- nagy sebességű átvitel, amely akár 3 m-es is lehet.

Legfontosabb jellemzői:

- Valós idejű (teljes sebességű) programvégrehajtás és futási adatok (*trace*) gyűjtése.
- Futási idők mérése stopperrel (*Stopwatch*).
- Változók folyamatos figyelése a watch ablakban.
- Teljes hardver hibavadászat: töréspontok, lépésekbeni futtatás, változók figyelése és módosítása;
- A PIC-hez csatlakozó vizsgálóvezetékek irányának állítása, figyelése;
- I/O kivezetések változásainak nyomon követése;
- Túlfeszültség-, túláramvédelem;
- Működési tartománya 2,0–5,5 V között lehet;
- Nagy sebességű USB 2.0 kommunikáció;
- Microchip szabványos ICD2 csatlakozó (RJ11);
- Nagy sebességű vizsgáló adó-vevő csatlakozás: Egy adó és egy vevő áramkori kártyát két CAT5 kategóriájú kábellel. Segítségével a hibavadászat zavarvédettebb.
- Egyszerű a továbblépés a megjelenő új PIC mikrovezérlőkkel történő fejlesztés irányába: csupán a legújabb MPLAB IDE programot kell installálni, és az ott megtalálható legújabb firmware automatikusan letöltödik a REAL ICE eszközbe.

7.4. PICKIT 1

PICKIT™ 1 Flash Starter Kit egy könnyen kezelhető fejlesztőeszköz 8/14 lábú flash PIC mikrovezérlök fejlesztésére. A mellékelt mintapéldákon a használó megismeredhet a bekimenetek kezelésével, az A/D átalakító és az analóg komparátorok, időzítők, adattáblák kezelésével.



7.2. ábra
PICKIT fejlesztőpanel

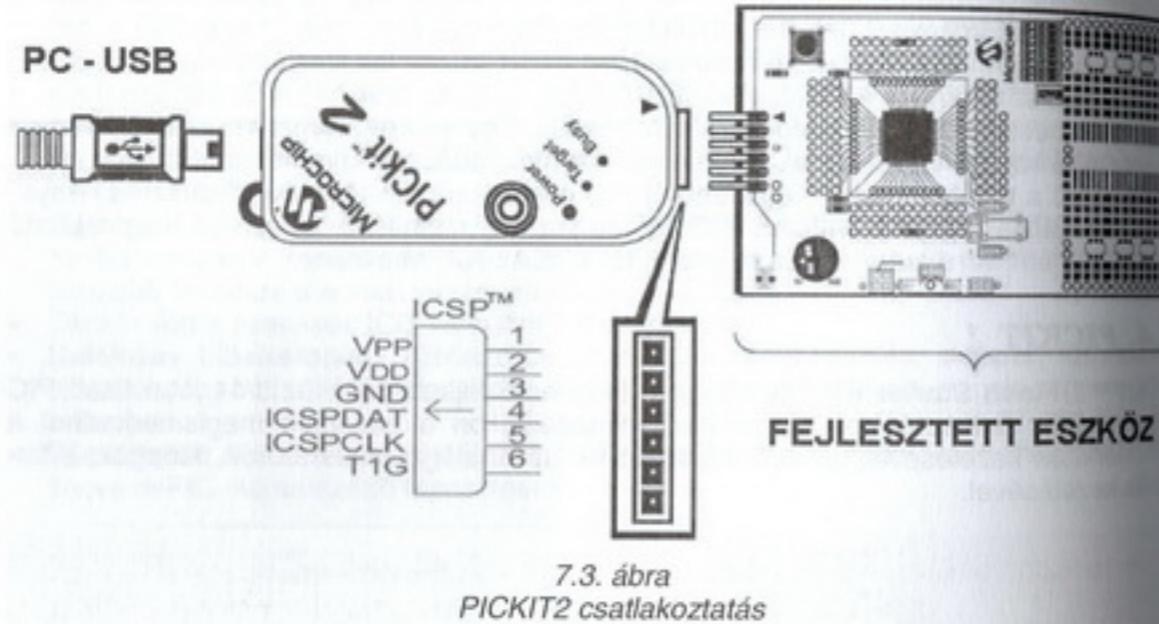
A fejlesztő egy nyomtatott áramkori lapra van szerelve. USB-n keresztül kapcsolódik a PC-hez, illetve az MPLAB-hoz. Külön tápegysége nincs, az 5 V-os működési feszültséget az USB portról kapja. A működtető programja egy PIC16C745-ös USB perifériával is rendelkezik, ami a körülbelül 12 LED, egy nyomógomb és egy analóg potenciometterrel rendelkezik. Beültethető soros port és tetszőlegesen felhasználható „csupalyuk” terület is található a lapon. Egy bővíthető csatlakozóra a tápfeszültséget és a be/kimeneti lábakat is kivezették.

Érdekes a LED-ek vezérlése: az I/O lábak közé LED-eket kötve kihasználhatjuk, hogy a lábaknak három állapota lehetséges: kimenetként 0 vagy 1, míg bemenetként definiálva a lábon nagyon kicsi áram folyik át. Ez a megoldás 12 LED meghajtását biztosítja, 4 I/O láb felhasználásával. (Például RA4 = 1 és RA5 = 0 esetén D0 jelű LED fog világítani.)

A PICKit1 áramkörrel kapcsolatos további ismeretek a DVD-melléklet /PICKIT1 könyvtárában, illetve a www.microchip.com/PICKIT1 oldalon találhatók.

7.5. PICKIT 2

A PICkit 2 nagyon olcsó, jól használható fejlesztőeszköz. Segítségével a legtöbb flash memoriás 8/16/32 bites PIC mikrovezérlöt, valamint Microchip EEPROM-okat is képes programozni, illetve a debug képességét kihasználva programot fejleszteni. (Futtatás-megállítás törésponton, lépésekkel utasítás-végrehajtás.)

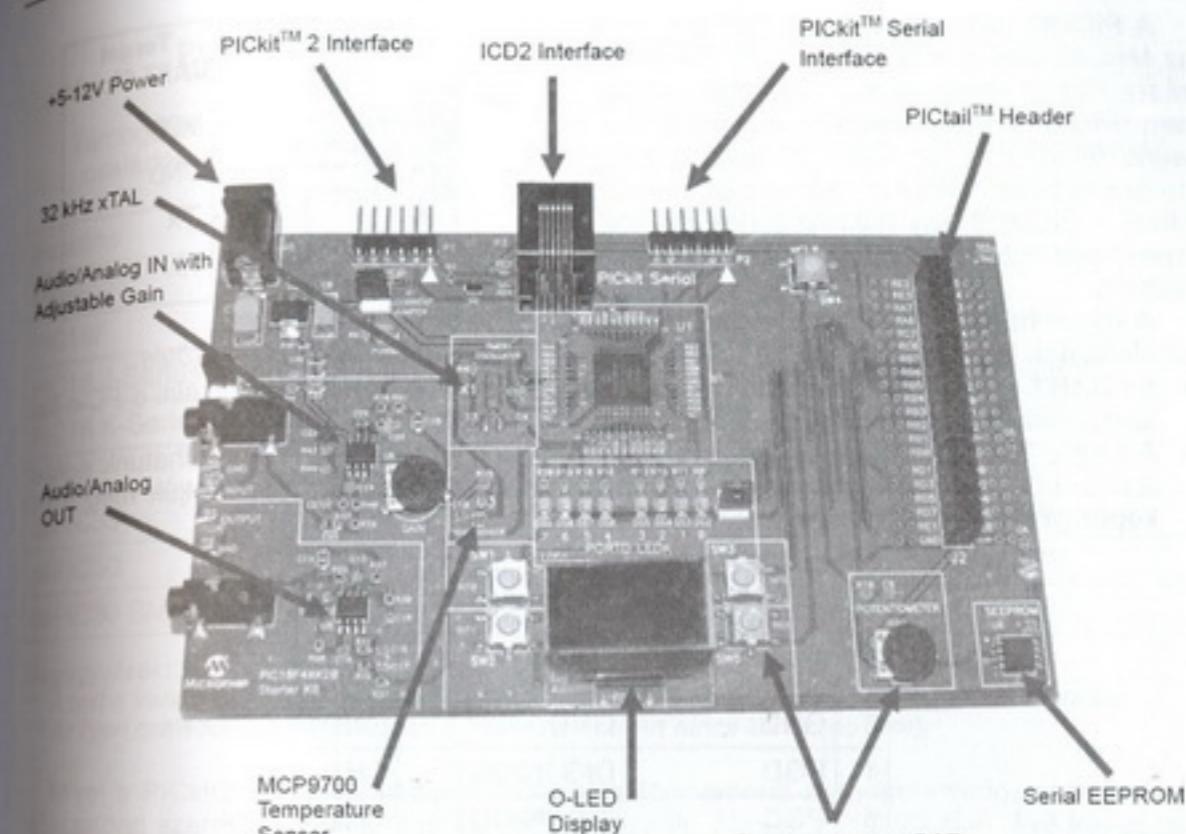


7.3. ábra
PICKit2 csatlakoztatás

Több kialakításban kapható:

- **PICkit 2 Development Programmer/Debugger:** önállóan maga a fejlesztőeszköz.
- **PICkit 2 Starter Kit:** itt mellé csomagolnak egy próbápanelt, amely támogatja a 8/14/20 lábú, kis lábszámú mikrovezérlök fejlesztését, a panelen egy PIC16F690 típusú mikrovezérlő eszköz található. Perifériaként 4 LED és egy analóg bemenetet biztosító potenciometré használható.
- **PICkit 2 Debug Express:** itt is mellé csomagolnak egy próbápanelt, a panelen egy PIC16F887 típusú mikrovezérlő eszköz található. Perifériaként 8 LED, egy nyomógomb és egy analóg bemenetet biztosító potenciometré használható.
- **PICF4XK20 Kit:** Ez a kialakítás kimondottan a 18-as család használatának elsajátítására szolgál. Itt a próbápanel mellé adják a PICkit2-t. A panelen egy PIC18F46K20 típusú mikrovezérlő eszköz található. A panel kialakítása és a rajta lévő perifériák a 7.4. ábrán láthatók. A panelhez szintén tartoznak mintapéldák, de ezek már C-ben íródtak. A mellékelt demók mutatják be az OLED kijelző, a belső RTCC óra és a digitális szűrés használatát.

7. fejezet: Hardver fejlesztőeszközök PIC mikrovezérlőkhöz



7.4. ábra
PICF4XK20 gyakorlópanel

A PICF4XK20 kártyán a következők találhatók:

- 128x64 Organic LED Display (SPI)
- 32.768 kHz külső oszcillátor (Timer1)
- Analog bemeneti szűrő és erősítő (RE1)
- PWM kimeneti szűrő (RC2)
- 4 nyomógomb (PORTB0-PORTB3)
- 1 MCLR gomb (RESET)
- 8 LED (PORTD)
- Potenciometér (RE0-AN5)
- 1024 KBit soros EEPROM
- PICtail™ csatlakozó
- 6 lábú ICSP™ programozó tüskecső
- 6 lábú PICkit soros analizátor interfész
- Árammérő-átkötés
- RJ-11 ICSP programozó aljzat.

A PICF4XK20 Kit fejlesztőeszközzel kapcsolatos további ismeretek a DVD-melléklet /picf4xk20_kit könyvtárában találhatók.

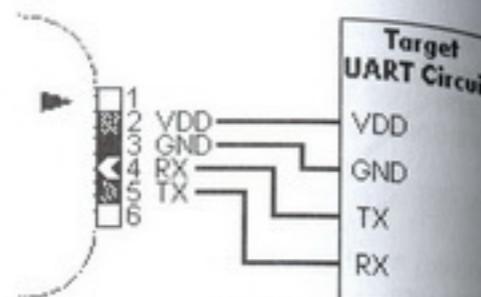
A verzióhoz 12 leckéből álló tanulnivalót mellékelnek, és a gyakorlópanelen azonnal kipróbálható. Ezzel a C programozás alapjait lehet elsajátítani, mivel az összes forráskód a magyarázatokkal rendelkezésre áll. A főbb témaik: I/O, A/D átalakítás, időzítők, megszakítások, adattáblák kezelése.

A PICkit2 programozóként, debuggerként az MPLAB-ban is használható, de rendelkezésre áll egy Windows alatt futó önálló program (**PICkit 2 Programmer**), aminek elsőleges feladata a legtöbb PIC mikrovezérlő programozása MPLAB környezet nélkül. Mivel a PICkit 2 egy mikrovezérlős eszköz, ezért más feladatok megoldására is használható.

A könyv írásakor a PICkit2 három további alkalmazása lehetséges:

- Az **UART Tool** programot betölve képes egy soros vonalon kommunikálni a PC-n futó terminálszerű programmal (7.5. ábra).
- A **Logic Tool** program segítségével logikai be- és kimenetként használhatunk 4 lábat (LOG. I/O), valamint logikai jelek (3 csatorna) időfüggvényét jeleníthetjük meg a PC képernyőjén. A bekötést az alábbi táblázat tartalmazza.

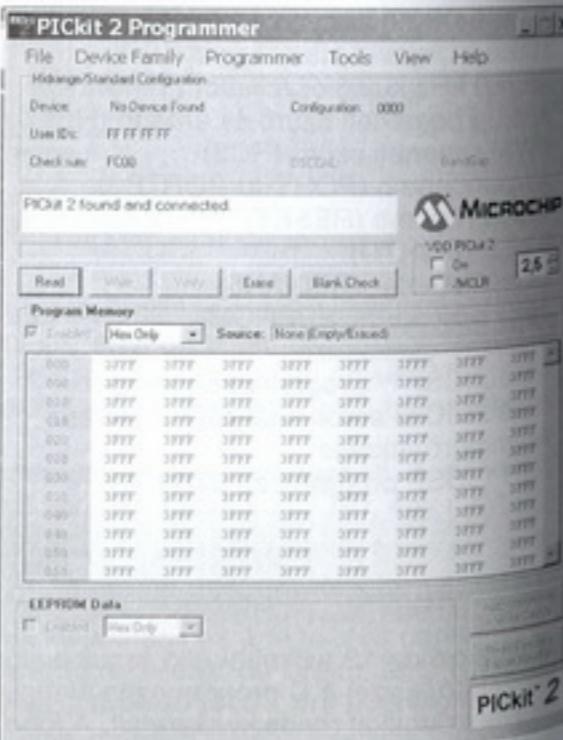
| # | PRG/DBG | LOG. I/O | ANA. |
|---|----------|------------|------|
| 1 | VPP/MCLR | DIG.OUT | - |
| 2 | VDD | VDD | VDD |
| 3 | GND | GND | GND |
| 4 | PGD | DIG IN/OUT | CH1 |
| 5 | PGC | DIG IN/OUT | CH2 |
| 6 | AUX | DIG IN/OUT | CH3 |



7.5. ábra
PICkit 2 UART-bekötés

• A **Programmer-To-Go** támogatással mikrokontrollereket programozhatunk MPLAB nélkül. A PICkit2 Programmer-To-Go funkciója lehetővé teszi, hogy egy programozandó PIC tartalmát a PICkit2-be letöltsük azért, hogy ezt később írjuk be az eszközbe. Ehhez csupán csak egy PICkit2-nek 5V/100mA táplálást biztosító USB csatlakozásra van szükség, ami teleppel is megvalósítható. Mivel a PICkit2-ben lévő, a letöltendő program ideiglenes tárolására szolgáló memória véges, ezért a következő oldalon található táblázatban közzöljük a korlátokat.

7.6. ábra
PICkit 2 programozó



PROGRAMMER-TO-GO ÁLTAL TÁMOGATOTT ESZKÖZÖK

| Támogatott családok | Támogatott részek | Programmemória méretkorlátozások ** | |
|---------------------|---|-------------------------------------|------------------------------------|
| | | Bájtok szerint | A maximálisan használt cím szerint |
| Baseline | | *** | *** |
| Midrange | | *** | *** |
| PIC18F | Az összes típus, amit a PICkit2 Programmer alkalmazással programozni lehet. Ez az alkalmazás Help → ReadMe menüjében található. | 107264 bájt | 0x1A2FF |
| PIC18 J-Series | | 111872 bájt | 0x1B4FF |
| PIC18 K-Series | | 107264 bájt | 0x1A2FF |
| PIC24 | | 106560 bájt | 0x1157F |
| dsPIC33 | | 106560 bájt | 0x1157F |
| dsPIC30 | | 103872 bájt | 0x10E7F |
| dsPIC30 SMPS | | *** | *** |

Megjegyzések:

** A tokba írandó EEPROM adatok, UserID és konfigurációs memória is bele számít a méretbe.

*** Az ilyen családoknak nagyon kicsi a memóriája, ezért nincs korlát (no limit).

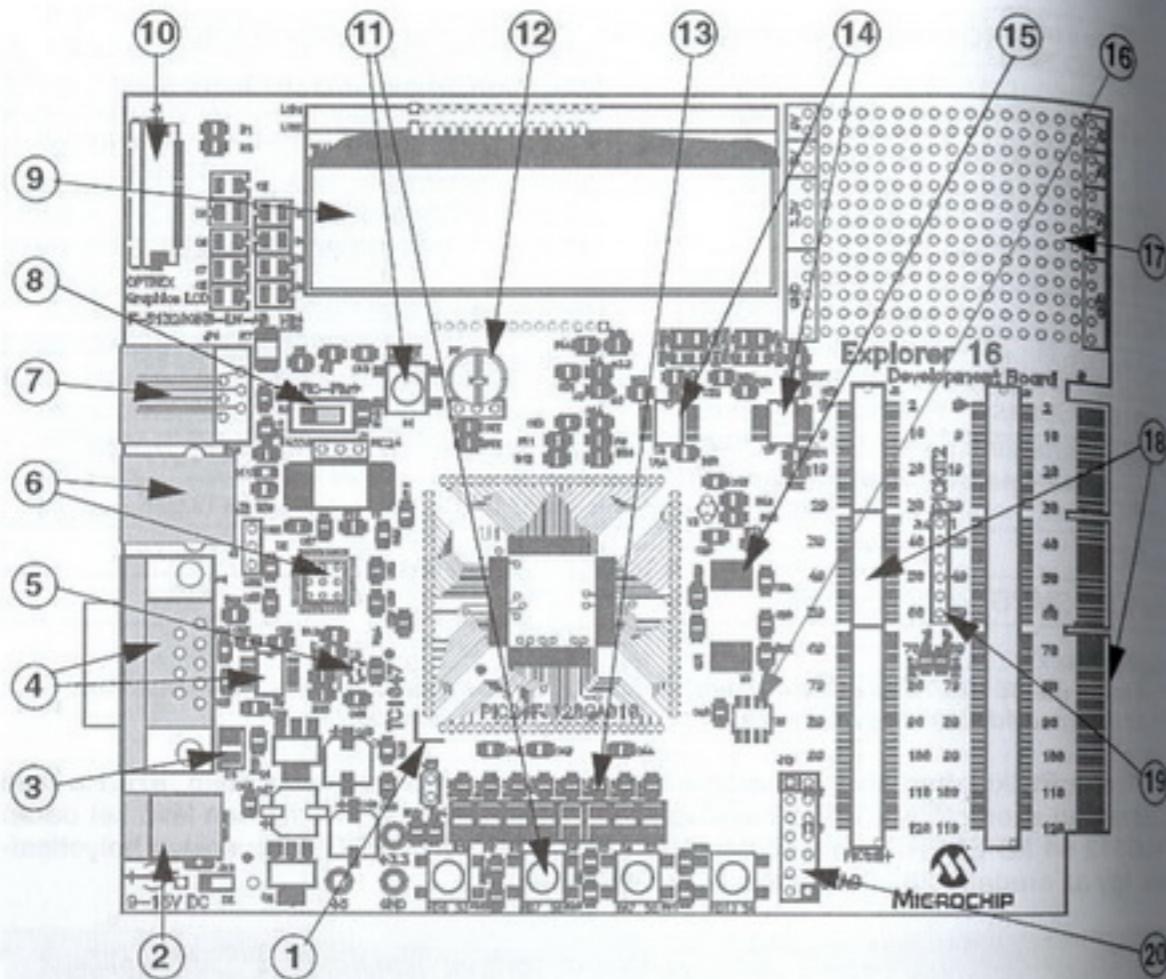
Mivel a PICkit2-ben lévő kód tárolásra szolgáló memória megduplázható, ezért a fenti táblázatban szereplő értékek is megduplázódnak. Ehhez a PICkit2-ben lévő két darab 24LC512 64 kB EEPROM-ot kell 24LC1025-I/SM 128 kB EEPROM típusokkal helyettesíteni. Így az eredeti 128 kB megnövekszik 256 kB-ra.

A PICkit 2 fejlesztőszkózzal kapcsolatos további ismeretek a DVD-melléklet /pickit2 könyvtárában megtalálhatók.

7.6. EXPLORER 16 DEMO BOARD

Az Explorer 16-os fejlesztőkártya jól átgondolt eszköz 16 és 32 bites PIC mikrovezérlők fejlesztésére. A kártya főbb részei a 7.7. ábrán mutatjuk be.

- 1) 100 lábú PIM processzormodul-foglalat, amibe bármelyik Microchip PIM modul behelyezhető (PIC24F/24H/dsPIC33F).
- 2) 9 VDC tápadapter-bemenet, előállítva a működtető +3.3 V és +5 V stabilizált feszültségeket.
- 3) Tápfeszültségjelző LED
- 4) RS-232 soros port és kapcsolódó áramkörei
- 5) Analóg hőmérséklet-érzékelő
- 6) USB csatlakozás kommunikációhoz és mikrovezérlők programozásához, debugolásához.
- 7) Szabványos 6 vezetékes In-Circuit Debugger (ICD) csatlakozó MPLAB ICD 2 modulhoz.
- 8) Tolókapcsoló PIM vagy beforrasztott PIC választására (későbbi fejlesztésekben – in future versions)
- 9) 2 soros, 16 karakteres LCD



7.7. ábra
Az Explorer 16 részei

- 10) OPTREX grafikus LCD csatlakozója
- 11) Nyomógombok Reset és felhasználó által kezelt bemenetekhez
- 12) Potenciometér, analóg bemenetként használható
- 13) Nyolcas LED-sor
- 14) 74HCT4053 multiplexerek a soros kommunikációs vonalak kiválasztható átkapcsolásához
- 15) Soros EEPROM
- 16) Kvarcok mikrovezérlő-órájel (8 MHz) és valós idejű órajel RTCC (32.768 kHz) előállításához
- 17) Felhasználható terület saját áramkörökhez
- 18) PICtail™ Plus csatlakozófoglalat és élcsatlakozó ilyen csatlakozással rendelkező kártyák számára
- 19) Hatlábú csatlakozó PICkit 2 programozó számára
- 20) JTAG csatlakozó, peremfigyeléshez

A PIC24/PIC30/PIC33 eszközök fejlesztéséhez a kártyán egy előre programozott PIC24FJ128GA010 Processor Installation Module (PIM) található, és ez lecserélhető a kártyához adott, előre programozott dsPIC33FJ256GP710 PIM modulra.

A könyvhöz mellékelt CD ROM-on mind a PIC24, mind a dsPIC33F családhoz számos demóprogram található, C nyelvű forrás és lefordított, letölthető .hex formátumban. A kártyán lévő PICtail Plus csatlakozás

A Microchip által kifejlesztett kiegészítő kártya illesztését teszi lehetővé, és a hozzájuk adott demóprogramokkal azonnal ki is próbálhatók:

- Audio PICtail Plus Daughter Board – AC164129
- ECAN/LIN PICtail Plus Daughter Board – AC164130
- USB PICtail Plus Daughter Board – AC164131
- PICtail board for SD and MMC – AC164122
- PICtail Plus board for Ethernet – AC164123
- Prototype PICtail Plus Daughter Board – AC164126
- IrDA PICtail Plus Daughter Board – AC164124
- Speech Playback PICtail Plus Daughter Board – AC164125
- PICDEM Z MRF24J40 2.4 GHz Daughter Card – AC163027-4
- Motor Control Interface PICtail Plus D-Card – AC164128
- Graphics PICtailTM Plus Daughter Board – AC164127

MICROCHIP

| | | | | | | |
|-------|-----|-------|-------|-----|-------|-----|
| J3 | RB2 | RF6 | J2 | RF2 | RF3 | |
| | RF7 | | | RC2 | RC3 | |
| | GND | | | QND | RB1 | |
| | RB3 | | | RB4 | CND | |
| | RE9 | | | RD0 | RE8 | RD1 |
| +3.3V | | | +3.3V | | +5V | |
| +9V | | D+ | +9V | | RF0 | +5V |
| | | | | | | |
| D- | | | | | RF1 | |
| RG9 | | RG6 | RF4 | | RF5 | |
| RG7 | | RG8 | RD7 | | RD8 | |
| GND | | RB8 | CND | | RB9 | |
| RB6 | | QND | RB7 | | CND | |
| RA15 | | RD2 | RA14 | | RD3 | |
| | | | | | | +5V |
| | | +3.3V | | | | |
| +9V | | P59 | +9V | | RG1 | |
| | | | | | | |
| P61 | | | | | RG0 | |
| RG15 | | RG12 | RG13 | | RG14 | |
| RA8 | | RA2 | RA1 | | RA3 | |
| RA4 | | RA6 | RA5 | | RA7 | |
| RA9 | | RB5 | RA10 | | MCLR | |
| R810 | | RB12 | RB11 | | RB13 | |
| R814 | | RC1 | RB15 | | RC2 | |
| RC3 | | RC13 | RC4 | | RC14 | |
| P93 | | P59 | P94 | | P96 | |
| RD4 | | RD6 | RD5 | | RD9 | |
| RD18 | | RD12 | RD11 | | RD13 | |
| RD14 | | -3.3V | RD15 | | +3.3V | |
| RE8 | | RE2 | RE1 | | RE3 | |
| RE4 | | RE6 | RE5 | | RE7 | |
| RF12 | | GND | RF13 | | GND | |

7.8. ábra
PICtail™ Plus csatlakozó kiosztás

Az Explorer 16 Demo Board fejlesztőeszközzel kapcsolatos további ismeretek a DVD-melléklet /expdemo16 könyvtárában megtalálhatók.

7.7. PIC PROGRAMOZÓK

A már bemutatott ICD2, PICkit1 és PICkit2 eszközök mellett kaphatók kimondottan PIC programozók is. Ezek is az MPLAB IDE környezetet használják. MPLAB PM3 esetén parancssoros, illetve az MPLAB-tól független kis grafikus programozói felület is rendelkezésre áll. PICSTART Plus DIP tokozású mikrovezérlök programozására alkalmas. Az emulátorok és a debugerek, mint az MPLAB REAL ICE, MPLAB ICD 2, és PICkit 2, szintén rendelkeznek programozási képességekkel.

- MPLAB PM3 (DV007004):** Ez a programozó képes a PC-hez illesztve, vagy önállóan működni és programozza az összes PIC mikrovezérlőt, még a legújabb dsPIC típusokat is. Ha önállóan használjuk, akkor a programozandó tartalmat SD/MMC kártyán kell elhelyezni. Az MPLAB IDE minden újabb verziója PM3 frissítést tartalmaz az újabb tokok programozásához.
- PICSTART Plus (DV003001):** Az eszköz olcsó programozó DIP tokos PIC-ek programozására. A PC-hez RS-232 soros vonalon kapcsolódik, és ingyenes kezelő szoftvere az MPLAB fejlesztői környezetbe van integrálva. Az MPLAB IDE minden újabb verziójában PICSTART Plus frissítést tartalmaz az újabb tokok programozásához.
- MPLAB Starter Kit soros memóriák számára (DV243003):** Mint a nevéből is következik, a Microchip által gyártott soros EEPROM memóriák programozását végezhetjük vele.

8. PIC PROGRAMOZÁS ASSEMBLER SEGÍTSÉGÉVEL

A programfejlesztés célja olyan gépkód- és adatsorozat létrehozása, amely alapján az adott mikroprocesszor, mikrokontroller a kívánt feladatot végrehajtja. (A szükséges aritmetikai, logikai műveleteket elvégzi, a perifériákat kezeli, stb.) Egy program általában a következő fő részekből áll:

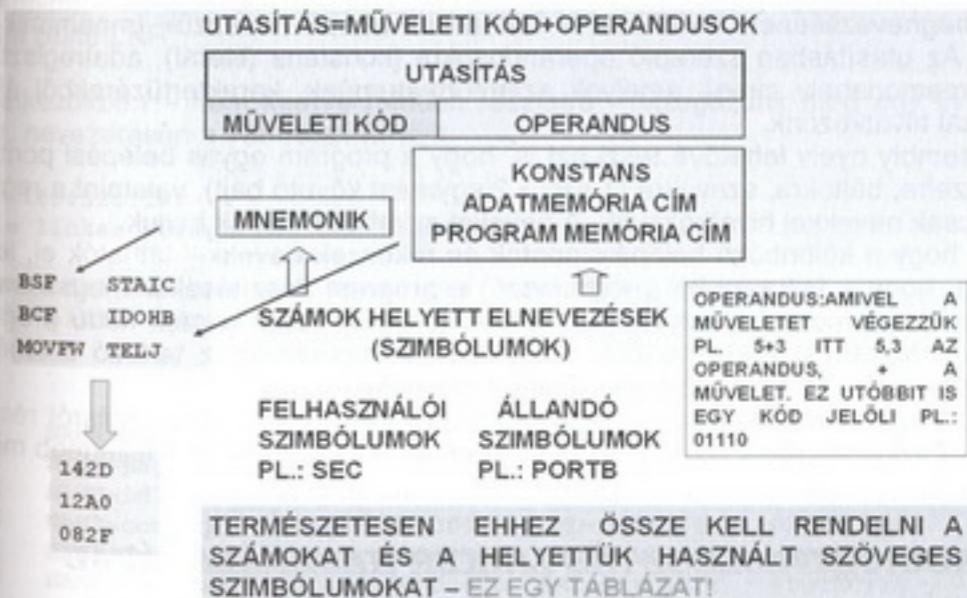
- Változók, állandók, definíciók, adatstruktúrák
- Főprogram
- Egyedi programblokkok (szubrutinok)

Mindezeket szövegesen és folyamatábrákon célszerű rögzíteni. Ez jelentősen segíti a programírást, illetve a menet közbeni ellenőrzéseket, majd a programélesztést.

8.1. UTASÍTÁSKÉSZLETEK – PROGRAMOZÓI MODELL

Az utasításkészlet felépítését a RISC-megoldás „ minden utasítás egyszavas” elve alapján meghatározza. Emlékezzünk, hogy minden utasítás egy, a műveletet meghatározó bitcsoportból, valamint a műveletben használt operandus címét vagy értékét meghatározó bitcsoportból áll. Az operandusok három csoportba sorolhatók:

- egy állandó, amivel a műveletet el kell végezni,
- egy adatmemória-cím, az ezen a címen található regiszter tartalmával végezzük el a műveletet,
- egy programmemória-cím, amivel a programban el tudunk ágazni.



8.1.ábra
Utasítás felépítése

Programozói modellnek nevezzük azt a leírást, ami a mikrovezérlő rendszerregisztereit, adat- és programmemoriáját a programozó szemszögéből mutatja be. Ezekkel az előző fejezetekben már megismertedtünk. Elsőként foglaljuk össze a PIC assembler programozással kapcsolatos ismereteket.

8.2. ASSEMBLER PROGRAMOZÁS

A mikroszámítógépek működtető programjainak elkészítésekor a programozó több lehetőség közül választhat. A leginkább „emberközeli” megoldás valamelyen, a feladathoz illeszkedő magas szintű nyelv választása (pl. BASIC, PASCAL, C). Egyes feladatoknál azonban ez nem járható út: a magas szintű nyelvek használatával nyerhető gépi kódú program általában nagyméretű és aránylag lassú. Ha a rendelkezésünkre álló memóriakapacitás kicsi, vagy a program végrehajtásának ideje kritikus, akkor a legcélsoberűbb megoldás a gépi kódú programozás mellett dönteni. Természetesen ilyenkor a program megírása, „belövése” több munkát igényel, cserébe a programozó a legszélesebb lehetőségekkel rendelkezik, ami a magas szintű nyelvre nem minden mondható el.

Minden számítógép, bármilyen bonyolult programot hajson is végre, végső soron a bináris számokat utasításként értelmezve végzi el a műveleteket a szintén bináris formájú adatokon. Ez teljesen nyilvánvaló, mivel a gépben csak bináris 1 és 0 alakú információ feldolgozása lehetséges. Az ilyen 0, 1 számok formájában előálló programot nevezük **gépi kódú programnak**.

A számítógép programozásában rejlö összes lehetőség legjobb kihasználását a gépi kódú programozás teszi leginkább lehetővé. Ezen a programozási szinten írhatók a gép működése szempontjából a leghatékonyabb programok, itt használhatók ki a legjobban az egyes utasítások (vagy utasításcsoportok) hatásai és mellékhatásai, itt alkalmazhatja a programozó a legszellemeesebb megoldásokat és trükköket.

A gépi kódon történő programozás azonban az ember számára rendkívül nehézkes, ezért helyette a szimbólumokat használó assembly programozást használják.

Az assembly tulajdonképpen a legegyszerűbb programozási nyelv. Lehetővé teszi, hogy azokat az utasításkódokat, amelyeket a gépi kódban számmal adtunk meg, könnyen megjegyezhető nevekkel írjuk le. Ezeket a neveket – melyek az utasítás által végrehajtott funkció megnevezésének rövidítései – **mnemonikoknak** nevezzük (mnemonik – emlékeztető). Az utasításban szereplő operandusokra [konstans (literál), adatregiszter címe, program-memória hely címe], amelyek számformátumúak, karakterfűzérekből álló szimbólumokkal hivatkozunk.

Az assembly nyelv lehetővé teszi azt is, hogy a program egyes belépési pontjaira, tárolórekeszeire, bájtokra, szavakra (1 szó = 2 egymást követő bájt), valamint a regisztekre ugyancsak nevekkel hivatkozzunk. A neveket szimbólumoknak hívjuk.

Azzal, hogy a különböző belépési pontok és rekeszek nevekkel láthatók el, lehetőség nyílik arra, hogy a felhasználó (programozó) a program készítésekor megszabaduljon a gépi kódú programozásnál szükséges címszámítástól. Ezzel a gépi kódú programozás említett két hátrányát kiküszöböltük. Nem kell az utasításokhoz tartozó számértékeket memorizálni, nem kell a programban címeket számolni.

Ez a bevezető összefoglaló elegendő, mert az assembly programozásról részletesen írtunk az előző kiadásban. Ezért a továbbiakban az ott nem szereplő fejlettebb módszereket tárgyaljuk.

8.3. PROGRAMFEJLESZTÉS LINKER HASZNÁLATÁVAL

A szokásos assembler fejlesztés során a programozó jelöli ki a programok és az adatok elhelyezkedését az adatmemoriában. Az assemblernek szóló direktívákkal (ORG, CBLOCK, EQU...) pontosan megadja azt, hogy mi hova kerüljön.

Abban az esetben, amikor többen írnak programot, ez módszer nehézkes, mert minden résztvevőnek meg kell adni egy általa kizártlagosan használt programterületet, általa helyileg (lokálisan) használt, illetve a mások által is használható (globális) adatmemória-terület címeit. Ezek alapján a végső programot összeállító személy fogja a forrásfájlokat egy fájlba egyesíteni, a programot lefordítani, ami a nehézkessége mellett sok hiba forrása is lehet.

Természetesen ez a módszer egy erre a célra íródott szerkesztőprogram segítségével automatizálható. Elsőként azt kell észrevennünk, hogy a programmemoriában a megszakításvektor csak adott címről kerülhet, és ez igaz a programunk kezdetére is. A programunk egyéb részeire ilyen megkötések nincsenek, azok bárhol elhelyezkedhetnek.

Hasonló megmondás igaz az adatmemoriára is, azzal a kiegészítéssel, hogy a kitüntetett speciális funkciójú regiszterek kivételével az adatmemória többi regiszterét szabádon felhasználhatjuk.

A Microchip a hagyományos és a linkerrel történő programfejlesztéshez alapértelmezés szerint a „C:\ProgramFiles\Microchip\MPASM Suite\Template” alatti két könyvtárban előre elkészített működő keretprogramokat ad minden processzortípushoz. Ezeket lehet felhasználni a megírandó programunk alapjának.

- A **Code** könyvtárban vannak a hagyományos, abszolút-kód-fejlesztés fájljai. Ez azt jelenti, hogy minden pontosan úgy valósul meg, ahogy a programozó megadja: a program az **ORG** direktívával megadott helyen kezdődik, a változók a **CBLOCK** vagy az **EQU** direktívával megadott helyre kerülnek. A direktívákat az assembler kezeli, folyamatosan értelmezi, és végrehajtja a direktívában leírtakat. A könyvtárban szereplő fájlokban az **ORG** direktívával adjuk meg a program kezdetét kijelölt **RESET**, illetve a megszakítási kezdőpontok helyét.
- Az **Object** könyvtárban az áthelyezhető kódot használó fejlesztéshez tartozó fájlok vannak. Ez azt jelenti, hogy a szerkesztő (*linker*) fogja elhelyezni a programkódot a memoriában, aminek kezdetét a **CODE** direktíva jelöli. Az assembler állítja elő az utasítások alapján a gépi kódot minden különálló, **CODE** direktívával jelölt kód részletre, amelyek természetesen önálló fájlként, egy-egy önálló feladatot megvalósító, újrafelhasználható kódként is megjelenhetnek. Azt, hogy az egyes gépi kód részletek ténylegesen hova kerülnek, a program és adatmemória fizikai méretét figyelembe vevő **LKR** fájlban megadottak határozzák meg. Ezek helye: „C:\ProgramFiles\Microchip\MPASM Suite\LKR”

A következőkben – nem kitérve minden részletre – vizsgálunk meg egy ilyen linker script fájlt, nevezetesen a **18F4520.lkr** fájlt.

```
// File: 18f4520.lkr
// Sample linker script for the PIC18F4520 processor

// Not intended for use with MPLAB C18. For C18 projects,
// use the linker scripts provided with that product.

LIBPATH .

CODEPAGE NAME=_vectors START=0x0 END=0x29 PROTECTED
CODEPAGE NAME=page START=0x2A END=0x7FFF
CODEPAGE NAME=idlocs START=0x200000 END=0x200007 PROTECTED
CODEPAGE NAME=config START=0x300000 END=0x30000D PROTECTED
CODEPAGE NAME=devid START=0x3FFFFE END=0x3FFFFFF PROTECTED
CODEPAGE NAME=eedata START=0xF00000 END=0xF000FF PROTECTED

ACCESSBANK NAME=accessram START=0x0 END=0x7F
DATABANK NAME=gpr0 START=0x80 END=0xFF
DATABANK NAME=gpr1 START=0x100 END=0x1FF
DATABANK NAME=gpr2 START=0x200 END=0x2FF
DATABANK NAME=gpr3 START=0x300 END=0x3FF
DATABANK NAME=gpr4 START=0x400 END=0x4FF
DATABANK NAME=gpr5 START=0x500 END=0x5FF
ACCESSBANK NAME=accesssfr START=0xF80 END=0xFFFF PROTECTED
```

A // jelöli a megjegyzések kezdetét, ahol az is kiderül, hogy ez a fájl C nyelvű fejlesztés esetén nem használható.

LIBPATH: Az aktuális könyvtárban van ez a fájl.

A NAME direktíva után álló név tetszőleges lehet.

START, END: ez jelöli ki a terület kezdetét és végét.

PROTECTED: ez a direktíva azt jelzi, hogy a megfelelő program vagy adatrész (szegmens) mint erőforrás speciális tulajdonságú, általánosan nem használható.

- **Programmemória megadása:**

- **CODEPAGE** direktívával jelölhetjük meg a kódmemória szegmenseit.
- **vectors:** Amint látható, van egy máshova nem helyezhető, vectors nevű szegmens, amit RESET, illetve a két megszakítási cím helyét tartalmazó területet foglalja magában.
- **page:** szegmensbe kerülnek a programunk moduljai.
- **idlocs:** a tokba beírható hét karakteres azonosító.
- **config:** a konfigurációs memória területe.
- **devid:** a tok azonosítója.
- **eedata:** a tokban lévő adat EEPROM memória szegmense.

- **Adatmemória megadása:**

- **DATABANK:** Az adatmemória a bankhasználat miatt 256 bájtos RAM bankokra van felosztva, és ezeket jelöljük meg egyenként ezzel a direktívával.
- **ACCESSBANK:** A 18-as családban lévő ACCESSRAM speciális tulajdonságú, ezért ezt külön direktíva jelöli.

Az alább következő táblázatban összehasonlítjuk az abszolút és az áthelyezhető módon felépített programot.

A linkerrel dolgozó programnál a változók megadásánál a RES (reserve –foglalás) direktívát használjuk. Ezzel csak jelezük a helyigényt, és majd a linker program ad a változóknak konkrét adatmemória-címet.

Az adatmemória-területek megadására a következő direktívák szolgálnak:

- **udata** – az így kijelölt területen lévő regisztereknek nincs kezdőértéke.
- **udata_acs** – az így kijelölt tartomány a PIC18 Access Ram területén helyezkedik el.
- **udata_ovr** – az így kijelölt tartományban lévő címeket más változónevekkel többször használhatjuk, felülírva az előző értékeit.
- **udata_shr** – az így kijelölt tartomány a bankok közös területén (0x70-0x7F) helyezkedik el.
- **idata** – az így kijelölt területen lévő regisztereknek kezdőértéket adhatunk, vele kapcsolatosan szintén használhatók az _acs, _ovr, _shr kiterjesztések.

| HAGYOMÁNYOS ASSEMBLER PROGRAMOZÁS | PROGRAMOZÁS LINKER HASZNÁLATÁVAL |
|--|---|
| <ul style="list-style-type: none"> UJ.mcp <ul style="list-style-type: none"> Source Files <ul style="list-style-type: none"> 18F4520TEMP.ASM Header Files Object Files Library Files Linker Script Other Files | <ul style="list-style-type: none"> UJ.mcp <ul style="list-style-type: none"> Source Files <ul style="list-style-type: none"> 18F4520TMPO.ASM Header Files Object Files Library Files Linker Script <ul style="list-style-type: none"> 18f4520.lkr Other Files |

```

LIST P=18F4520 ;PROC.TIPUS MEGADÁSA
#include <P18F4520.INC>

;VÁLTOZÓK MENTÉSHEZ
ORG 0x080
    WREG_TEMP      ;VÁLTOZÓK MENTÉSHEZ
    STATUS_TEMP    ;MENTÉSHEZ
    BSR_TEMP       ;MENTÉSHEZ
ENDC

;FELH. VÁLTOZÓ DEF.
ORG 0x000
    EXAMPLE        ;FELH. VÁLTOZÓ DEF.
ENDC

;UGRÁS FÓPROGRAMRA
ORG 0x0000
    goto Main      ;UGRÁS FÓPROGRAMRA

;MAGAS IT
ORG 0x0008
    bra HighInt   ;UGRÁS ISR-RE

;ALACSONY IT
ORG 0x0018
    movff STATUS, STATUS_TEMP ;MENTÉS
    movff WREG, WREG_TEMP
    movff BSR, BSR_TEMP
;ALACSONY SZINTŰ IT-T KISZ. PR.RÉSZ
    movff BSR_TEMP, BSR
;VISSZAÁLLÍTÁS
    movff WREG_TEMP, WREG
    movff STATUS_TEMP, STATUS
    retfie

;HIGHINT:
;MAGAS SZINTŰ IT-T KISZOLGÁLÓ PR.RÉSZ
    retfie      FAST

;MAIN:
;    *** ITT VAN A FÓPROGRAM ***
END

;*** ITT VAN A FÓPROGRAM ***

```

```

LIST P=18F4520 ;PROC.TIPUS MEGADÁSA
#include <P18F4520.INC>

UDATA
    WREG_TEMP      RES 1 ;VÁLTOZÓK
    STATUS_TEMP    RES 1 ;MENTÉSHEZ
    BSR_TEMP       RES 1

UDATA_ACS
    EXAMPLE        RES 1 ;FELH.VÁLT. DEF.

RESET_VECTOR
    CODE 0x0000
    goto Main      ;UGRÁS FÓPR.-RA

HI_INT_VECTOR
    CODE 0x0008
    bra HighInt   ;UGRÁS MAGAS IT.

LOW_INT_VECTOR
    CODE 0x0018
    bra LowInt     ;UGR. LOW IT.-RE

CODE

HighInt:
;MAGAS SZINTŰ IT-T KISZOLGÁLÓ PR.RÉSZ
    retfie      FAST

LowInt:
    movff STATUS, STATUS_TEMP ;MENTÉSEK
    movff WREG, WREG_TEMP
    movff BSR, BSR_TEMP
;ALACSONY SZINTŰ IT-T KISZ. PR.RÉSZ
    movff BSR_TEMP, BSR
;VISSZAÁLLÍTÁSOK
    movff WREG_TEMP, WREG
    movff STATUS_TEMP, STATUS
    retfie

Main:
;    *** ITT VAN A FÓPROGRAM ***
END

;*** ITT VAN A FÓPROGRAM ***

```

Összefoglalva: a program- és adatmemoriát felosztjuk a mikrokontroller típusa által meghatározott fizikai szegmensekre, a programozó a saját programját és adatváltozót pedig logikai szekciókra bontja.

A linker feladata a logikai szekciókat a fizikai szegmensekbe elhelyezni. A szimbólumokkal jelölt összerendeléseket egy .lkr kiterjesztésű linker szkript fájl tartalmazza.

8.3.1. Map fájl használata

A linker használatakor a program automatikusan generál egy ún. **Map fájlt *.map** kiterjesztéssel, melyet a projektet tartalmazó mappában érhetünk el. Ha nincs szükségünk fájlra, alkalmazzuk a Project → Build Options → Project parancsot. Az MPLINK Linker fólión letölthetjük az automatikus generálást, ehhez a **Generate map file** jelölönégyzetből vegyük ki a pipát.

A fájlt felhasználhatjuk például a programunk dokumentálására, így az esetleges módszertáskor könnyedén szerezhetünk információt a már lefoglalt memóriaterületekről,

használt szimbólumokról. Az MPLAB IDE, ha módosítjuk a programunkat, felülírással automatikusan újra létrehozza a *map* fájlt.

A *map* fájl tartalmát három részre oszthatjuk:

Az **első részben**, a felosztott szekciókról láthatunk információt, táblázatszerűen elrendezve. Az információ tartalmazza a szekció nevét, a típusát, a címét, az elhelyezkedését és a méretét bajtokban.

A **második rész** egy táblázat, a lefoglalt programmemória-területeket sorolja fel, megadva a kezdő- és végcímeket. Ezenkívül számszerű információt kapunk a felhasznált programmemóriáról.

| Program Memory Usage | |
|---|----------|
| Start | End |
| 0x000001a | 0x000133 |
| 0x000500 | 0x00050f |
| 298 out of 32768 program addresses used, program memory utilisation is 0% | |

8.2. ábra
Map fájl – Programmemória

A *map* fájl **harmadik része** a programban használt szimbólumokat jeleníti meg, szintén táblázatba foglalva. Láthatjuk a szimbólum nevét, címét, a memóriában történő elhelyezkedését (program- vagy adatmemória), kapcsolódását (extern vagy static) és annak a fájlnak az elérési útvonalát, amelyben a szimbólum szerepel.

| Symbols - Sorted by Name | | | | |
|--------------------------|----------|----------|---------|---|
| Name | Address | Location | Storage | File |
| BCD2BIN16 | 0x0000b6 | program | extern | C:\Project\Soros_modul\BCD_to_HEX.asm |
| BCD2BIN16LOOP | 0x0000ba | program | static | C:\Project\Soros_modul\BCD_to_HEX.asm |
| BCD_TO_HEX | 0x0000e6 | program | extern | C:\Project\Soros_modul\elokeszit.asm |
| BIN_ASCII | 0x000076 | program | extern | C:\Project\Soros_modul\hex_to_ascii.asm |
| GETBYTE | 0x000032 | program | extern | C:\Project\Soros_modul\foprogram.asm |
| HEX_ASCII | 0x0000a4 | program | static | C:\Project\Soros_modul\hex_to_ascii.asm |
| HEX_TABLA | 0x000500 | program | static | C:\Project\Soros_modul\hex_to_ascii.asm |
| HUROK | 0x00011e | program | static | C:\Project\Soros_modul\elokeszit.asm |

8.3. ábra
Map fájl – Szimbólumok

8.4. TÖBBMODULOS PROGRAMOZÁS

Mikrovezérlökkel megoldandó feladatok esetén nagyon sokszor használjuk fel a már megrajzolt programjaink egy-egy megoldását, mivel a már elkészült és tesztelt részletet nem kell ismételten kitalálni. Ez természetesen nagyon sokszor csupán úgy történik, hogy egy szövegszerkesztő segítségével a megoldást az eredeti helyéről kivágjuk, és az aktuális helyre beillesztjük. A linker használata lehetővé teszi, hogy a programjainkban az önálló tevékenységet, feladatot megvalósító, bizonyítottan helyesen működő modulokat használunk.

Ilyen modul lényegében egy program, amiben utasítások, változók és címek szerepelnek – hasonlóan más programokhoz. A modulok együttműködése igényli egymás változóinak, eljárásainak, címeinek, más néven objektumoknak az ismeretét és esetleges felhasználhatóságát. Vagyis egy modulban három típusú objektum lehet:

8. fejezet: PIC programozás assembler segítségével

- Lokális** – csak a modulon belül használt objektum,
 - Globális** – azok az objektumok, amelyeket mások felhasználhatnak, ennek jelölése: **GLOBAL**,
 - Külső** – a modulban használt, de más modulban GLOBAL-ként megadott objektum, ennek a jelölése: **EXTERN**.
- Amint látható, a lokális objektumokat nem kell megjelölni, mert ennek hiánya jelzi a lokalitást. Az együttműködés és a helyes linkelés feltétele, hogy az egy modulban EXTERN-ként definiált objektumok valamelyik másik modulban GLOBAL-ként legyenek megadva. Néhány szabályt be kell tartanunk ahhoz, hogy modulárisan tudjunk programozni:
- Minden modulban szerepelnie kell a használt processzorhoz tartozó, *.inc kiterjesztésű fájlnak, amelyet **#include** utasítással fűzünk a programhoz (ez nem csak a moduláris programozásra igaz).
 - Minden modulban el kell helyezni az **END** utasítást; nem kötelező, hogy a program végén legyen, de a fordító az **end** utasítás után írt programrészeket már nem veszi figyelembe (ami nem csak a moduláris programozásra igaz). **Vagyis ez után akár a program vagy modul leírását is elhelyezhetjük.**

| | |
|--|--|
| <pre> LIST P=16C54 #include "P16C5X.INC" EXTERN MULCND, MULPLR EXTERN H_BYTE, L_BYTE, MPY CODE START CLRW OPTION MAIN MOVP PORTB, W MOVWF MULPLR MOVF PORTB, W MOVWF MULCND CALL MPY ; THE RESULT IS IN ; H_BYTE & L_BYTE RESET GOTO MAIN CODE H'03FF' GOTO START END LOOP BTFSC STATUS, C ; CLEAR CARRY BIT RRF MULPLR, F ADDWF H_BYTE, F RRF H_BYTE, F RRF L_BYTE, F DECFSZ COUNT, F GOTO LOOP RETLW 0 END </pre> | <pre> LIST P=16C54 #include "P16C5X.INC" UDATA MULCND RES L ; 8 BIT MULTIPLICAND MULPLR RES I ; 8 BIT MULTIPLIER H_BYTE RES 1 ; HIGH BYTE OF RESULT L_BYTE RES 1 ; LOW BYTE OF RESULT COUNT RES 1 ; LOOP COUNTER GLOBAL MULCND, MULPLR, H_BYTE, L_BYTE, MPY CODE GLOBAL MPY CLRF H_BYTE CLRF L_BYTE MOVLM 8 MOVWF COUNT MOVP MULAND, W BCF STATUS, C ; CLEAR CARRY BIT RRF MULPLR, F BTFSC STATUS, C ADDWF H_BYTE, F RRF H_BYTE, F RRF L_BYTE, F DECFSZ COUNT, F GOTO LOOP RETLW 0 END </pre> |
|--|--|

- az egyes modulokban szerepeljen a **GLOBAL** és az **EXTERN** direktívákat,
 - az egyes modulok összefűzését az **MPLINK** végzi, használatához egy *.lkr kiterjesztésű fájlt kell létrehozni (a kódszegmensek definíálása itt történik), vagy használhatunk egy sablonfájlt is, amelyet átszerkesztünk,
 - az **MPLINK** használatakor a **RES** direktívát kell használni a változó területek létrehozására a RAM-ban,
 - minden modulban hivatkoznunk kell a *.lkr fájlból definált kódszegmensekre (pl. program_set code).
- a főprogramban a változók memóriában történő elhelyezését is meg kell adnunk. A hagyományos programozástól eltérően ez esetben egymás alá kell írnunk a változókat (nem írhatjuk egymás mellé vesszővel elválasztva) és megadnunk, hogy a memoriában mennyi helyet foglalunk le az egyes változóknak.

Az **EXTERN** és **GLOBAL** direktívák után kell felsorolni az objektumokat, amelyek ténylegesen szimbolikus változónevek, illetve adott programmemória-helyre, szubrutinra mutató címkék.

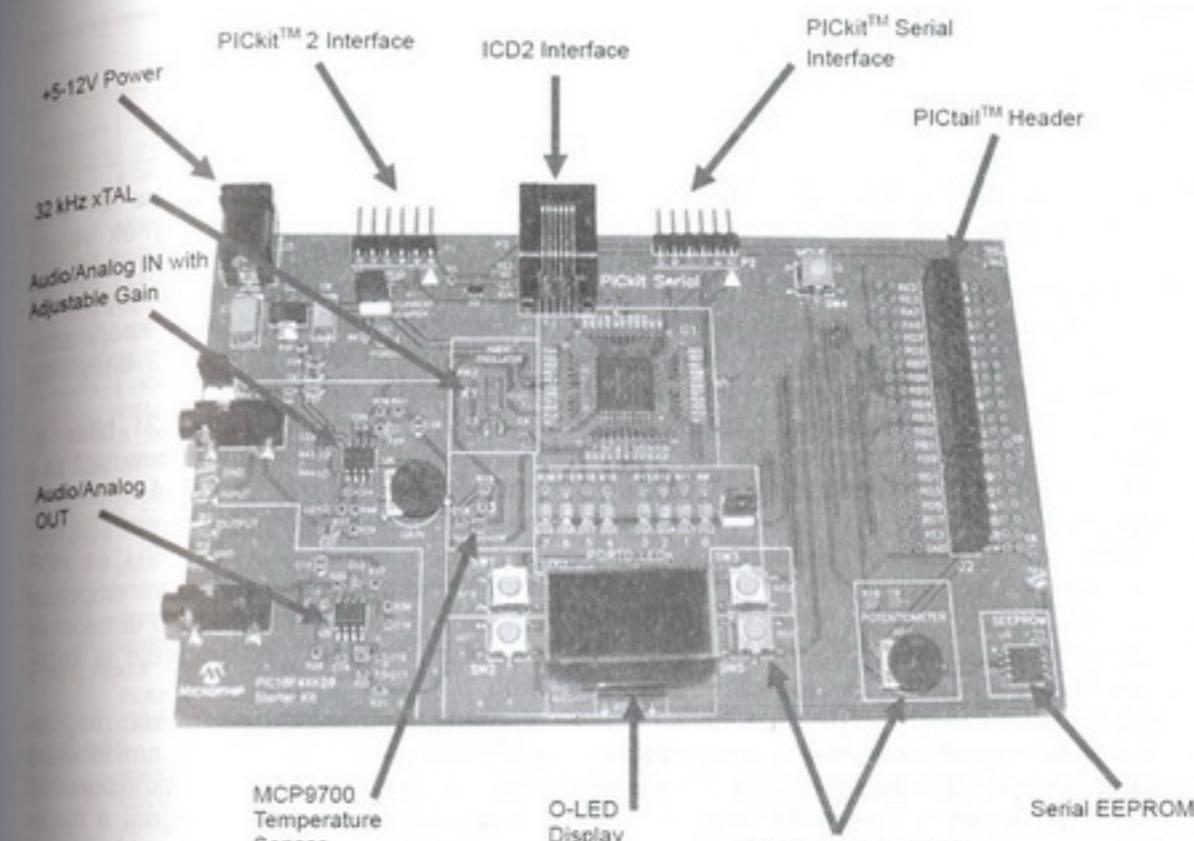
Sajnos egyes bitek nem lehetnek objektumok. Ilyenkor egy közös (legyen a neve **COMMON.INC**) include fájlban kell szerepelni a regiszter bitjeire vonatkozó hozzárendeléseket. A modulokban az adott regiszterre vonatkozó **EXTERN**, illetve **GLOBAL** direktívákat használjuk, és a modulok elején végezzük a **COMMON.INC** fájl beillesztését az **INCLUDE** direktívával.

8.5. ASSEMBLER MINTAPÉLDÁK

Tizenhat assembler mintapéldán keresztül mutatjuk be a legfontosabb programozási ismertetéket. Elsőként röviden összefoglaljuk az egyes mintapéldákban megjelenő tématörököt.

- 1) **mina_00.asm** – Assembler programminta. A programozáshoz célszerű egy megszokott, jól kialakított sémát használni. A következő feladatok felépítése ezt a sémát követi.
- 2) **blink_01.asm** – LED-villogtatás. Szoftveridőzítés. Billentyűállapot LED-re. Ez a legelső, sikeresményt adó programozási feladat.
- 3) **bcnt_02.asm** – LED-soron egy nyomógomb-megnyomást számláló regiszter bináris megjelenítése. Hosszú-rövid gombnyomás detektálása.
- 4) **rotate_03.asm** – Futófényprogram készítése. Időzítés szubrutinban. Gombnyomásra jobbra-balra léptetés.
- 5) **adled_04.asm** – A/D mérés eredménye LED-sorra. Az A/D átalakító használata.
- 6) **blinkpol_05.asm** – LED-villogtatás tmr0 számláló túlcordulásának figyelésével. Az időzítést már nem egy programozott késleltetéssel, hanem a beépített számítóperiferiával valósítjuk meg.
- 7) **ittabla_06.asm** – Futófény megszakítással. Alakzat megjelenítése táblakezeléssel. Megismerkedünk a megszakítás használatával és a táblakezeléssel.
- 8) **adtabla_07.asm** – Ezt a feladatot az előzőek felhasználásával oldjuk meg. A/D mérés kivezérlésmérőszerű, táblakezeléses megjelenítéssel.
- 9) **indir_08.asm** – Indirekt címzés – adatgyűjtés gombnyomásra, LED-soron darabszám. Mért adatok átlagának számítása, LED-soron kijelzése.
- 10) **stmach_09.asm** – A programozási gyakorlatban nagy szerepe van a feladatok állapotgéppel történő megoldásának. A feladat egy ilyen állapotgép programozása: állapotok: 0 = LED nem világít, 1 = folyamatosan világít, 2 = egyik LED villog, 3 = másik LED villog, más frekvenciával. 4 = ugrás 0 állapotba. A váltások nyomógombkezeléssel valósulnak meg. Makrók használata.
- 11) **preempt_10.asm** – Szintén fontos programozási gyakorlat az elvégzendő feladatok „párhuzamos” kezelése. Preemptív multitasking. A feladatok (taszkok) különböző időpontbeli LED-villogtatások. Feltételes assemblálás bemutatása.
- 12) **bilkez_11.asm** – Billentyűkezelés. Billentyűnyomások és -elengedések kezelése.
- 13) **eeprom_12.asm** – EEPROM-írás, olvasás. A/D mérés gombra. Eredményt LED-soron megjeleníteni, majd EEPROM-ban tárolni. Ha az eredmény nulla, akkor kilép LED-et villogtat 2 mp-ig. Majd gombnyomásokra kiolvassa a LED-sorra az EEPROM-ban tárolt mérési eredményt.
- 14) **frekgen_13.asm** – Különböző frekvenciájú négyzetgyűjtelek generálása.
- 15) **kodzar_14.asm** – Egy nyomógombos kódzár.
- 16) **adkod_15.asm** – Analóg feszültséget vizsgáló kódzár.
- 17) **jatek_16.asm** – Játék.

A mintapéldákat az előző fejezetben bemutatott **PICF4XK20** Kit-en fejlesztettük, ami kimondottan a 18-as család megismerésére szolgál. A panelen egy **PIC18F46K20** típusú mikrovezérlő található. Emlékeztetőül a panel kialakítása és a rajta lévő perifériák megnevezése a 8.4. ábrán, illetve az utána következő felsorolásban látható.



8.4. ábra
PICF4XK20 gyakorlópanel

- 128x64 Organic LED Display (SPI)
- 32 768 kHz külső oszcillátor (Timer1)
- Analóg bemeneti szűrő és erősítő (RE1)
- PWM kimeneti szűrő (RC2)
- 4 nyomógomb (PORTB0-PORTB3)
- 1 MCLR gomb (RESET)
- 8 LED (PORTD)
- Potenciométer (RE0)
- 1024 KBit soros EEPROM
- PICtail™ csatlakozó
- 6 lábú ICSP™ programozó tüskesor
- 6 lábú PICkit soros analizátor interfész
- Árammérő-átkötés
- RJ-11 ICSP programozóaljzat

A feladatok ismertetése során a teljes programok helyett csak a fontos kód részletek szerepelnek, magyarázatokkal kiegészítve. A programok projektekből szervezett alakja a DVD-melléklet **ASM16PRG** könyvtárában találhatók, a pld00...pld16 alkonyvtárakban. A mintapéldák PIC16-os családra írott változatai az **ASM16PRG** könyvtár pld00...pld16 alkonyvtáraiba kerültek, és a PICKIT DEBUG EXPRESS fejlesztőhöz adott PIC18F887-es mikrovezérlőt tartalmazó demókártyán történt a tesztelésük.

A következő táblázatban röviden összefoglaltuk azokat a különbségeket, amik a 8/14 bites és a 8/16 bites mikrovezérlők között vannak.

Fontosabb különbségek

| PIC16 | PIC18 |
|--|---|
| <ul style="list-style-type: none"> Az utasítások 14 bit szélesek, ennek következménye az adat- és programmemória címzésében mutatkozik: az utasításban a korlátozott hossz miatt csak logikai címzést (az utasításban szereplő cím) használhatunk. A fizikai adat- és programmemóriát csak a logikai címek bittel történő kiegészítéssel kezelhetjük. Ezt a programmemória esetén lapozásnak, az adatmemória esetén bankváltásnak hívjuk. Fizikai memóriacím = utasításban szereplő cím + kiegészítő bitek. Következmény: lapozni kell a programmemóriát – 2 kiszavas programlapok vannak, és szükséges a regiszterbankok váltása is. A programmemóriában elhelyezett adatokat a RETLW utasítással kell tárolni, és a PCLATH regiszter határozza meg a helyét. Verem mélysége: 8 Egyszintű megszakításkezelés (nincs prioritás). | <ul style="list-style-type: none"> A programmemória címzésére 21 bites PC regiszter szolgál, amelynek részeit PCU<20:16>PCH<15:8>PCL<7:0>. nem kell a programmemóriát lapozni! A maximum 4096 bájtos (4 kb által) adatmemória kezelését három 12 bites FSR regiszterrel is végezhetjük. A hardververem 31 darab 21 bites regiszterből áll, az ötbites veremmutató és a verem alul-, illetve túlcordulását jelző bitek egy írható/olvasható verem-státusregiszterben találhatók. A PUSH és POP utasításokkal, amelyekkel a verempointer állítható, a verem tetején lévő tartalom és az utasításmutató (PC) cseréje is kezelhető. A megszakításkezelésben két megszakítási cím van, a 08h és 18h, ami lehetővé teszi a prioritásos megszakításkezelést olyan módon, hogy a 08h című a magasabb, a 18h című az alacsonyabb prioritású megszakítás. A hatékonyúság miatt szakítottak a „ minden utasítás egyszavas” elvvel. A call, goto utasítások a teljes memóriát elérő [mivel ezek két szóból (= 32 bit)] állnak. Ilyen kétszavas utasítás még két 12 bites című fájlregiszter tartalmának mozgatása (MOVFF), illetve az FSRx-et 12 bites címliterállal (=címkonstans) feltöltő utasítás (LFSR). |

8.5.1. minta00.asm

Már az előzőekben írtunk róla, hogy célszerű egy olyan programot készíteni, ami formalag az összes program alapjának tekinthető. Ez a program a következő:

```
;-----[ MINTAPROGRAM ]-----
;#1                      MINDEN FAJLHOZ-NYULASKOR 1-GYEL NOVELNI!!!
;VER: 1.0                  DATE: 2007.12.09 - UTOLSO MODOSITAS
;FAJLNEV:                 MINTA.ASM NOTE: PICKEZDO TANF.
;IRTA:                    DR. KONYA LASZLO EMAIL: KONYA.LASZLOGKV.K.BMF.HU
;CEG:                     BMF KANDO
;-----[ MI EZ? ]-----
;MINTAPROGRAM - EGY "KITÖLTENDŐ ÜRLAP"
;MINDEN NAGYBETŰS.
;MINDEN SZÍMBÓLUM A TÍPUSÁRA UTALÓ BETÜVEL KEZDŐDIK, ÉS EGY ALÁHÚZÁS
;UTÁN VANNAK AZ ELNEVEZÉSEK.
;JELÖLÉSEK:
;L-CIMKE S-SZUBRUTIN M-MAKRO R-REGISZTER(VÁLTOZÓ) C-KONSTANS(LITERAL)
;T-TÁBLÁZAT B-BIT BI-BEMENETI BIT BO-KIMENETI BIT BA-ANALÓG BEMENET
```

```
;-----[ TORTENET IDE KERUL MINDEN INFORMACIO ]-----
;-----[ SPECIFIKACIOK, KIEGESZITESEK ]-----
;  
[ D E F I N I C I O K ] ****
;  
[ PROCESSZOR + KONFIGURACIO + INCLUDE ] ****
;INCLUDE "..\PFEJ.INC" ;PROCESSZOR TÍPUS DEFINICIO VAN BENNE
;  
[ ALLANDOK ] -----
;DEFINE GOMB PORTB,0
;DEFINE DSPORT LATD
;DEFINE IRPORT DDRD
;  
C_ADCON0 EQU 1      ;ADON, 0. CSATORNA
C_ADCON1 EQU 0X0E   ;ANO ANALOG TÖBBI DIG. INPUT , VDD-VSS REF.
C_ADCON2 EQU 0X3F   ;BALRA IG RC ORAJEL
;  
[ VALTOZOK ] -----
CBLOCK 0X000          ;EXAMPLE OF A VARIABLE IN ACCESS RAM
R_EXAMPLE
ENDC
;  
[ M A K R O K ] -----
;MOSZ MEG ITT NINCS ILYEN
;  
*****[ PROGRAMKÓD KEZDETÉ PROGRAM VÉGREHAJTÁSA INNEN KEZDŐDIK ]*****
ORG 0X0000
;  
L_INI_CLK
    MOVLW B'01110010' ;MŰKÖDTETŐ ORAJEL BEÁLLÍTÁS 8 MHZ BELSÖ
    MOVWF OSCCON ;OSCCON: IDLEN IRFC2 IRFC1 IRFC0 OSTS IOFS SCS1 SCS0
;  
L_INI_PERIP
    CLRF DSPORT ;KIMENET
    CLRF IRPORT ;
;  
L_INI_ADO
    MOVLW C_ADCON1
    MOVWF ADCON1
    MOVLW C_ADCON2
    MOVWF ADCON2
    MOVLW C_ADCON0
    MOVWF ADCON0
;  
*****[ FÓPROGRAM KEZDETÉ ]-----
L_MAIN:
;  
    *** MAIN CODE GOES HERE ***
;  
*****[ S Z U B R U T I N O K ]-----
; IDE KERÜLNÉK A SZUBRUTINOK
;
*****[ PROGRAMKÓD VÉGE ]*****
;  
PONTOS! AZ END UTÁNI RÉSZT AZ ASSEMBLER NEM VESZI FIGYELEMBE, EZÉRT IDE BÁRMILYEN, A
PROGRAMMAL KAPCSOLATOS, VAGY EGYÉB INFORMÁCIÓT ELHELYEZNETÜNK.
```

Ez a programfelépítés természetesen egyéni, szerkezete röviden: információs rész – definiciós rész – program – szubrutinok. Ez a felépítés azért előnyös, mert olvasása kezdetén megismerhetjük a program feladatát, majd definiáljuk a program változóit, állandóit, kisebb szerkezeti elemeit. Utána található maga a program, ami számos, sokszor folyamatosan bővülő szubrutint tartalmaz. Ezért célszerű ezt a részt a program végén hagyni.

Mivel minden program esetén ugyanaz a hardverkörnyezet, a példaprogramokat tartalmazó alkönyvtárak szintjén elhelyeztük a processzordefiníciót tartalmazó PFEJ.INC include fájlt, amire minden program elején hivatkozunk. Ennek tartalma:

```
LIST P=18F46K20
#include <P18F46K20.INC>
#include <..\CONF18F46K20.INC>
;  
DIRECTIVE TO DEFINE PROCESSOR
;PROCESSZORRA VONATKOZÓ DEFINICIÓK
;A KONFIGURÁCIÓ BEÁLLÍTÁSA
```

A conf18F46K20.INC állományt a P18F46K20.INC állományban található mintakonfiguráció átszerkesztésével kaptuk meg. Ott eredetileg minden konfigurációs bejegyzés portosvesszővel volt kezdve. A megfelelő sorok elől ezt kivéve és a CONFIG direktívával kiegészítve kaptuk meg a konfigurációs fájlt.

8.5.2. blink01.asm

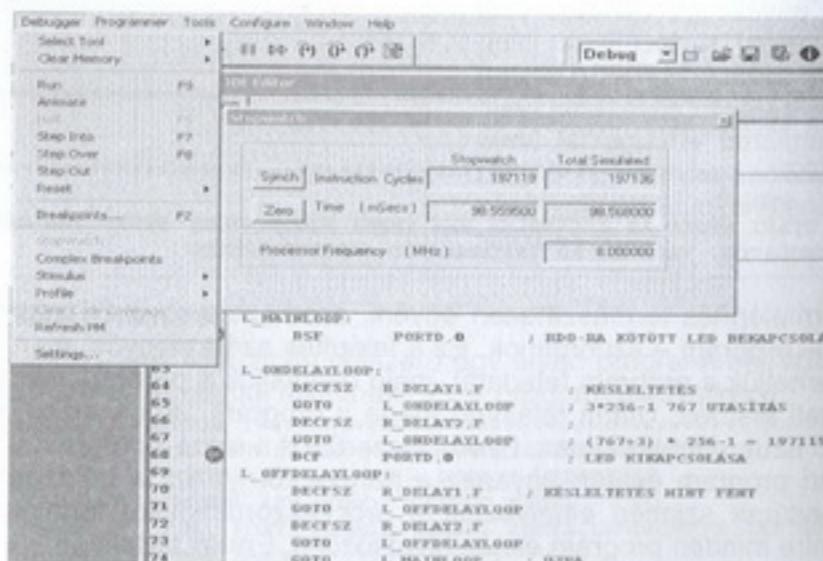
Ez lesz az első, tényleges feladatot végrehajtó programunk. A PortD,0 kivezetésére kötött LED-et fogjuk villogtatni.

```

BCF    TRISD,0      ; RDO LÁB KIMENET
CLRF   R_DELAY1      ; KÉSL.REG. = 0
CLRF   R_DELAY2      ; " "
;-----[ LEDVILLOGTATÁS ]
L_MAINLOOP:
BSF    LATD,0      ; RDO-RA KÖTÖTT LED BEKAPCSOLÁSA
L_ONDELAYLOOP:
DECFSZ R_DELAY1,F   ; KÉSLELTETÉS
BRA   L_ONDELAYLOOP ; 3*256-1 = 767 UTASÍTÁS
DECFSZ R_DELAY2,F   ;
BRA   L_ONDELAYLOOP ; (767+3)*256-1 = 197119 UTASÍTÁS
BCF    LATD,0      ; LED KIKAPCSOLÁSA
L_OFFDELAYLOOP:
DECFSZ R_DELAY1,F   ; KÉSLELTETÉS MINT FENT
BRA   L_OFFDELAYLOOP
DECFSZ R_DELAY2,F   ;
BRA   L_OFFDELAYLOOP
BRA   L_MAINLOOP    ; ÚJRA

```

A megjegyzések alapján a programrész érhető. Először mivel RESET után minden láb bemenet, a TRISD regiszter 0. bitjének 0-ba írásával kimenetnek állítjuk a PORTD,0 lábat. Majd a lábat 1-be állítva bekapcsoljuk a LED-et.



8.5. ábra
A stopper használata

Utána várunk egy ideig (L_ONDELAYLOOP). R_DELAY1 regiszter értékét 1-gel csökkentjük (DECFSZ – DECrement File register Skip if Zero), és a következő utasítást átlépjük (skip), ha a csökkentés eredményeként a regiszter tartalma nulla lesz. Különben

8. fejezet: PIC programozás assembler segítségével

nem lépjük át, visszaugrunk (BRA) a csökkentő utasításra. Mivel az első csökkentéskor a 0-s tartalom FF-re vált, ezért összesen 256-szor fogjuk az első két utasítást végrehajtani (csökkentés + ugrás), és utoljára átlépjük a visszaugrást (R_DELAY1 tartalma 0), és végrehajtjuk a következő DECFSZ R_DELAY2,F utasítást. Mivel elsőnek itt is a nullát csökkentettük 1-gel, ezért nem lépjük át, hanem visszalépünk az első ciklus elejére. Ezt ismét 256-szor végrehajtjuk, majd R_DELAY2 értékét ismét 1-gel csökkentjük. 256-szor hajtjuk végre az első ciklust, utána R_DELAY2 értékét is 256-szor csökkentjük.

Az első ciklust 256-szor hajtjuk végre, ahol minden ciklusmag 3 ciklusból áll (csökkentés 1 ciklus, ugrás 2 ciklus), kivéve az utolsót, amikor a feltétel teljesül, ez összesen $3*256-1 = 767$ utasítás. Az R_DELAY2-t növelő ciklus ideje: a külső ciklus idejéhez még hozzáadódik a vizsgálat és ugrás, $767+3 = 770$ ciklus, ezt 256-szor hajtjuk végre, de az utolsó utasítás csak 2 ciklus hosszúságú: $770*256-1 = 197\,119$ ciklus lesz a teljes késleteti rutin végrehajtási ciklusainak a száma. Ha az órajel-frekvencia 8 MHz, akkor a ciklusidő $0,5 \mu s$. A késletetési idő: $197\,119 * 0,5 \mu s = 98\,559,5 \mu s \sim 0,1 s$.

A BCF LATD,0 utasítással a LED-et kikapcsoljuk, hasonló módon a LED kikapcsolási idejét is 0,1 s-ra választjuk. Utána ez folyamatosan ismétlődik a visszaugrás miatt (BRA L_MAINLOOP).

Szimulációval is ellenőrizhetjük a késletetést. Ehhez a Debugger → Select Tool → MPLAB_SIM menüpontot aktivizáljuk, rákattintással. A Debugger → Settings → Osc/Trace fülön beállíthatjuk a processzor frekvenciáját 8 MHz-re. Utána kiválasztjuk a Debugger → StopWatch menüpontot.

Mivel ugyanazt a késletető programrészletet használjuk kétszer egymás után, ezért célszerű ezt szubrutinná átalakítani. Adunk neki egy nevet (S_DLY), a programunk végére másoljuk, és a befejezéséhez egy RETURN utasítást helyezünk el. Felhasználjuk a BTG (Bit ToGgle) bitváltó utasítást, ami a bitet minden az ellentétre váltja. Programunk jóval rövidebb lett:

```

;-----[LEDVILLOGTATÁS: 2. VÁLTOZAT ]
L_UJRA
CALL   S_DLY
BTG    LATD,0
BRA   L_UJRA
===== [S Z U B R U T I N O K ] =====
S_DLY
DECFSZ R_DELAY1,F   ; KÉSLELTETÉS
BRA   S_DLY          ; 3*256-1 = 767 UTASÍTÁS
DECFSZ R_DELAY2,F   ;
BRA   S_DLY          ; (767+3)*256-1 = 197119 UTASÍTÁS
RETURN

```

Harmadik részfeladatként egy nyomógomb állapotát jelenítjük meg a LED-en. Itt felhasználjuk a karakterfüzér helyettesítő #define direktívát, ami az argumentumában szereplő első füzért a betűközzel elválasztott második füzérre cseréli. Ez a megadás sokkal jobb, mert a programban csak két formális szimbólum fog szerepelni: GOMB és LED. A fizikai hozzárendelés külön szerepel, és bármikor megváltoztatható. A nyomógomb megnagyomása alacsony szintre állítja a bemenetet, a LED pedig magas szinten világít. Elsőnek a LED-et kikapcsoljuk (BCF LED). Utána megvizsgáljuk a GOMB állapotát: BTFS – Bit Test Fileregister Skip if Set. Ha GOMB, vagyis PORTB,0 bit értéke 0 (a gomb nyomva), akkor nem lépjük át a BSF LED utasítást, vagyis a LED-et kigyűjtjük. Különben átlépjük ezt az utasítást, és mivel először a LED-et kikapcsoltuk, az úgy marad. Utána ciklikusan ismételjük ezeket a lépéseket.

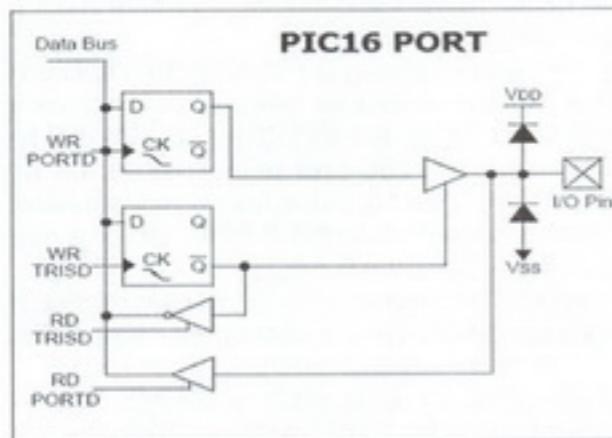
```

;-----[ GOMB ÁLLAPOT LEDRE ]
#DEFINE GOMB  PORTB,0 ; SZTRINGHELYETTESÍTŐ DIREKTÍVA AKTÍV ALACSONY
#DEFINE LED   LATD,0 ; SZTRINGHELYETTESÍTŐ DIREKTÍVA AKTÍV MAGAS

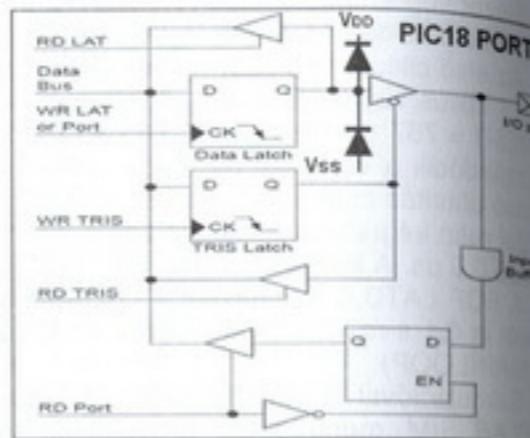
```

;EZ A DEFINÍCIÓ ITT IS LEHET, DE JOBB A DEFINÍCIÓK KÖZÖTT ELHELYEZNI
L_UJRA1:

| | | |
|------|---------|--|
| BCF | LED | ; LED KI |
| BTFS | GOMB | ; HA NEM NYOMTUK MEG, A KÖVETKEZŐ UT.-T ÁTLÉPI |
| BSF | LED | ;MEGNYOMTUK, VILÁGÍT |
| BRA | L_UJRA1 | |



Tárolóba írunk, de lábról olvasunk.
Lábhoz tartozó bit kezelése:
READ – MODIFY – WRITE

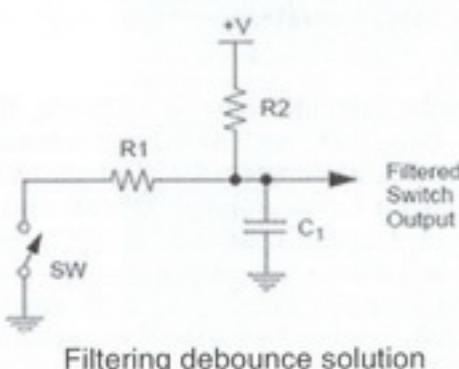


LATCH regiszter teszi lehetővé a portbeolvasás igénylő READ – MODIFY – WRITE utasítás végrehajtását, anélkül, hogy a kimenet helyes állapotának megőrzése miatt a portra kiküldött értéket egy tükröregiszterben tárolnánk.

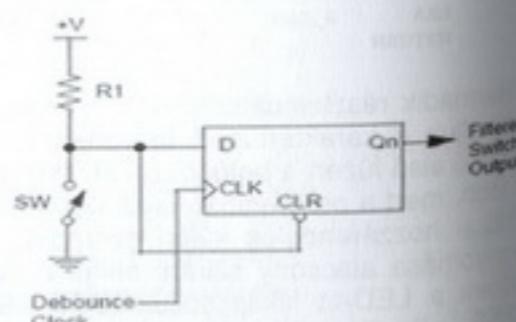
8.6. ábra
PIC16-PIC18 I/O láb áramkör

8.5.3. bcnt_02.asm

A feladat egy nyomógomb megnyomásainak számlálása és bináris alakban egy LED-soron történő megjelenítése. Mechanikus érintkezőket tartalmazó nyomógombok esetén az érintkezők rugalmas kapcsolódása miatt a zárt és nyitott állapotok több érintkezés-szétváltás folyamatán keresztül valósulnak meg, amint ezt a 8.7. ábrán is illusztráltuk. A jelenség neve prell, aminek kiszürésére (prellmentesítés – debouncing) áramköri megoldás is lehetséges: egy megfelelő RC tag mint szűrő alkalmazása, illetve egy D térfogó felhasználása.



Filtering debounce solution



Shift register debounce solution

8.7. ábra
Prellmentesítés

A billentyűfigyelő rutinban elhelyezett késleltető hurok is megoldása a prellmentesítésnek (8.8. ábra).

Ez a prell: a perges ideje kb. 10 ms.

Algoritmus:

- figyelni a billentyűnyomást
- várakozás
- figyelni a billentyű elengedését.

MEGNYOMÁS ELENGEDÉS



; = [PRELLMENTESÍTÉS (DEBOUNCING)] =

```

S_VAR_GOMB
    BTFS  GOMB
    BRA   S_VAR_GOMB ; MÉG NEM NYOMTÁK MEG
                      ; MEGNYOMTÁK
    CALL  S_DLY
    L_ELENGED
    BTFS  GOMB
    BRA   L_ELENGED ; MÉG NEM ENGEDTÉK EL
    CALL  S_DLY
    RETURN ; ELENGEDTÉK

```

8.8. ábra
Prellmentesítés programmal

Kevés nyomógomb esetén célszerű egy nyomógombhoz a nyomógomb megnyomásának a hosszát mint eseményt hozzárendelni, vagyis a rövid és hosszú gombnyomást megkülönböztetni. Ezt a Z jelzőbit állapota fogja jelezni. A rutin meghívásakor addig várakozik, míg a gombot meg nem nyomjuk. Utána R_VAR értékét feltöljük egy számmal, esetünkben 8-cal, ez lesz a hosszú idejű várakozás időtartama. Egy ciklusban (kezdete: L_VE) késleltetési ideig várakozunk, és utána megvizsgáljuk, hogy még meg van-e nyomva a gomb. Ha elengedtük, akkor a program végére megyünk. Ha még nyomva van a gomb, R_VAR értékét eggyel csökkentjük, ha még nem nulla, különben nulla értéken „beragad”. A MOVF r_var,f látszólag értelmetlen utasítás egyetlen szerepe az operandus nulla állapotát jelző Z jelzőbit állítása anélkül, hogy más változót felhasználnánk. Ha R_VAR = 0, akkor Z értéke 1 lesz, és R_VAR értéke nem változik, hiszen önmagára másoljuk.

; FLAG 0 - RÖVID NYOMÁS, HA 1 - HOSSZÚ NYOMÁS
S_VAR_LONG

```

L_VE
    BTFS  GOMB
    BRA   S_VAR_LONG ; MÉG NEM NYOMTÁK MEG
                      ; MEGNYOMTÁK
    MOVLW .8
    MOVWF R_VAR ; GOMB NYOMVATARTÁSI IDŐ
    CALL  S_DLY
    BTFS  GOMB
    BRA   L_VEGE ; NYOMVA?
    MOVP R_VAR,F ; HA ELENGEDTEK
    BTFS  STATUS,Z ; HA TOVÁBB NYOMJÁK ÉS R_VAR=0?
    DECP R_VAR,F ; MÉG NEM NULLA R_VAR ÉS CSÖKKENTJÜK
    BRA   L_VE ; NULLA, ÚJABB VÁRAKOZÁS
    MOVP R_VAR,F ; EZ AZ UTASÍTÁS CSAK A Z JELZŐBITET ÁLLITJA.
    RETURN

```

8.5.4. rotate_03.asm

A feladat egy folyamatosan mozgó fény (futófény) előállítása. Ez úgy történik, hogy egy LED-soron mindenkor egy, a soron következő LED-et gyújtunk ki. Kezdőértékként pl. 00000001-et írunk abba a regiszterbe, aminek a tartalmát kiírjuk a LED-sorra, majd ezt az értéket forgatjuk a következő pozícióba.

A PIC16-os családban a jobbra és balra forgatás csak a C jelzőbit felhasználásával (azon keresztül) történhet (RLC, RRC – *Rotate Right through Carry*). Azért valósították meg ezt, mert így egy regiszter bármelyik bitjét forgatással a C-be juttathatjuk, és az érték kétől függően tudunk a programban elágazni (pl. BTFSS STATUS.C).

Ha C felhasználása nélkül kívánunk forgatni, ezt a következő két utasítás segítségével valósíthatjuk meg (C bit és W regiszter értéke elromlik, jelöljük REG bitjeit a 76543210 módon):

| | | | |
|-----------|-------------|------------|---------|
| RRF REG,W | ;R=76543210 | W=C7654321 | C=REG.0 |
| RRF REG,F | ;R=07654321 | W=C7654321 | C=REG.0 |

A PIC18-as családban már létezik a regiszterek önmagukban való forgatása (no carry) is: RRNCF, RLNCF utasítások.

Mivel a PIC18 család több utasítással rendelkezik, ezért a 16–18 közötti átkódolást segítendő bemutatunk egy bővíthető makrógyűjteményt, amely PIC18 utasítások PIC16-os kódban való megadását teszi lehetővé makrók használatával.

Ezzel könnyebb a 18-ra készült forráskódot 16-os kódába átírni. A makrók nevében megtartottuk az eredeti PIC18-as utasításmnemonikont, csak egy M_ előtag előzi meg. Természetesen a makrók használatánál mindenkor minden regiszter szem előtt, hogy a makróban lévő kód bizonyos változók értékeit elronthatja, ezért a változók esetleges mentéséről mindenkor gondoskodjunk.

```

; KONSTANST -> REGISZTERBE, WREG-ET RONTJA
M_MOVLF MACRO LIT,REG
    MOVlw LIT
    MOVwf REG
ENDM

; REG2 := REG1 W ROMLIK!
M_MOVFF MACRO REG1,REG2
    MOVF REG1,W
    MOVWF REG2
ENDM

; BALRA ROTÁLÁS CY NÉLKÜL
M_RRNCF MACRO REG ;W,C ROMLIK
    RRF REG,W
    RRF REG,F
ENDM

; JOBBRA ROTÁLÁS CY NÉLKÜL
M_RLNCF MACRO REG ;W,C ROMLIK
    RLF REG,W
    RLF REG,F
ENDM

; BITBILLEGTEKÖTŐ MAKRÓ
M_BTG MACRO REG,PIN
    LOCAL L_NULLARA,L_EGYRE,L_VEG
    BTFSS REG,PIN
    GOTO L_EGYRE
L_NULLARA
    BCF REG,PIN
    GOTO L_VEG
L_EGYRE
    BSF REG,PIN
L_VEG
ENDM

; REGISZTER = FF
M_SETF MACRO REG
    CLRF REG
    DECF REG,F
ENDM

; BZ FELTÉTELES UGRÁS
M_BZ MACRO CIMKE
    BTFSC STATUS,2
    GOTO CIMKE
ENDM

```

A DVD mellékletben szereplő forráskód ezek alapján már érhető.

8/12, illetve 8/14 mikrovezérlők programozásánál a MASM megengedi a következő, valójában nem szereplő általános (pszeudo-) utasítások használatát a programban, és assembláláskor rendeli hozzá a megfelelő létező utasítást.

PL ADDCF f,d helyett

BTFSC 3,0

INCF f,d utasításpárt.

Ezek a pszeudoutasítások:

Cy, illetve DC hozzáadása egy regiszterhez: ADDCF, ADDDCF

B

BC, BDC, BNC, BNDC, BNZ, BZ

CLRC, CLRDC, CLRZ

LCALL, LGOTO

MOVF

NEGF

SETC, SETDC, SETZ

Következő utasítás feltétele átlépése (Skip): SKPC, SKPDC, SKPNC, SKPNDC, SKPNZ

8.5.5. dled_04.asm

A feladat egy analóg értéknek megfelelő bináris szám kiküldése a LED-sorra. A PIC-eknél egy láb lehet digitális bemenet vagy kimenet, illetve analóg bemenet. Reset esetén ez utóbbit az alapértelmezett.

Digitális I/O esetén a TRIS regiszter tartalma határozza meg, hogy egy láb digitális bemenet vagy kimenet. A PIC16-os családnál azon kivezetések működését, amelyek analóg bemenetek is lehetnek, az ANSEL, illetve az ANSELH regiszterek határozzák meg (1 – analóg; 0 – digitális).

```

; HA AZT AKARJUK, HOGY B PORT DIGITÁLIS BEMENET LEGYEN:
BSF STATUS,RP0          ;
BSF STATUS,RP1          ; REGISZTER BANK3 VÁLASZTÁSA
MOVLM 0X00
BSWF ANSELH             ; PORTB DIGITÁLIS BEMENETEL LETT
BSF STATUS,RP0          ;
BSF STATUS,RP1          ; REGISZTER BANK0 VÁLASZTÁSA

```

Az A/D átalakító inicializálásához a PIC18-as családnál az ADCON0, ADCON1, ADCON2 regisztereit kell beállítani.

A mintapéldában második feladatként a futófény sebességét és irányát a potenciométer forgatásával szabályozhatjuk. Önálló feladatként elemezzük, hogyan működik ez a programrész!

8.5.6. blinkpol_05.asm

A feladat egy LED-sor léptetése hardverkésleltetéssel: a TMR0 számláló túlcsordulásának a figyelésével. A számláló túlcsordulását az INTCON regiszter TMR0IF = 1 állapota jelzi, amit érzékelés után a programban kell törölni, előkészülni a következő túlcsorduláshoz. A számláló (és minden periféria) kezelésének két módja van:

Polling: folyamatosan figyeljük a számláló túlcsordulását, ha megtörténik, végrehajtjuk a tevékenységet.

Megszakítás: a számláló túlcsordulása megszakítást okoz. Az eredetileg futó, megszakított program állapota elmentődik, és a megszakítási alprogram hajtja végre a tevékenységet, befejezésekor visszaadódik a vezérlés a főprogramnak.



A programban először TMR0 időzítő és D port (itt a LED-sor) működésmódját állítjuk be. A nyomógombbal tudjuk a számlálás irányát (fel/le) beállítani.

```

L_INI_TMR0
    MOVLW B'10000010'      ; TIMERO INDÍTÁSA. LÉPTETI A PROCESSZORÓRAJEL
    MOVWF TOCON             ; MAXIMÁLIS ELŐOSZTÓ

L_INI_PORTS
    CLRF TRISD             ; LATD KIMENET
    CLRF LATD

L_HUROK
    BTFFS INTCON,TMROIF   ; ITT VÁRUNK, AMIG TIMERO TÚLCSORDUL
    BRA L_HUROK
    BCF INTCON,TMROIF     ; JELZŐBITET PROGRAMBÓL KELL TÖRLNI
    BTFFS GOMB              ; ELŐL DEFINIÁLTUK: #DEFINE GOMB PORTB,0
    BRA L_ATLEP
    DECF LATD
    BRA L_HUROK
    L_ATLEP INCF LATD      ; INCREMENT DISPLAY VARIABLE
    BRA L_HUROK

```

Önálló feladatként vizsgáljuk meg, hogy hogyan tudjuk a villogás idejét megváltoztatni, és ezt a program melyik részében kell megtenni.

8.5.7. ittabla_06.asm

A feladat hasonló a futófényhez, azzal a különbséggel, hogy egy táblázatban adott bitcsoport-sorozatot küldünk a kijelzőre, valamint a késleltetést megszakítással valósítjuk meg.

Elsőként két praktikus makrót mutatunk be. M_MOVLF makró az utasításkészlet „hiányosságát” hivatott pótolni: a segítségével egy regisztert tölt fel egy konstanssal. (Vigyázunk! A WREG tartalmát elrontja!) Az M_TCIMTOLT makró segítségével egy táblázat kezdőcímét irjuk be a táblakezeléshez szükséges mutatóba (TBLPTR).

```

*****[ M A K R O K ]*****
;-----[ M_OVLF MACRO LIT,REG ;KONSTANST -> REGISZTERBE, WREG-ET RONTJA
    MOVLW LIT
    MOVWF REG
    ENDM

;-----[ TÁBLA KEZDŐCÍM TBLPTR-BE]
M_TCIMTOLT MACRO TBLCIM ;TÁBLA KEZDŐCÍM MEGÁLLAPÍTÁSA
    MOVLW UPPER(TBLCIM)
    MOVWF TBLPTRU
    MOVLW HIGH(TBLCIM)
    MOVWF TBLPTRH
    MOVLW LOW(TBLCIM)
    MOVWF TBLPTRL
    ENDM

*****[ PROGRAMKÓD KEZDETÉ ]*****
    ORG 0X0000

L_INI_CLK
    MOVLW B'01110010'      ;CLOCK BEÁLLÍTÁS 8 MHZ BELSÓ
    MOVWF OSCCON
    BRA L_MAIN             ;UGRÁS A FÓPROGRAM KEZDETÉRE

    ORG 0X0008             ;MAGAS PRIORITÁSÚ MEGSZAKÍTÁS KEZDŐCÍME

L_IT_HIGH
    BTFFS INTCON,TMROIF   ;TMRO MEGSZAKÍTÁS?
    RETFIE FAST            ;NEM
    BCF INTCON,TMROIF     ;IGEN, TÖRLÉS
    CALL S_KIIR             ;A VÉGREHAJTANDÓ PROGRAM
    RETFIE FAST

L_MAIN:
L_INI_PORTS
    CLRF TRISD             ; LATD KIMENET
    CLRF LATD

L_INI_TMR0
    MOVLW B'10010000'      ;8MHZ ORA 0.5 MIKROSEC SZÁMLÁLÓ LÉPTETÉS!!!!!!
    MOVWF TOCON

```

```

L_INI_IT
    CLRF INTCON           ;MEGSZAKÍTÁS INICIALIZÁSA
    BSF INTCON,TMROIE     ;BEBILLENT IT BITEK TÖRLÉSE
    BSF INTCON,GIE        ;GENERAL IT ENG
    M_TCIMTOLT T_ALAK     ;TÁBLA KEZDŐCÍM TÁBLAMUTATÓBA
    BRA $                  ;UGRÁS ÖNMAGÁRA, EZ A FÓPROGRAM!!!
    *****[ S Z U B R U T I N O K ]=====
S_KIIR
    TBLRD *+               ;MUTATOTT TÁBLAELEM LEDSORRA, KIOLVAS+NÖVEL
    MOVF TABLAT,F          ;NULLA A TARTALOM, HA IGEN ÚJRAININDÍT
    BZ L_UJOLT
    MOVFF TABLAT,LATD      ;KIVITEL LEDSORRA
    RETURN

L_UJOLT
    M_TCIMTOLT T_ALAK     ;ISMÉT A TÁBLÁZAT ELEJÉRE ÁLLÍTJUK A MUTATÓT
    RETURN

*****[ T Á B L Á Z A T ]=====
T_ALAK
    DE B'11111111',B'11100111'
    DE B'11000011',B'10000001'
    DE B'11000011',B'11100111'
    DE B'11111111',B'11111111'
    DE B'00000000'
    END

```

A program elején megtörténnek az inicializálások (processzor-órajel, port, TMRO, megszakítás). Ezután a makró felhasználásával feltöltjük a táblamutatót a táblázat kezdőcímével. A fóprogram nem csinál semmit, önmagára ugrik, a tevékenységet a megszakítási alprogram végzi majd el, aminek végrehajtása minden számlálótúlcorduláskor fog bekövetkezni. Az S_KIIR rutin működése egyszerű: kiolvassuk és a LED-portra kivisszük a következő táblaelementet. Utána megvizsgáljuk, hogy elértek-e a táblázat végét. A táblázat megadása a DE direktívával történik, mert ilyen módon minden szomszédos memóriarekeszbe kerül egy érték. (DB direktíva esetén egy 16 bites utasítás egyik bájtjába kerülne érték, a másik bájt tartalma 0 lenne.)

Mivel a PIC16-os családban nincs külön táblázatkezelés, ott a RETLW utasításokban szereplő konstansok olvashatók ki az utasításokat tároló programmemoriából. További különbség, hogy nincs automatikus regiszterményes, ezért a megszakítási program elején el kell menteni, a végén vissza kell állítani a használt regisztereiket. Maga a táblakezelés a következő:

Az R_OFFSET regiszterben van az, hogy hányadik táblaelementet választjuk. Szubrutinhíváskor a PC alsó felébe (PCL-be) a szubrutin kezdőcímére kerül – hiszen a szubrutinhívás egy elugrás, az ADDWF PCL,F utasítás PCL-hez hozzáadja az előzőleg a W regiszterbe tett R_OFFSET értékét. Az így kialakult új PCL-tartalom határozza meg a következő végrehajtandó utasítás címét, ami egy RETLW utasítás lesz. Vagyis a szubrutinból viszszatérve W regiszter fogja tartalmazni a táblaelemkonstansot. Az elmeleti anyagban leírtuk, hogy PC alsó 8 bit feletti PCH értékét a PCLATH regiszterből kapja. Itt ezzel mi nem fogalkoztunk, mivel PCLATH kezdet értéke nulla, és rövid programunk végrehajtása során PCH értéke is nulla (amíg az első 256 utasítást hajtjuk végre). DT – Define Table – direktíva helyére az assembler RETLW utasítást helyez (DT .23 ugyanaz, mint RETLW .23).

```

    CLRF R_OFFSET ;EZ A TÁBLAPOZÍCIÓ REGISZTER
    GOTO $          ;UGRÁS ÖNMAGÁRA, EZ A FÓPROGRAM!!!
    *****[ S Z U B R U T I N O K ]=====

S_KIIR
    MOVF R_OFFSET,W      ; W = OPSZET (ELTOLÁS)
    CALL T_ALAK
    MOVWF R_TEMP
    MOVF R_TEMP,F

```

```

BTFSC STATUS, Z
GOTO L_UJRA
MOVWF PORTD
INCF R_OFFSET
RETURN

L_UJRA
CLRF R_OFFSET
RETURN
;-----[T Á B L Á Z A T ]-----
T_ALAK
ADDWF PCL, F           ; PC ALSÓ FELE=TÁBLAKCIM + R_OFFSET
DT    B'11111111', B'11100111' ; RETLW SOROSZAT DEFINIÁLÁSA.
DT    B'11000011', B'10000001'
DT    B'11000011', B'11100111'
DT    B'11111111', B'11111111'
DT    B'00000000'

;-----[ S Z U B R U T I N O K ]-----
S_DSP_KIV
CALL S_MER_ADO          ; MÉRÉS EREDMÉNYE WREG-BE
ANDLW B'11100000'         ; FELSŐ 3 BITJÉT LEVÁLASZTJUK
RRNCF WREG               ; WREG = 0XXX0000
SWAPP WREG               ; WREG = 00000XXX
BZ   L_BIT50 ; HA W=0, AKKOR TOVÁBBI VIZSGÁLAT
L_NOVEL ADDWF TBLPTRL, F ; TBLPTR:=TBLPTR+1
CLRF WREG
ADDMPC TBLPTRH, F
ADDMPC TBLPTRU, F        ; 3 BÁJTOS ÖSSZEADÁS!
RETURN

L_BIT50
BTFS S, 4                ; HA A MÉRÉS 4. BITJE 0,
DECFS WREG, F             ; AKKOR A LEGELŐSÖ ELEMET VÁLASZTJUK
BRA L_NOVEL

T_KIVE
        ; LEDEK          ; MÉRT FESZÜLTSÉG FELSŐ BITJEI
DE    B'00000000', B'00000001' ; 0000 0001
DE    B'00000011', B'00000111' ; 001 010
DE    B'00001111', B'00011111' ; 011 100
DE    B'00111111', B'01111111' ; 101 110
DE    B'11111111', B'11111111' ; 111

```

8.5.8. adtabla_07.asm

A cél az AD-mérés eredményének kivezérlésszerű megjelenítése a LED-soron, vagyis a kigyújtott LED-ek száma a feszültséggel lesz arányos. Ez az előző feladatok részeinek a felhasználásával oldható meg. Ha végigondoljuk a feladatot, ez nem bonyolult: egy táblaelem kiválasztását a mérési eredmény felső három bitje határozza meg.

Itt csak a PIC18-as családra írt fontosabb kód részleteket mutatjuk be. A főprogram nagyon egyszerű: Végtelen ciklusban a táblamutatót feltöljtük, meghívjuk az AD-mérést elvégző, majd ez alapján a táblaelemet kiválasztó és a LED-soron megjelenítő szubrutint.

Ez a rutin elvégzi a mérést, leválasztja felső három bitjét, ami a táblaelem sorszámát adja. Ezt kell hozzáadni a táblamutatóhoz, de a túlcordulást is figyelni kell. Ha belegen dolunk, nem nyolc, hanem kilenc kijelzendő állapotunk van. Amikor nem ég egyetlen LED sem, ez tartozzon ahhoz az állapothoz, amikor a mérési eredmény felső 4 bitje nulla.

```

L_CIKL_MAIN
    _TCIMTOLT      T_KIVE
    CALL S_DSP_KIV
    TBLRD *
    MOVFF TABLAT, PORTD
    BRA L_CIKL_MAIN

;-----[ S Z U B R U T I N O K ]-----
S_DSP_KIV
    CALL S_MER_ADO          ; MÉRÉS EREDMÉNYE WREG-BE
    ANDLW B'11100000'         ; FELSŐ 3 BITJÉT LEVÁLASZTJUK
    RRNCF WREG               ; WREG = 0XXX0000
    SWAPP WREG               ; WREG = 00000XXX
    BZ   L_BIT50 ; HA W=0, AKKOR TOVÁBBI VIZSGÁLAT
L_NOVEL ADDWF TBLPTRL, F ; TBLPTR:=TBLPTR+1
    CLRF WREG
    ADDMPC TBLPTRH, F
    ADDMPC TBLPTRU, F        ; 3 BÁJTOS ÖSSZEADÁS!
    RETURN

L_BIT50
    BTFS S, 4                ; HA A MÉRÉS 4. BITJE 0,
    DECFS WREG, F             ; AKKOR A LEGELŐSÖ ELEMET VÁLASZTJUK
    BRA L_NOVEL

T_KIVE
        ; LEDEK          ; MÉRT FESZÜLTSÉG FELSŐ BITJEI
    DE    B'00000000', B'00000001' ; 0000 0001
    DE    B'00000011', B'00000111' ; 001 010
    DE    B'00001111', B'00011111' ; 011 100
    DE    B'00111111', B'01111111' ; 101 110
    DE    B'11111111', B'11111111' ; 111

```

8. fejezet: PIC programozás assembler segítségével

8.5.9. indir_08.asm

A következő feladat az indirekt címzést mutatja be: 16 adatot gyűjtünk gombnyomásra, a LED-soron a hátralévő darabszám. Utána kiszámítjuk a mért adatok átlagát, és ezt a LED-soron kijelezzük.

A PIC16-os családban az FSR regiszter tartalma határozza meg annak a regiszternek a címét, amelynek tartalmával az utasítás által kijelölt műveletet elvégezzük. Hogy ne kelljen külön utasításformát használni, ezért a direkt című utasításokat használjuk. Ha operandusként nullát adunk meg, akkor indirekt a címzés. Vagyis ha az operandus címe nulla, akkor nem a nullás című regiszter tartalmával, hanem az FSR regiszter tartalma által meghatározott című regiszter tartalmával végezzük el az utasítás által meghatározott műveletet. Azért előnyös ez, mert formailag a direkt és indirekt címzés azonos formájú lesz, az operandus címe kódolja a különbséget, természetesen azzal a veszteséggel, hogy a kódoló regisztercím mögött nincsen tényleges regiszter.

A PIC18-as családban ezt két irányban is továbbfejlesztették: nem egy, hanem három indirekt címet hordozó regiszter van: FSR0, FSR1, FSR2, valamint a mutatott regiszterrel elvégzett művelet mellett a mutató értékét is meg tudja változtatni az utasítás. A program működése az alábbi kód részletből és a blokkvázlatból megérthető.

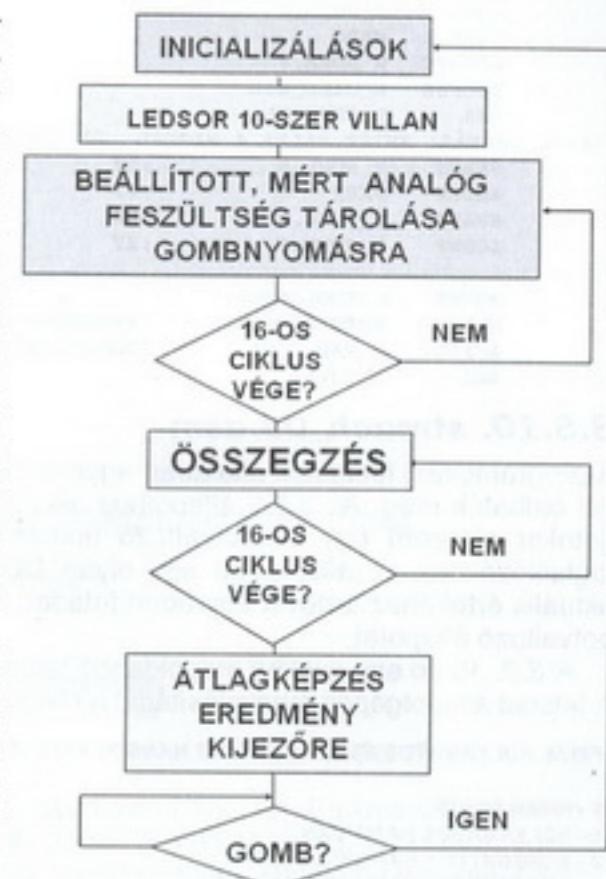
L_UJRA

```

    CLRF DSPPORT ;
    CLRF DDRPORT ; LEDSOR INIT
    SETF TRISB
    SETF PORTB
    MOVW .10 ; LEDSOR TIZSZER FELVILLAN
    CALL S_VILLOG
    MOVW .16
    MOVWF R_CIKL_NUM
    LPTR 0, R_PUFFER

L_CIKL_MAIN
    MOVFF R_CIKL_NUM, DSPPORT ; MARADÉK DARAB DSP
    CALL S_VAR_GOMB ; GOMBRA MÉRÉS
    CALL S_MER_ADO
    MOVFF WREG, POSTINC0 ; TÁROLÁS
    MOVFF WREG, DSPPORT ; MÉRT ÉRTÉK DSP
    DECF R_CIKL_NUM
    BNZ L_CIKL_MAIN ; CIKLUS
    CLRF DSPPORT
    MOVW .10
    CALL S_VILLOG ; ÚJABB 10 VILLOGÁS
    ADDWF R_CIKL_NUM, DSPPORT ; 16 ADAT ÖSSZEADÁSA
    CLRF R_SUMH
    CLRF R_SUML

```



```

MOVWF .16
MOVWF R_CIKL_NUM
MOVFF POSTDEC0,WREG ;UTOLSÓTÖL VISSZAFELÉ SZÁMOLUNK
L_SUMCIKL ;TÁROLT ADATOK ÖSSZEGZÉSE
    MOVFF POSTDEC0,WREG
    ADDWF R_SUML,F
    CLRF WREG
    ADDWFC R_SUMH,F
    DECFSZ R_CIKL_NUM
    BRA L_SUMCIKL
;16-AL OSZTÁS: SHIFT BALRA 4 BITTEL [0X:YZ]/16=XY
    SWAPP R_SUML,W ;ZY
    ANDLN OX0F ;0Y
    SWAPP R_SUMH,F ;X0
    IORWF R_SUMH,F ;XY
    MOVFF R_SUMH,WREG
    MOVFF WREG,DSPPORT ;ERedmény KIJELZÉSE
    RCALL S_VAR_GOMB ;GOMBNYOMÁS UTÁN minden előlről
    BRA L_UJRA

```

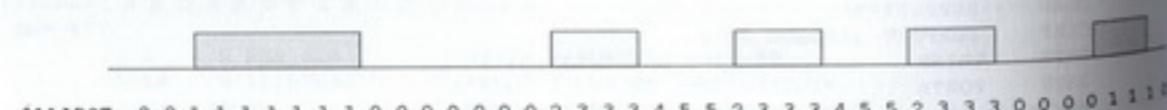
8.5.10. *stmach_09.asm*

A programozási feladatok általában egymást követő állapotok egymás utáni végrehajtásával oldhatók meg. Az adott állapothoz akciók tartoznak, amiket el kell végezni. Az állapotokat célszerű egy állapotváltozó numerikus értékével azonosítani. Programozással foglalkozóknak az állapotgép egy olyan **DO_CASE** ciklus, amelyben az állapotváltozó aktuális értékéhez tartozik egy adott feladat, és ez a feladat esetleg módosíthatja az állapotváltozó állapotát.

A 8.9. ábrán egy gyakori megoldandó feladatra ad megoldást az állapotgépes szemlélet. A feladat állapotgépes megvalósítását a DVD-melléklet **hang_mo.asm** fájl tartalmazza.

Példa: FOLYAMATOS ÉS SZAGGATOTT HANGOT KELL GENERÁLNUNK. AZ ÁLLAPOTOK:

- 0 - HANG NINCS
- 1 - FOLYAMATOS HANG VAN
- 2 - SZAGGATOTT HANG INDITÁSA
- 3 - SZAGGATOTT HANG IDEJE ELTELT?
- 4 - SZÜNET INDITÁSA
- 5 - SZÜNET HANG IDEJE ELTELT?



8.9. ábra
Hangprogramozás állapotgéppel

Mivel a gyakorlópanelen nincs hangkeltő eszköz, ezért LED-ek villogásával illusztráljuk a különböző állapotokat, az állapotok váltása nyomógomb aktivizálásával történik, de csak a megszakítás által kijelölt időpontban változhat. Az aktív állapot számkódját az **R_STATE** állapotváltozó tartalmazza.

```

; [ ALLAPOTGEP ]
S_TST_STATE
    M_ST_MACH 0,R_STATE,L_ST0 ;LED NEM ÉG
    M_ST_MACH 1,R_STATE,L_ST1 ;LED FOLYAMATOSAN ÉG
    M_ST_MACH 2,R_STATE,L_ST2 ;LED NEM ÉG
    M_ST_MACH 3,R_STATE,L_ST3 ;LED0 VILLOG
    M_ST_MACH 4,R_STATE,L_ST4 ;LED2 VILLOG
    M_ST_MACH 5,R_STATE,L_ST5 ;LED4 VILLOG

```

```

M_ST_MACH 6,R_STATE,L_ST6 ;LED6 VILLOG
M_ST_MACH 7,R_STATE,L_ST7 ;ALAPHELYZET
    L_ST0 CLR LATD
    BRA L_STVEG
    SETF LATD
    BRA L_STVEG
    CLRF LATD
    BRA L_STVEG
    BTG LATD,0
    BRA L_STVEG
    BCF LATD,0
    BTG LATD,2
    BRA L_STVEG
    BCF LATD,2
    BTG LATD,4
    BRA L_STVEG
    BCF LATD,4
    BTG LATD,6
    BRA L_STVEG
    CLRF R_STATE
    RETURN

```

Az állapotgép leírását az **M_ST_MACH** makró használata könnyíti meg.

```

M_ST_MACH MACRO C_STATE,R_STATE,L_STATE
    MOVLM C_STATE
    XORWF R_STATE,W
    BTFSC STATUS,Z
    BRA L_STATE ;HA EGYENLŐ AKKOR TEVÉKENYSÉG
    ENDM ;NEM EGYENLŐ, KÖV. ÁLLAPOT VIZSGÁLATA

```

8.5.11. *preempt_10.asm*

Egy program végrehajtandó részfeladatokból, taszkokból épül fel. Egyszerűbb esetben az egyik taszk befejezése után történik a következő taszk végrehajtása, a taszkok egymásnak adják át a vezérlést. Ez a kooperatív multitasking. Hatékonyabb megoldás a preemptív multitasking használata. Itt a taszkok futását egy ütemezőprogram (*scheduler*) végzi, amely lényegében időszeletekben futtatja a taszkokat.

Ütemezőként a főprogramban folyamatosan egy programhurok fut, amelyben a megszakítás által 1 ms, 10 ms, 100 ms, 1 s, 60 s időközönként 1-be állított jelzóbiteket vizsgálja meg. Ha valamelyik jelzobit állapota 1, akkor végrehajtja az abban az időszeletben elvégzendő eseményt, és törli a jelzobitet.

Megszakítási programban történik az eltelt idő figyelésével a jelzobitek 1-be állítása, akkor, ha az idő eltelt.

```

TMRREVENT MACRO TVAR,TLIT,TBITBYTE,TBIT,TVLABEL
    DECFSZ TVAR,F,A ;CSÖKKENTJÜK
    BRA TVLABEL ;HA NEM TELT LE, TOVÁBB
    MOVWF TLIT
    MOVWF TVAR,A ;MODULUS ÚJRAÍRÁSA
    BSF TBITBYTE,TBIT,A ;JELEZZÜK A TIMER EVENT-ET
    ENDM

```

- **tvar:** regiszter tartalmazza az eltelt időegységeket.
- **tlit:** konstans, ami megadja a jelzobit 1-be állításához szükséges időszeletek számát.
- **tbitbyte,tbit:** bites változó értéke akkor lesz 1, ha eltelt a tlit számú időszelet.
- **tvlabel:** az a címke, ahová akkor ugrunk, ha még nem telt el a tlit számú időszelet.

```

:-----[ ITT KEZDŐDIK A FÓPROGRAM HUROK ]-----  

L_MAIN_LOOP  

    BTFSS   B_TIMSEV      ;TIMER EVENT FIGYELÉSE  

    BRA     L_MAIN_LOOP_V ;HA NEM JÖTT EVENT TOVÁBB  

:-----[ 1MS-KÉNT VÉGREHAJTANDÓ TEENDÓK ]-----  

    BCF     B_TIMSEV      ;TIMER EVENT NYUGTÁZÁSA  

L_MAIN_LOOP_10MS  

    BTFSS   B_T10MSEV  

    BRA     L_MAIN_LOOP_V  

:-----[ 10MS-KÉNT VÉGREHAJTANDÓ TEENDÓK ]-----  

    BCF     B_T10MSEV  

L_MAIN_LOOP_100MS  

    BTFSS   B_T100MSEV,A  

    BRA     L_MAIN_LOOP_V  

:-----[ 100MS-KÉNT VÉGREHAJTANDÓ TEENDÓK ]-----  

    BTG LATD,0  

    BCF     B_T100MSEV,A  

L_MAIN_LOOP_1S  

    BTFSS   B_T1SEV,A  

    BRA     L_MAIN_LOOP_V  

:-----[ 1S-KÉNT VÉGREHAJTANDÓ TEENDÓK ]-----  

    BTG LATD,2  

    DECFSZ R_TOUT1  

    BRA     L_TOVA1  

    M_MOVLW C_TOUT1,R_TOUT1  

    BTG     LATD,4  

L_TOVA1 DECFSZ R_TOUT2  

    BRA     L_TOVA2  

    M_MOVLW C_TOUT2,R_TOUT2  

    BTG     LATD,7  

L_TOVA2 BCF     B_T1SEV,A  

L_MAIN_LOOP_60S  

    BTFSS   B_T60SEV,A  

    BRA     L_MAIN_LOOP_V  

:-----[ 60S-KÉNT VÉGREHAJTANDÓ TEENDÓK ]-----  

    BCF     B_T60SEV,A  

L_MAIN_LOOP_60M  

    BTFSS   B_T60MEV,A  

    BRA     L_MAIN_LOOP_V  

:-----[ 60M-KÉNT VÉGREHAJTANDÓ TEENDÓK ]-----  

    BCF     B_T60MEV,A  

L_MAIN_LOOP_24H  

    BTFSS   B_T24HEV,A  

    BRA     L_MAIN_LOOP_V  

:-----[ 24H-KÉNT VÉGREHAJTANDÓ TEENDÓK ]-----  

    BCF     B_T24HEV,A  

L_MAIN_LOOP_V  

    GOTO   L_MAIN_LOOP  

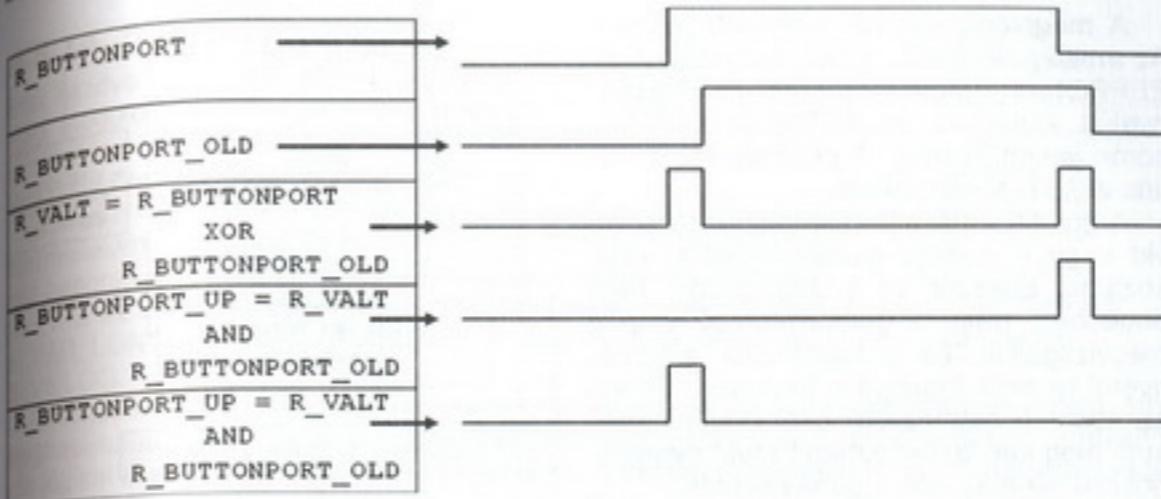
:-----[ A FÓPROGRAM HUROK VÉGE ]-----  


```

8.5.12. bilkez_11.asm

Az egyik leggyakoribb feladat a bemeneti változások figyelése, hiszen négy állapot lehetséges: a bemenet alacsony szintű, magas szintű, alacsony szintről magasra változott (ez a felfutó él), magas szintről alacsony szintre változott (lefutó él). Célszerű egy portot kijelölni a bemeneteknek: ez a legtöbb esetben a PORTB port, mert ezeken belső felhőzű ellenállások vannak. Ezeket bekapcsolva, külső áramköri áramköri elem nélkül tudunk biztosítani a bemenetek magas szintjét, amit egy kapcsoló alacsony szintre tud állítani.

A program azon alapul, hogy egy pl. 10 ms-os időszeletben beolvassuk a port aktuális állapotát, és annak az előző beolvásásból származó állapotával „kizárt vagy” (XOR) műveletet végzünk. A 8.10. ábrán ábrázoltuk, hogyan kapjuk meg a különböző változásokat.



8.10. ábra
Billentyűfigyelés

A billentyűkezelést végző szubrutin a következő:

```

S_END_INPUT
    MOVFF  C_INPUTPORT,WREG ;BEMENETI PORT OLVASÁSA
    XORIW  ;AHOL MEGNYOMTÁK, OTT 1 LESZ
    MOVWF  R_BUTTONPORT ;GOMBOK ÁLLAPOTÁNAK A TÁROLÁSA
    MOVWF  R_BUTTONPORT_TEMP ;ÉS MENTÉSE
    XORWF  R_BUTTONPORT_OLD,W ;VÁLTOZÁSFIGYELÉS
    ANDWF  R_BUTTONPORT_TEMP,W ;FELFUTÓ ÉLEK
    MOVWF  R_BUTTONPORT_UP ;MENTÉSE
    MOVFF  R_BUTTONPORT_TEMP,WREG ;W-BE VISSZATÖLTÉS
    XORWF  R_BUTTONPORT_OLD,W ;VÁLTOZÁSFIGYELÉS
    ANDWF  R_BUTTONPORT_OLD,W ;LEFUTÓ ÉLEK
    MOVWF  R_BUTTONPORT_DOWN ;MENTÉSE
    MOVFF  R_BUTTONPORT_TEMP,R_BUTTONPORT_OLD ;(RÉGI=UJ)
; ITT TÖRTÉNHET A GOMBESEMÉNYEK (EVENTEK) VIZSGÁLATA
    RETURN

```

8.5.13. eeprom_12.asm

Bizonyos adatok, beállítási paraméterek tárolására nagyon alkalmas az EEPROM memória. Ezeknek két típusát használhatjuk: vagy különálló tokban helyezkedik el, és a vezérlohöz vezetékenek kapcsolódik, vagy magában a vezérlőben kerül kialakításra. A PIC18-as családnál ez utóbbi minden családelemnél megvalósul.

Az EPROM-ok használatát az adatlapokban megadott írási és olvasási eljárások közlese segíti. Az EEPROM-tartalmat néhány 8 vagy 16 bites regiszterrel kezeljük (a regiszterek száma, hossza a memória méretétől és a vezérlő típusától függ):

- EECON – írás/olvasás vezérlő és állapot bitjeit tartalmazza;
- EEDAT – ide kell tölteni a memoriába kerülő értéket, itt jelenik meg kiolvasáskor;
- EEADR – a beírandó/kiolvasandó memória címe kerül ide.

Az EEPROM-kezelés négy szubrutin segítségével valósul meg:

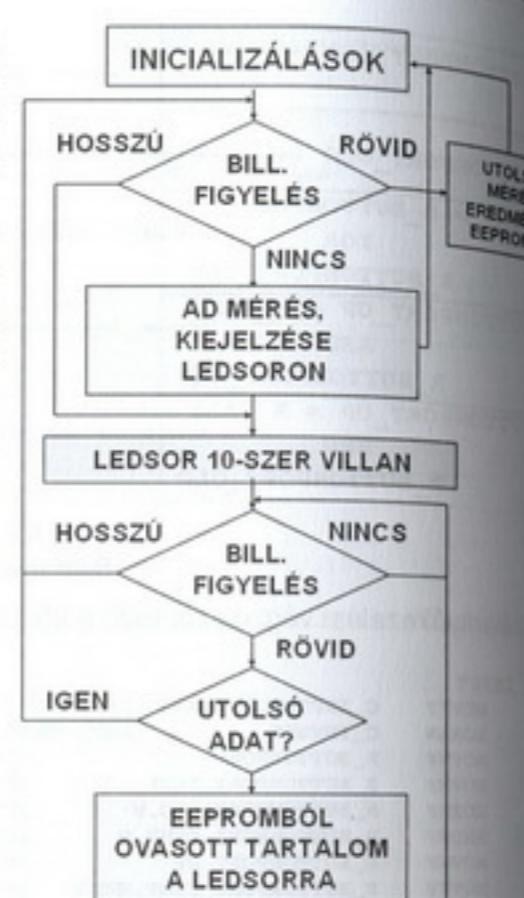
- S_RAMTOEE – az adatmemória-blokk EEPROM-ba írása,
- S_EEWRITE – egy bájt EEPROM-ba írása,

- S_EETORAM – az adatmemória-blokk EEPROM-ból kiolvasása,
- S_EEREAD – egy bájt kiolvasása EEPROM-ból.

A kezelt adatmemória-blokk kezdetét az R_DATA_SAVEFIRST, a végét az R_DATA_SAVELAST szimbólum azonosítja.

A megoldott feladat mérésadatgyűjtés. Az analóg méréseket gomb aktivizálásával EEPROM-ba tároljuk el, majd az eredményeket kiolvasva az EEPROM-ból LED-soron jelenítjük meg. A program blokkvázlata a 8.11. ábrán látható.

A gombfigyelésnél az eddigiekben a rövid vagy a hosszú gombnyomásra vártunk. Sokszor ez a „beragadás” nem engedhető meg, a gombállapotot csupán megvizsgáljuk, és a kiértékelés eredményétől tesszük függővé a folytatást. Ebben az esetben már három eseményünk van, amit meg kell különböztetni: rövid nyomás, hosszú nyomás, nincs gombnyomás.



8.11. ábra
EEPROM használata

```

;W=0 NINCS NYOMVA W=1 RÖVIDEN NYOMVA W=2 HOSSZAN NYOMVA
S_VAR_LONG
    BTFSC   GOMB
    RETLW   0           ;NEM NYOMTÁK MEG
    MOVLW   .8          ;MEGNYOMTÁK, GOMB NYOMVA IDŐ
    MOVWF   R_VAR
    CALL    S_DLY        ;KÉSLELTETÉS
    BTFSC   GOMB
    BRA    L_VEGE       ;HA ELENGEDTEK
    MOVF   R_VAR, F      ;HA TOVÁBB NYOMJÁK ÉS R_VAR=0?
    BTFSS   STATUS, Z
    DECF   R_VAR, F      ;MÉG NEM NULLA R_VAR ÉS CSÖKKENTJÜK
    BRA    L_VEGE       ;NULLA, ÚJABB VÁRAKOZÁS
L_VEGE
    MOVF   R_VAR, F
    BNZ    L_ROV
    RETLW   2
L_ROV
    RETLW   1

CALL    S_VAR_LONG
    BTFSC   WREG, 1
    BRA    L_LONG         ;HA 2 HOSSZÚ
    BTFSS   WREG, 0
    BRA    L_NONY        ;HA 0 - NONYOM
    BRA    L_SHORT        ;HA 1 - RÖVID
  
```

8.5.14. frekgen_13.asm

Ettől kezdve a feladatokat már a linker felhasználásával oldjuk meg, modulokra bontva az eddig egyetlen forrásfájlt. A szétválasztásnál célszerű a minden programban szereplő perifériakezelő részt önállóan szerepelte tenni.

A modul forráslistája, aminek a neve: perifini_mo.asm, a lemez mellékletben található. A felépítése a szokásos programokhoz hasonló, ez esetben lényegében egy szubrutiningyűjtemény, amit már az előző feladatok során megismertünk: S_INI_PERIF, S_VAR_LONG, S_DLY, S_DLYF, S_INI_AD, S_MER_AD, S_VILLOG.

Emlékeztetőül: A program- és adatmemóriát felosztjuk a mikrokontroller típusa által meghatározott fizikai szegmensekre. A programozó pedig a saját programját és adatváltató logikai szekciókra bontja. A linker feladata a logikai szekciókat a fizikai szegmensekbe elhelyezni. A szimbólumokkal jelölt összerendeléseket egy .lkr kiterjesztésű linker szkript fájl tartalmazza.

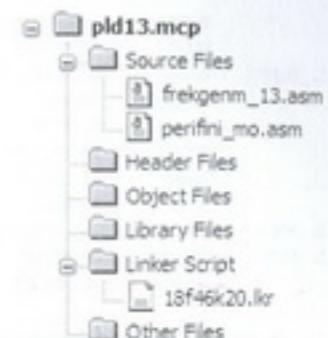
A konkrét feladat: 10 Hz–250 Hz között négyszögel generálása 10 Hz-es lépésekben. Mivel csak a periódusidőt tudjuk állítani, és annak a frekvenciával való kapcsolata 1/x típusú, ezért az adott frekvenciához tartozó periódusidőt egy táblázatból olvassuk ki. Az időalap 100 µs.

Két megközelítés:

- Fix 100 µs hosszúságú megszakítás generálása, ebben egy 16 bites regiszter csökkenése, amit előzőleg minden az adott frekvenciához tartozó 100 µs léptékben megadott számértékkal töltünk fel. Ha a regiszter elérte a nullát, akkor a kimenethez tartozó lábat ellenétesre változtatjuk.
- A timer0-t 16 bitesen használjuk. A bemenő órajele 100 µs hosszúságú. A számlálóba minden túlcordulásakor beírjuk a 65 536 periódusidő-értéket (a számláló felfelé számol). Amikor túlcordul, akkor újratöljük ezzel az értékkel, és a kimenethez tartozó lábat ellenétesre változtatjuk.

```

***** [ PROGRAMKÓD KEZDETÉ ] *****
RESPECT    CODE    0X0000
            CALL    S_INI_PERIF      ; PERIFÉRIÁK INICIALIZÁLÁSA
            BRA    L_MAIN          ; UGRÁS A FÓPROGRAMRA
;MEGSZAKÍTÁSKOR BE KELL ÍRNI A 16 BITES TMRO SZÁMLÁLÓBA A FREKVENCIÁNAK
;MEGFELELŐ SZÁMÉRTÉKET
L_IT_HIGH   CODE    0X0008
            BTFSS   INTCON, TMROIF   ; TMRO MEGSZAKÍTÁS?
            RETFIE  FAST             ; NEM
            BTG    BO_LAB          ; LÁB ELLENETTJEIRE
            BCF    INTCON, TMROIF   ; MEGSZAKÍTÁS JELZŐBIT TÖRLÉSE
            MOVFF   R_PERH, TMROH    ; 16 BITES SZÁMLÁLÓ ÚJRA TÖLTÉSE
            MOVFF   R_PERL, TMROL
            RETFIE  FAST
            CODE
L_MAIN
L_IT_MAIN
L_INI_TMRO
;MEZŐ OLA 0.5 MIKROSEC SZÁMLÁLÓ LÉPTETÉS!!!!!!!
            MOVLW   B'00000011'      ; 16 BITES SZÁML. 1:16 OSZTÓ 8 MIKROSEC LÉPÉS
            MOVWF   TOCON
L_INI_TABLE
            M_TCIMTOLT T_PERIODO
L_INI_IT
            CALL    S_KIIR          ; ELSŐ FREKVENCIA [R_PERH:R_PERL]-BE
            CLRF    INTCON
            BSF    TOCON, TMROON
  
```



```

BSF    INTCON, TMROIE   ;TMRO IT ENG
BSF    INTCON, GIE     ;GENERAL IT ENG
CALL   S_VAR_LONG      ;GOMBNYOMÁSRA LÉPTETJÜK A FREKVENCIÁT
BTFS C WREG, 1          ;HA 2-HOSSZÚ - ELÖLRŐL ÚJRA INDUL A LÉPTETÉS
BRA   L_RES             ;HA 0-NEM VOLT GOMBNYOMÁS
BTFS S WREG, 0          ;HA 1-RÖVID GOMBNYOMÁS, KÖVETKEZŐ FREKVENCIA
BRA   L_UJFI             ;HA 0-NEM VOLT GOMBNYOMÁS
BRA   L_NOV              ;HA 1-RÖVID GOMBNYOMÁS, KÖVETKEZŐ FREKVENCIA
L_RES  BRA   L_INIT_TABLE ; UJRA ELÖLRŐL
CALL   S_KIIR            ; KÖVETKEZŐ FREKVENCIA [R_PERH:R_PERL]-BE
BRA   L_UJFI

```

A program indítása után a táblázatbeli legkisebb frekvenciával fog villogni a 0. helyen lévő LED. Az első kiadott frekvencia legyen 1 Hz. Milyen értéket kell beírni a számlálóba? A rendszerrajel 8 MHz. Ennek a $\frac{1}{4}$ -e jut az előosztó bemenetére, vagyis 0,5 μ s lépteti. Mivel az előosztó 16-os osztásviszonnyal működik, a 16 bites számlálót 8 μ s periódus-idejű órajel lépteti. Ha a periódusidő 1 s, akkor a váltás 0,5 s-onként kell hogy bekövetkezzen. Vagyis a számlálónak $500\ 000/8=62\ 500$ beérkező órajel után kell túlcordulnia. Mivel a számláló felfelé számol ezért a számlálónak minden 65 535-62 500=3035 érékről kell indulnia.

| T_PERIOD: | T/2 [SEC] | FREQ. |
|-----------------------|-----------|-----------|
| DW 0xFFFF-(.500000/8) | :.50 | - 1.00 Hz |
| DW 0xFFFF-(.400000/8) | :.40 | - 1.25 Hz |
| DW 0xFFFF-(.300000/8) | :.30 | - 1.66 Hz |
| DW 0xFFFF-(.200000/8) | :.20 | - 2.50 Hz |
| DW 0xFFFF-(.100000/8) | :.10 | - 5.00 Hz |
| DW 0xFFFF-(.080000/8) | :.08 | - 6.25 Hz |
| DW 0xFFFF-(.50000/8) | :.05 | - 10 Hz |
| DW 0xFFFF-(.30000/8) | :.03 | - 16.6 Hz |
| DW .0000 | | ;VEGJELZO |

A táblázatban szerepelnek a rövid gombnyomásokra változó frekvenciák.

8.5.15. kodzar_14.asm

Egy gombbal működő kódzárát szimulálunk. A kódolás adott rövid billentyűnyomás-sorozatokat elhatároló hosszú billentyűnyomásokkal történik. A kódszó tetszőleges hosszúságú lehet. A kódszóban lévő kódok (rövid nyomások száma) helyességét az M_TST_KOD makró felhasználásával ellenörizzük.

```

M_TST_KOD MACRO C_KOD, R_COUNT, L_UJRA
LOCAL L_UJFI, L_RES, L_DEC, L_KILEP
MOVlw C_KOD
MOVwf R_COUNT ;HÁNYSZOR KELL RÖVIDEN NYOMNI
L_UJFI CALL S_VAR_LONG ;GOMBFIGYELÉS
BTFS C WREG, 1
BRA L_RES ;HA 2 (HOSSZÚ)
BTFS S WREG, 0
BRA L_UJFI ;HA 0 - VÁRAKOZÁS
BRA L_DEC ;HA 1 - CSÖKKENTÉS
L_RES MOVf R_COUNT, F
BZ L_KILEP ;HA 0, AKkor ADOTT KÓDSZÁM OK.
BNZ L_UJRA ;NEM JÓ A KÓDSZÁM
L_DEC DECF R_COUNT ;RÖVID GOMBNYOMÁSRA CSÖKKENT
BRA L_UJFI
L_KILEP ENDM
-----
L_MAIN:
L_INI_PORTS
CLRf TRISD ; LATD KIM.
CLRf LATD

```

```

BCF    INTCON2, RBPU
SETF   PORTB
SETF   TRISE
L_ISMET
M_TST_KOD C_KOD1, R_COUNT, L_ISMET
M_TST_KOD C_KOD2, R_COUNT, L_ISMET
M_TST_KOD C_KOD3, R_COUNT, L_ISMET
L_NYITVA CALL S_DLY
STG PORTD, 4
BRA L_NYITVA

```

8.5.16. adkod_15.asm

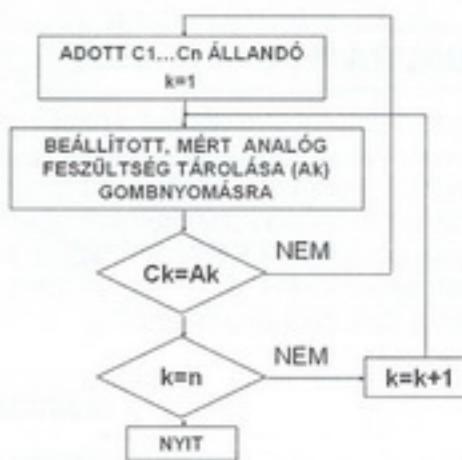
Ebben a feladatban is egy kódzár szerepel, de a kulcs ez esetben egy potenciométer és egy nyomógomb. Sorban egymás után beállítjuk a potenciométerrel az analóg feszültséget mint kódot, és megnyomjuk a gombot. Ekkor összehasonlítjuk ezt az értéket a bevitt, tárolt kód-dal. Ha bizonyos határértéken belül egyenlő, akkor azonosnak fogadjuk el, és továbblépünk a következő kód vizsgálatára, különben visszalépünk a vizsgáló rész legelejére.

Az AD-re kapcsolódó potenciométerrel sorban egymás után feszültségszinteket állítunk be, amiket egyenként, a nyomógomb megnyomásával érvényesítünk. Az így kapott feszültségeket hasonlítjuk össze a letárolt megfelelőjükkel, és ha egyeznek, akkor a következő feszültsséget vizsgáljuk meg.

```

L_UJPROBA
M_KODVI 0X0F
M_KODVI 0X00
M_KODVI 0X0C
M_KODVI 0X03
CLRF DSPORT
MOVlw .10
CALL S_VILLOG
BRA $

```



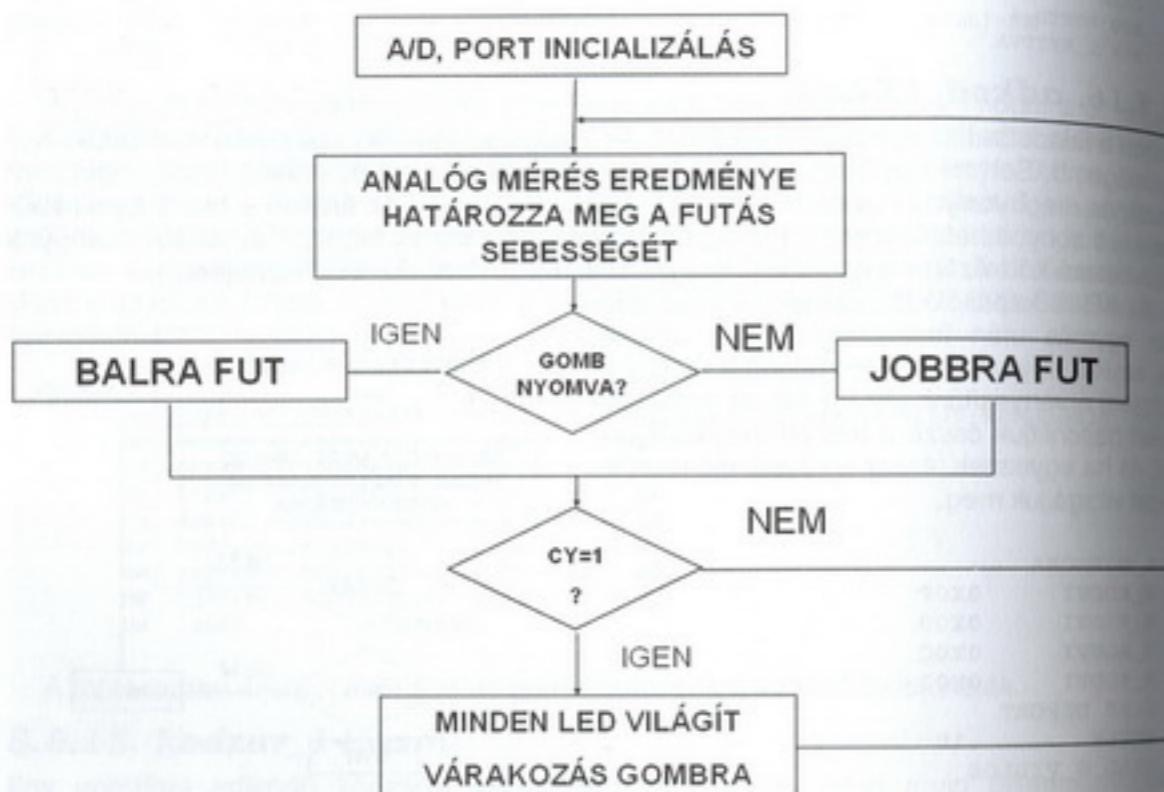
8.12. ábra
Analóg feszültségmérésen alapuló kódzár

| M_KODVI | MACRO | C_KOD | LOCAL | L_UJRA |
|---------|-------|-------|-------|----------------|
| L_UJRA | | | CALL | S_MER_ADO |
| | | | MOVFF | ADRESH, DSPORT |
| | | | CALL | S_VAR_LONG |
| | | | MOVF | WREG, F |
| | | | BZ | L_UJRA |
| | | | MOVFF | ADRESH, WREG |
| | | | ANDLW | 0XF0 |
| | | | SWAPP | WREG |
| | | | XORLW | C_KOD |
| | | | BNZ | L_UJPROBA |
| | | | ENDM | |

Hogy könnyű legyen a beállítás, a LED-soron megjelenő, digitálisan kijelzett feszültség első 4 bitjét figyeljük.

8.5.17. jatek_16.asm

Utolsó feladatként egy kis játékprogramot készítünk. A feladat egy oda-vissza futó fény két szélső LED mint végpontok között tartása. A gombnyomás hatására a futófény iránya az ellentettjére vált. A feladat változó nehézségű lehet, mert a potenciométer állításával a futófény sebességét tudjuk változtatni bizonyos határok között. A 8.13. ábra mutatja a program működési blokkdiagramját.



8.13. ábra
Játék blokkvázlata

9. PIC PROGRAMOZÁS BASIC-BEN

A programozói körökből induló tévhittel szemben BASIC-ben is nagyon hatékonyan és gyorsan lehet programozni. Mivel az első BASIC-megvalósítások a könnyebb hibakeresés érdekében a megírt programot soronként elemezve és értelmezve (interpretálva) hajtották végre, ezért viszonylag lassúak és korlátozott képességük voltak. A korszerű BASIC nyelvek már fordítást használnak a gépi kód előállítására, ezért gyorsak és hatékonyak. PIC-ek esetén ez a **microEngineering Labs, Inc.** cég **PicBasic**, illetve **PicBasic Pro** fordítóprogramja. Az előbbi BASIC-Bélyeg I.-gyel, míg az utóbbi BASIC-Bélyeg II.-vel kompatibilis.

A **BASIC-Bélyeg** a Parallax cég fejlesztése, amely 16 ki- és bemeneti kivezetéssel elérhető komplett számítógép egy kis, bélyegnagyságú NYÁK-lemezen kialakítva. A BASIC-Bélyeg lényegében egy BASIC értelmezőprogramot ROM-ban tartalmazó mikrokontroller és hozzá sorosan kapcsolódó EEPROM memória. Ez a memória a futtatandó BASIC program sűrűbben kódolt (tokenizált) alakját tartalmazza, amelyet a mikrokontroller utasításoknál értelmezve hajt végre. A programfejlesztéshez egy IBM PC-kompatibilis gépre és a rajta futó fejlesztőprogramra van szükség.

A cégi honlapján (<http://www.microengineeringlabs.com/>) lehet további információkat szerezni a PicBasic fordítóról. A továbbiakban a fejlettebb PicBasic fordítóval foglakozunk.

A fordítót az MPLAB fejlesztői környezetben is lehet használni. A telepített MPLAB alá kell integrálni. Első lépésként telepíteni kell a PicBasic Pro fordítót, cél szerűen a C:\PBP könyvtárba. Ezután a pluginokat tartalmazó tömörített fájlt a C:\PBP könyvtárba kicsomagoljuk, és a PBregister.bat parancsfájlt elindítjuk (Start → Run → C:\PBP\PBregister.bat). Ezek után a BASIC fordító az MPLAB részévé vált, és a Tools menüpont alatt kiválasztható és használható.

9.1. PICBASIC™ PROGRAMOZÁS

Röviden összefoglaljuk a programozással kapcsolatos ismereteket. A BASIC programban számos olyan elemet használunk, amit az assembler programozásnál már megismertünk.

• Megjegyzések

A program olvashatóságát, megértését segítik a megjegyzések. A PicBasic Compiler (a továbbiakban PBC) megjegyzések vagy a REM kulcsszóval, vagy egy szimpla idézőjellel ('') kezdődnek, és a sor végéig tartanak.

• Számkonstansok és karakterek

Három típus lehetséges: decimális, bináris vagy hexadecimális. Bináris szám esetén a 0-kból és 1-esekből álló sor '%' -kal kezdődik. Hexadecimális értékeknél a kezdő karakter: '\$'. A decimális alak az alapértelmezett, nincs külön jelölve. A karaktereket dupla idézőjelek közé kell zárni és az ASCII kódjukra lesznek átalakítva.

```

100      ' DECIMÁLIS 100
%100     ' BINÁRIS ÉRTÉK, DECIMÁLISAN 4
$100     ' HEXADECIMÁLIS ÉRTÉK, DECIMÁLIS 256
'A'       ' ASCII ÉRTÉKE DECIMÁLIS 65
'D'       ' ASCII ÉRTÉKE DECIMÁLIS 100
  
```

• Szövegfüzérek (sztringek)

Bár a PBC-nek nincsenek sztringkezelő függvényei, de néhány parancsban használhatók. Ezek egy vagy több karakterből állnak, és dupla idézőjelek közé vannak zárva.

| | |
|---------|--------|
| "HELLO" | STRING |
|---------|--------|

- Azonosítók**

Az azonosító egyszerűen egy név. A PBC-ben sorok címkézésére és szimbólumok elnevezésére használatos. Betük, számok, aláhúzások sorozata, de számmal nem kezdődhet. Nem kisbetű-nagybetű érzékeny ezért a LABEL és Label egyformán van kezelve. Bár tetszőleges hosszúságú lehet, PBC csak az első 32-t ismeri fel.

- Címek**

Utasítások helyének megjelölésére használt, hogy hivatkozni lehessen rá a GOTO vagy a GOSUB parancsoknál. A PBC nem sorszámozott, hanem címkézett sorokat használ, és nem kell minden sort címkével ellátni. A címke a sor elején egy kettősponttal végződő azonosító.

HERE: SEROUT 0,N2400, ("HELLO, WORLD!",13,10)

GOTO HERE

- Változók**

Számos előre definiált változót használhatunk a programban. A bájtos változók neve: B0, B1, B2,... Szó (16 bites) változók: W0, W1, W2,.. Ez utóbbiak lényegében kétbájtos változóból állnak. Például W0 B0-ból és B1-ből áll, W1 együtt B2 és B3 stb. Ezeket a SYMBOL parancssal tudjuk átnevezni, „beszélő” névvé. Ezeket a változókat a PICmicro adatmemoriája tárolja, B0 az első felhasználó által használható RAM címen van.

A változók száma 22 (B0–B21), az e fölöttieket B22-től a fordító könyvtárrutinjai használják. Ezek a hozzárendelések PBH.INC fájlban találhatók az INC alkönyvtárban. Azoknál a PIC-eknél, amelyeknek csak 36 bájt RAM-juk van a 22 felhasználó változó és a könyvtárrutinok változói a teljes RAM-területet lefoglalják. Nagyobb RAM-területtel rendelkező tokoknál további felhasználói RAM-terület áll rendelkezésre. A fordító ez nem ellenőrzi, vagyis ha olyan RAM-változóra hivatkozunk, amik mögött nincs fizikai regiszter, akkor a fordító nem generál hibaüzenetet, de a programunk nem azt csinálja majd, amit szeretnénk.

Az első kétbájtos változó, B0 és B1 tartalma bites változóként használható: Bit0, Bit1, ..., Bit15.

A Port, Dirs and Pins változónevek előre definiáltak. Pins a PORTB kivezetéseit jelöli, Dirs pedig PORTB irányregisztere (TRISB). Dir nulla értéke a lábat bemenetnek, egy pedig kimenetnek állítja. A legtöbb utasítás Pins bitjeit automatikusan állítja. A Port változó 16 bites és Pins és Dirs bitjeit tartalmazza, Port bitjei: Pin0, Pin1...Pin7, Dir0, Dir1...Dir7.

Bekapcsoláskor vagy RESET után Dirs értéke \$00, azaz minden bemenet és minden változó értéke is nulla lesz. minden változó előjel nélküli.

Az előre definiált változók összefoglalva:

| 16 bites (szó) | W0 | | W2 | | ... | Port | |
|----------------|-------------------------------|--------------------------------|----|----|-----|-------------------------------|-------------------------------|
| Bájt | B0 | B1 | B2 | B3 | ... | Pins | Dirs |
| Bit | Bit0, Bit1, ... Bit7 | Bit8, Bit9, ... Bit15 | | | | Pin0, Pin1, ... Pin7 | Dir0, Dir1, ... Dir7 |

- Szimbólumok**

Olvashatóbbá, érhetőbbé teszik a programot. PBC teszi lehetővé, hogy állandókat, változókat vagy más szimbólumokat elnevezzünk. Mindig csak egy szimbólum definált ható a SYMBOL utasítással.

SYMBOL TEN = 10

KONSTANS JELÖLÉSE

SYMBOL COUNT = W3

ELNEVEZETT SZÓ VÁLTOZÓ

SYMBOL BITVAR = BIT0

ELNEVEZETTT BIT VÁLTOZÓ

SYMBOL ALIAS = BITVAR

BITVAR MÁSIK NVE ALIAS

- Több utasítást tartalmazó sorok**

Tömörebb lesz a programunk megjelenése. Kettősponttal választhatjuk el az utasításokat:

W2 = W0 : W0 = W1 : W1 = W2

Az egy sorban leírt utasítások folytatódhatnak a következő sorban, ha a sor utolsó karaktere egy aláhúzás karakter.

AZ INCLUDE direktívát is használhatjuk, megadhatunk programmodulokat, amiket be

kívánunk szűrni egy adott helyre.

INCLUDE "MODEDEFS.BAS"

A PIC assemblerben szereplő DEFINE direktíva itt is megjelenik: segítségével egy karakterSOROZATHOZ egy másik karakterSOROZATOT rendelhetünk:

DEFINE ADC_BITS 8 AZ A/D ÁTALAKÍTÁS BITJEINEK A SZÁMA

9.2. AJÁNLOTT PROGRAMOZÁSI STÍLUS

Célszerű a programozó által kialakított stílust következetesen alkalmazni. Sok megjegyzést (comment) használunk. Ez azért fontos, mert a program megírása után még emlékszünk, hogy mit miért csináltunk, de ez nem is hosszú idő múlva feledésbe merül. Ezért fontos az olyan megjegyzések használata, amik nem azt írják le, hogy mit hajtott végre a program, hanem azt, hogy miért csinálta ezt. Például a „Pin0 1-be állítása” megjegyzés csupán a program szintaxisát írja le, de azt nem, hogy miért is történt ez.

Célszerű egy kódrészlet előtt hosszabb, többsoros megjegyzésben röviden leírni, hogy mi a kódrészlet feladata. Ez nagyon informatív, de nem helyettesíti a soronkénti megjegyzéseket. Érdemes a program verziószámát, módosítási dátumokat és a hardverkörnyezetet is leírni.

A PICBASIC Pro a szokásos BASIC utasításokon túl számos perifériakezelő utasítást is tartalmaz. A következő táblázatban ezeket foglaltuk össze.

| | |
|-------------|---|
| @ | Egy assembly nyelvű utasítássor beillesztése a programba. |
| ASM..ENDASM | Egy assembly nyelvű kódrészlet beillesztése a programba. |
| BRANCH | Számított ugrás. Azonos az ON..GOTO) utasítással. |
| BRANCHL | BRANCH a lapon kívülre (hosszú BRANCH). |
| BUTTON | Prellengetés és ismételt automatikus állapotbeolvasás. |
| CALL | Assembly szubrutin hívása. |
| CLEAR | Minden változót nulláz. |
| COUNT | Megszámolja az impulzusok számát egy lábon. |
| DATA | Kezdeti érték íródik a tokban lévő EEPROM-ba. |
| DEBUG | Aszinkron soros kimenet fix lábon és sebességgel. |
| DISABLE | Tiltja ON INTERRUPT feldolgozást. |
| DTMFOUT | Telefon tone frekvenciák jelennek meg egy lábon. |

| | |
|-----------------------|--|
| EEPROM | A kezdeti értéket definiálja a tokban lévő EEPROM-ba. |
| ENABLE | Engedélyezi az ON INTERRUPT feldolgozást. |
| END | Programvégrehajtás befejezése és alacsony fogyasztási módba állás. |
| FOR..NEXT | FOR NEXT ciklus megadása. |
| FREQOUT | Max. két frekvenciát ad ki a lábon. |
| GOSUB | BASIC szubrutinhívás adott címkére. |
| GOTO | A programvégrehajtás a megadott címkénél folytatódik. |
| HIGH | A kimenet magas szintű lesz. |
| HSERIN | Hardveres aszinkron soros bemenet. |
| HSEROUT | Hardveres aszinkron soros kimenet. |
| I2CREAD | Bájtok olvasása az I2C buszról. |
| I2CWRITE | Bájtok írása az I2C buszra. |
| IF..THEN..ELSE..ENDIF | Feltételes utasításvégrehajtás. |
| INPUT | A láb bemenet lesz. |
| {LET} | Egy kifejezés eredményét egy változóhoz rendeli. |
| ON INTERRUPT | Megszakításkor egy BASIC szubrutint hajt végre. |
| LCDOUT | Karaktereket jelenít meg az LCD-n. |
| LOOKDOWN | Konstans táblázatban érték keresése. |
| LOOKDOWN2 | Konstans/változó táblázatban érték keresése. |
| LOOKUP | Táblázatból egy konstans értéket vesz ki. |
| LOOKUP2 | Táblázatból egy konstans/változó értéket vesz ki. |
| LOW | A láb kimenet lesz és alacsony szintű. |
| NAP | Rövid időre a processzort kikapcsolja. |
| OUTPUT | Az adott láb kimenet lesz. |
| PAUSE | Késleltetés (1ms felbontás). |
| PAUSEUS | Késleltetés (1μs felbontás). |
| PEEK | Regiszterből bájtot olvas ki. |
| POKE | Bájtot ír egy regiszterbe. |
| POT | A lábra kötött potenciometér analóg feszültségét olvassa ki. |
| PULSIN | Egy lábon megjelenő impulzus szélességét méri. |
| PULSOUT | A lábon impulzust hoz létre. |
| PWM | A lábon PWM jelet bocsát ki. |
| RANDOM | Véletlen számot generál. |
| RCTIME | Egy lábon megjelenő impulzus szélességét méri. |
| READ | A tokban lévő EEPROM-ból egy bájtot kiolvas. |
| RESUME | A megszakítás-kiszolgálás után folytatja a programvégrehajtást. |
| RETURN | Folytatja a programvégrehajtást az utoljára végrehajtott GOSUB után. |

| | |
|-------------|--|
| REVERSE | Felcseréli a láb be/kimeneti irányát. |
| SERIN | Soros aszinkron bemenet (BS1 szerü). |
| SERIN2 | Soros aszinkron bemenet (BS2 szerü). |
| SEROUT | Soros aszinkron kimenet (BS2 szerü). |
| SEROUT2 | Aszinkron soros kimenet (BS2 szerü). |
| SHIFTIN | Szinkron soros bemenet (SPI). |
| SHIFTOUT | Szinkron soros kimenet (SPI). |
| SLEEP | Egy időre kikapcsolja a processzort. |
| SOUND | Hangot vagy fehérzajt ad ki a megadott lábon. |
| STOP | Programvégrehajtás leállítása. |
| SWAP | Két változó tartalmának a megcserélése. |
| TOGGLE | A láb kimenet lesz és szintet vált. |
| WHILE..WEND | Amíg a feltétel igaz, addig végrehajtja a ciklusmagot. |
| WRITE | A tokban lévő EEPROM-ba ír egy bájtot. |
| XIN | X-10 bemenet |
| XOUT | X-10 kimenet |

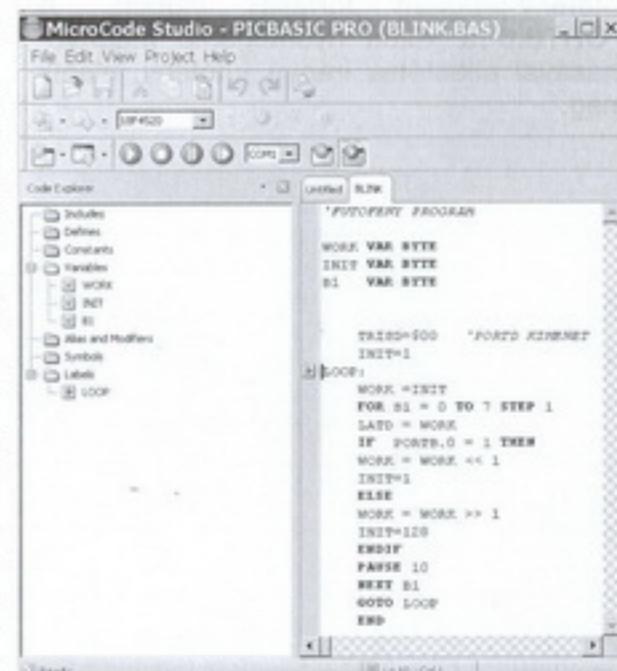
9.3. MEGOLDÁSOK BASIC PROGRAMOZÁSHOZ

A BASIC-ben történő PIC-programozáshoz a cégek két megoldást kínál.

Egyik megoldásként használhatjuk a MicroCode Studio elnevezésű, grafikus kialakítású integrált fejlesztőkörnyezetet a beépített hibakeresőjével, amit kimondottan a microEngineering Labs cégek PICBasic PRO fordítójához fejlesztettek ki.

Egyedi megoldás a „kódelfeledező” (code explorer). Lehetővé teszi, hogy automatikusan az include fájlokra, definíciókra, állandókra, változókra elnevezésekre, módszerekre, szimbólumokra, címkekre lépjünk, azokra, amelyek a programunkban szerepelnek.

Az In-Circuit Debugger (ICD) hibakeresővel lehet a PicBasic programot a PIC mikrovezérlőben futtatni, a változókat, SFR-, memória EEPROM tartalmakat megtekinteni a végrehajtáskor. Át lehet kapcsolni több megadott töréspont között is.



9.1. ábra
MicroCode Studio

A másik megoldás a BASIC fordító használata MPLAB alatt. A PICBasic PRO installálása után az MPLAB IDE *Project → Set Language Tool Locations* menüjében kiválaszthatjuk a *microEngineering PicBasic Pro Toolsuite* eszközt. Rákattintva meg kell adni a Basic fordító helyét, amit az alapértelmezett helyre hivatkozva automatikusan felkínál. Ezek után a szokásos módon létrehozhatunk egy projektet, amibe betölthetjük a .BAS kiterjesztésű állományunkat, és amit lefordíthatunk és futtathatunk. A fordítás eredménye számos könyvtári hivatkozást tartalmaz, egyszerűen nem olvasható .ASM kiterjesztésű program, amit az MPASM alakít át betölthető .HEX állományá.

A programunk fordításához szükség van egy állományra, amiben a felhasznált és a fordításkor (*Project → Compile*) beillesztett szubrutinokat és makrókat tartalmazza. PIC18-as család esetén ennek neve **PBPPIC18LIB.BAS**. A 9.2. ábra az MPLAB környezetben megjelenő BASIC-ben írt futófény-program forrását mutatja. A PORTB.0-ra kötött nyomógomb megnyomásával ellentétes irányú futófény jelenik meg.

```

basicproba - MPLAB IDE v8.14
File Edit View Project Debugger Programmer Tools Options Help
basicproba.mprj basicproba.bas
MPLAB IDE Editor
PEPPIC18LIB FUNKBAS
'FUTÓFÉNY PROGRAM
WORK VAR BYTE
INIT VAR BYTE
BL VAR BYTE

TRISD=FOO 'PORTB.0 KIENET
INIT=1
LOOP:
WORK = INIT
FOR BL = 0 TO 1 STEP 1
LATD = WORK
IF PORTB.0 = 1 THEN
WORK = WORK + 1
INIT=1
ELSE
WORK = WORK - 1
INIT=128
ENDIF
PAUSE 10
NEXT BL
GOTO LOOP
END

```

9.2. ábra
Futófény BASIC-ben

II. RÉSZ

PIC PROGRAMOZÁS C NYELVEN

KOPJÁK JÓZSEF

BEVEZETÉS

Manapság a beágyazott rendszerek programozóinak egyre nagyobb hányada tér át C nyelvű programfejlesztésre. Egy új programnyelven történő programfejlesztés általában sok örömmel, de bosszúságokkal is jár. Azért, hogy az öröm kerüljön főlénnyé a bosszúságokkal szemben, a könyv hátralévő fejezetei megpróbálnak segítséget nyújtani a C nyelv megismerésében, mind a programozással most ismerkedők és az assembly nyelvet már jól ismerők számára is. Reméljük, hogy már gyakorlott C programozók is találnak benne néha olyan információt, amit eddig nem használtak.

A könyv tervezésekor sokat gondolkadtunk azon, hogy a C nyelv alapjait melyik architektúrához tartozó fordítón keresztül mutassuk be. Várunk csak... C nyelvből nem csak egy van? ... A válasz nem olyan egyszerű. Az igaz, hogy C nyelvből, értsd alatta az ANSI C nyelvet, csak egy van, de az egyes hardverimplementációk, a hardver speciális tulajdonságai következtében, eltérnek az eredeti szabványtól.

Az eredeti C nyelvet 1972-ben a Dennis Ritchie vezette csapat fejlesztette ki a Bell Telephone Laboratoriesben, a Unix operációs rendszerekhez. Ezután a nyelv többször is módosult. Az első C szabványt az Amerikai Nemzeti Szabványügyi Hivatal adta ki 1989-ben ANSI X3.159–1989 szabványszám alatt. Az ezt a szabványt teljesítő fordítókat hívjuk ANSI C kompatibilis fordítóknak. Az eredeti amerikai szabványt egy évvel később a Nemzetközi Szabványügyi Szervezet is átvette, apróbb módosításokkal, ISO/IEC 9899:1990 néven. A nyelv eddigi utolsó szabványosítása 1999-ben történt ISO/IEC 9899:1999 (más néven C99) szabványszám alatt. Ezt a szabványt már nem mindegyik C fordító támogatja, de például a nyílt forráskódú GNU C fordító igen.

Végül, hosszú tanakodás után, úgy döntöttünk, hogy a C nyelv ismertetéséhez a Microchip C30 fordítóját hívjuk segítségül. A fordító a GNU C általános C fordító a 16 bites PIC mikrokontroller-család hardverspecifikus változata. A könyv végén architekturális szempontból le- és felfelé egyaránt kitekintünk. A következő fejezetekben a könyv segítséget ad az áttéréshez a PIC mikrokontrollerek 8 bites és 32 bites architektúráira is.

A C nyelv bemutatásánál az eredeti ANSI C szabvány szerinti nyelv kerül bemutatásra. A C99 kiegészítések általában megtalálhatók az egyes fordítók kézikönyveiben, így a Microchip C30 fordító kézikönyvének 2. fejezetében is (*Eltérések a 16 bites C fordító és az ANSI C között*).

A fejezeteket úgy próbáltuk összeállítani, hogy a kezdők és a már jártasabb programozók is találjanak maguknak érdekes részeket. A fejezetek elején a kezdőknek szóló részek találhatók. Az egyes fejezetek vége az adott téma kör bonyolultabb részeivel foglalkozik. Abban az esetben, ha úgy érzi, hogy az adott téma kör már nem köti le önt, akkor bátran ugorja át, majd később, ha szüksége lesz rá, akkor térjen vissza hozzá.

Ahogy az már az előbbi bekezdésekben kiderült, a C nyelvet a hetvenes években fejlesztették ki a UNIX operációs rendszerhez. A nyelv mai napig tartó sikerének alapja talán azt, hogy hardverfüggetlen és egyben hardverközeli nyelv is. Ez a kettőség eredményezi formok alá.

A C programozó elől nem kerül elrejtésre a hardver; ha szüksége van rá, akkor bármi-lyen mélységen el tudja érni azt. Természetesen a túl hardverspecifikus programozás fontja a kódunk platformfüggetlenségét, de segít gyors futású kódok írásában. A nyelv másik nagy előnye, hogy az egyes utasítások áttekinthető gépi kódot adnak, ami segíthet az esetleges alacsony szintű hibakeresésben is.

Abban az esetben, ha a programírás során nem használjuk ki a nyelv implementációs függő kiterjesztései, akkor kényelmesen készíthetünk platformfüggetlen kódokat, elősegítve ezzel az újabb technológiákra történő gyors áttérést.

A beágyazott rendszerek bonyolultsági szintje manapság már megköveteli a fejlesztőtől, hogy ne assembly nyelven programozzon. Az assembly nyelvű programozás hátránya, hogy nagy programknál átláthatatlanná válik, ami rontja a programjaink megbízhatóságát. Ezenkívül a gépi kódban készült programok nem vihetők át másik hardverre, így a fejlesztő nem tudja a gyártó cégek által készített új technológiákat alkalmazni.

A beágyazott rendszerek programjai elé támasztott legnagyobb követelmény a megbízhatóság, ezért célszerű olyan nyelvet választani, amelynek segítségével jól átlátható, robusztus programokat készíthetünk. A nyelv megbízhatóságát talán az támasztja alá legjobban, hogy az operációs rendszerek a mai napig nagy részben C nyelven íródnak.

C nyelvű programjaink készítésekor mindenleginként tudatában annak, hogy nagyon éles fegyverrel dolgozunk. A C nyelv, a nyelvet jól ismerők számára, igazi kezes bárány, bármit meg tudnak benne valósítani. A nyelv szabadsága a kezdő programozót saját csapdáiba is bevezetheti, ezért kerüljük az átláthatatlan, bonyolult kifejezések használatát. Mivel a nyelv nagyon rugalmas, könnyen lustává teszi a programozót, aki így egyre átláthatatlanabb kódokat kezd készíteni. Hogy ne essünk a saját csapdánkba, minden jogi átlátható, strukturált kódot készítsünk, és ne sajnáljuk az időt a dokumentációra sem!

10. AZ ELSŐ PROGRAM

Az első C nyelvű könyv megjelenése óta, melyet Brian Kernighan és Dennis Ritchie készített, egy valamit is magára adó C könyv nem kezdődhet „Hello World!” mintaprogram nélkül. Ezt a hagyományt én sem szeretném megszakítani, de azért egy kicsit módosítva fogjuk ezt a programot megírni.

Mielőtt belemélyednénk az első programunk elkészítésébe, ellenőrizzük le, hogy minden szükséges szoftvereszközzel rendelkezünk-e a programfejlesztéshez. A fejezetben tárgyalt program fejlesztéséhez, fordításához és teszteléséhez a következőre lesz szükségünk:

- MPLAB IDE ingyenes fejlesztőkörnyezet (legutolsó változata elérhető az alábbi webcímen: <http://www.microchip.com/mplab>);
- MPLAB SIM szimulátor, mely az MPLAB IDE része;
- Az MPLAB C30 PIC24 kontrollerekhez való C fordító tanuló-, ingyenes változata (legújabb változata elérhető a <http://www.microchip.com/c30> webcímén).

Az előbb említett programoknak a könyv írásakor aktuális változatai megtalálhatóak a könyvhöz tartozó DVD-mellékleten is, de ajánlom a legfrissebb változatok letöltését a fent említett webcímeken keresztül.

10.1. A FORDÍTÓ KÖNYVTÁRSZERKEZETE

Az Microchip MPLAB C30 fordítójának feltelepítése után a következő könyvtárak jönnek létre a számítógépünkön. A fordító alapértelmezett esetben a c:\Program Files\Microchip\MPLAB C30\ könyvtárba települ fel. A fordító alkönyvtárai a következők:

- A bin alkönyvtárban a fordító futtatható állományai találhatók.
- A docs alkönyvtárban a fordítóhoz tartozó dokumentáció található.
- Az include alkönyvtárban a standard C fejlécállományok találhatók, h kiterjesztéssel.
- A lib alkönyvtárban a standard C és a mikrokontroller-specifikus előfordított könyvtárak találhatók, a kiterjesztéssel.
- Az src alkönyvtárban a standard C és a mikrokontroller-specifikus előfordított könyvtárak forrásállományai találhatók.
- A support alkönyvtár alkönyvtáraiban a mikrokontroller-specifikus fejlécállományok és a linker állományok találhatók, lkr kiterjesztéssel.

10.2. AZ ELSŐ PROJEKT LÉTREHOZÁSA

A sikeres telepítés után hozzuk létre az első C nyelvű projektünket, amelyet a következő lépések segítségével tehetünk meg:

- 1) Indítsuk el az MPLAB IDE-t, majd hozzunk létre egy új projektet a varázsló segítségével, amelyet a Project → Project Wizard... menüpont alatt találunk.
- 2) Az Üdvözlőszöveget tartalmazó oldal után az eszköz (mikrokontroller) típusának kiválasztása következik. Itt, a legörökölő menüben, a PIC24FJ128GA010 típust válasszuk ki. A most kiválasztott mikrokontroller a PIC 16 bites mikrokontroller-család egyik alaptípusa.
- 3) Következő oldalon a fordító eszközt kell kiválasztani, itt a Microchip C30 Toolsuite-ot válasszuk.
- 4) Következő lépés a projektállomány létrehozása. Hozzunk létre egy Elso névű könyvtárat, majd abban egy projektet hello névvel.

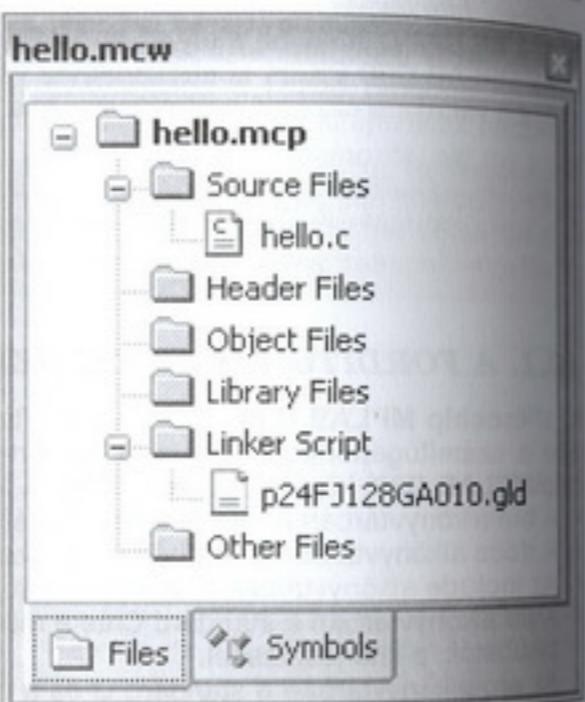
5) Ezek után a projekthez hozzá kell adni a **p24FJ128GA010.gld** szerkesztő (linker) állományt. Abban az esetben, ha a C30-as fordító az alapértelmezett helyre lett feltelepítve, akkor az előbb említett állomány a **c:\Program Files\Microchip\MPLAB-C30\support\PIC24F\gld** könyvtárban található. A szerkesztőállomány az adott IC memóriakiosztásáról tartalmaz adatokat. Ez az állomány mondja meg a szerkesztőnek, hogy hol van a reset vektor, az egyes megszakítások kezdőcímei és a speciális funkcióregiszterek memóriacímei.

Az új MPLAB-változatoknál már elhagyható az utolsó lépés, mivel a fejlesztői környezet automatikusan hozzárendeli a projekthez a szükséges szerkesztőállományt.

Ezzel az öt lépéssel létre is hoztunk egy új projektet. Következő lépésként nyissuk ki a projektablakunkat a **View → Project** menüpont segítségével, majd a **Project → Add New File to Project...** menüpontban adjunk hozzá egy új állományt a projekt-hez. Az új állományt, **hello.c** névvel, mentük el a projektet tartalmazó könyvtárba. Ha minden helyesen végeztünk el, akkor a projektablakunknak úgy kell kinéznie, mint ahogy azt a 10.1. ábra mutatja. Ha nem látszik a projektablak, engedélyezzük a **View → Project** menüpont segítségével.

Most, hogy elkészítettük a projektünket, nekiállhatunk az első programunk begépelésének.

10.1. ábra
Felkonfigurált projekt ablaka



10.3. „SZIA VILÁG!”

Nyissuk ki a **hello.c** állományt, és gépeljük be a következő mintapéldát. Begépeléskor figyeljünk arra, hogy a nyelv kis- és nagybetűérzékeny (*case-sensitive*). Ez az jelenti, hogy még például a **PORTA** kifejezés az A PORT egy regiszterére hivatkozik, addig **Porta** vagy a **porta** szavakat nem tudja a fordító értelmezni.

10.1. mintaprogram

```
/* Az első programom */

#include <p24fj128ga010.h>

main ( )           // Főprogram kezdete
{
    PORTA = 0xFF;   // PORTA-ba FF hexadecimális szám mozgatása
}
```

Nézzük, hogy mit jelentenek az egyes sorok. A legelső sorban egy megjegyzés található. A megjegyzés /*-gal kezdődik és a */-ig tart. A megjegyzések több soron át is tartthatnak, és a tartalmukat a fordító nem veszi figyelembe. Az utolsó előtti sor is tartalmaz egy megjegyzést. Abban a sorban a // (kettő darab törtvonal) jel után lévő szöveget a fordító szintén megjegyzésnek tekinti, egészen a sor végéig.

A C nyelvben az utasításokat két nagy csoportra lehet osztani. Az egyik csoportba az előfordítónak szóló utasítások tartoznak, míg a másik csoportba a fordítónak szóló utasítások tartoznak. A #-kal kezdődő utasítások az előfordítónak szólnak. Az #include <p24fj128ga010.h> utasítás hatására az előfordító a két relációs jel <> között lévő állomány tartalmát bemásolja a fordítás előtt a saját kódunkba. A **p24fj128ga010.h** állomány a mikrokontrollerre jellemző definíciókat tartalmaz. Itt találhatók az egyes regisztek és azok bitjeinek elnevezései. Aki gondolja, adja hozzá a saját projektjéhez a **Project → Add File to Project...** menüpont segítségével. A későbbiekben a perifériák pontos megnevezéseinek megkeresésében még segítségünkre lehet. Aki merész, az most is kinyithatja az állományt. Aki úgy érzi, hogy számára a benne lévő kód olyan, mintha ázsiai nyelven írt szöveget próbálna olvasni, azt meg szeretném nyugtatni, hogy pár fejezet mölvé nem is lesz annyira bonyolult az állomány tartalmának értelmezése. Fontos megjegyezni, hogy ne módosítsuk az állomány tartalmát, mert így csökkentjük annak az esélyét, hogy működő kódöt fordítson a fordító számunkra.

A C nyelv függvényszervezésű nyelv. A függvényszervezés azt jelenti, hogy az összes végrehajtó kód (olyan kód, amelyből valójában gépi kód keletkezik, azaz nem fordítónak szóló definíció) csak függvény belsejében lehet. Egyetlenegy kiemelt függvény létezik a nyelvben, a **main()** függvény. A **main()** függvény a programunk belépési pontja, így minden futtatható program írásakor implementálni kell. A függvények belső tartalmát hal-mazzárójelek (...) között tudjuk megadni. A hal-mazzárójelek közé foglalt utasításokat a nyelv egy logikai egységeknek tekinti, mint például egy ciklus magjának, elágazás egyik ágának vagy – mint a mostani esetben – egy függvény tartalmát határozza meg.

A C nyelven írt programoknak vannak íratlan formai szabályai. Az egyik ilyen szabály az, hogy az egy adott blokkon belüli egységet egy tabulátorral beljebb kezdjük. Ezt a módszert betartva a programunkban jól látható lesz, hogy melyik utasítás melyik utasításblokk része.

A **main()** függvény egy kifejezést tartalmaz: **PORTA = 0xFF;** amelynek jelentése az, hogy mozgassa a **PORTA** regiszterbe az **FF** hexadecimális számot. A C nyelv sorfüggetlen nyelv. Ez azt jelenti, hogy egy sorban több kifejezés is lehet, de egy kifejezés több sorban is elhelyezhető, ezért fontos, hogy a kifejezések végén megjelenik egy pon-tosszó, ami lezárja az adott kifejezést.

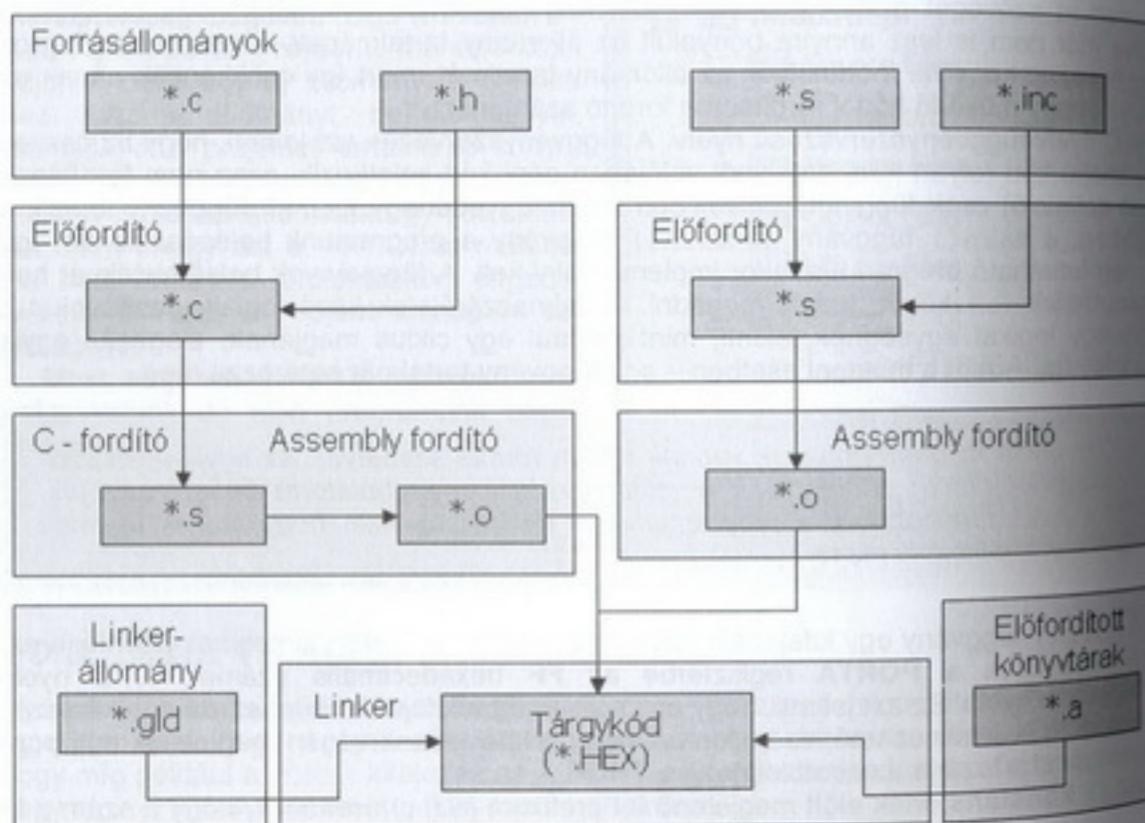
Az **FF** konstans érték előtt megjelenő **0x** prefixum jelzi a fordítónak, hogy a szám a tízenháros számrendszerben értelmezett szám. Ha egy szám előtt nincs semmilyen előtag, akkor tízes számrendszerben, ha **0b** található előtte, akkor binárisban, és ha csak egy nulla áll a szám előtt, akkor oktalis számrendszerben értelmezett.

10.3.1. Fordítás

Most, hogy készen vagyunk az első programunkkal, nekiállhatunk lefordítani. Mielőtt a gyakorlatban is elvégezzük a fordítást, nézzük meg elméletben, hogy mi történik. Egy projekt különböző nyelven írt állományokból állhat össze. Általában C és assembly nyelvű forrásállományokból állnak a programjaink. Az egyes forrásállományokat először az előfordító kapja meg, amely a makrók feldolgozásáért felelős. Az előfordító után a fordító folytatja tovább a munkát. Itt a fordító C kódóból assembly kódot készít. Ezek után az as-

sembly kódból már tárgykód készül. A tárgykódban a függvények és a globális változók hivatkozási nevekkel kerülnek eltárolásra. A fordítók nagy többsége estén a C kódból egyenesen tárgykód készül. A számunkra is olvasható assembly utasításlistát, sok esetben csak külön kérésre generálja a fordító. Legvégül a szerkesztő (linker) a különböző tárgykódokból végleges, futtatható kódot készít, a mikrokontroller-specifikus fordítói állomány (*.gld állomány) segítségével. A linker a gld állományban és/vagy a projektben meghatározott könyvtárállományokat (*.a) is betölteni, és szükség esetén a tárgykódhoz hozzáilleszti. A könyvtárállományok nem mások, mint előre lefordított programrészek. A könyvtárállományokból szerkeszti be a fordító a C kód futtatásához szükséges alapvető függvényeket (pl. indítókód, lebegőpontos aritmetika). A fordítás egyes lépései a 10.2. ábra szemlélteti.

A linker a futtatható kódon kívül (*.hex) még két állományt állít elő. A map kiterjesztésű szöveges állomány a programunk memóriatérképét tartalmazza. A cof kiterjesztésű állományt az MPLAB IDE fejlesztőfelület számára készül, és a hibakeresést segíti elő (például a lépésenkénti programfuttatáskor használja fejlesztőkörnyezet).



10.2. ábra
A fordítás menete

A fordítás végén a tárgykódunk három fő részből tevődik össze:

- Indítókód (Startup code):** Ha assembly nyelven írunk programot, akkor az összes inicializálást nekünk kell elvégeznünk. Abban az esetben, ha C nyelven programozunk, akkor a program futásához szükséges alapvető inicializálásokat az indítókód végezi el. Mielőtt a program meghívja a main() függvényünket, az előtt alaphelyzetbe állítja a stacket és az általunk használt változókat. Az indítókód forrását a crt0.s állomány tartalmazza. Az állomány a fordító src\pic30 alkönyvtárában található.
- Saját kódunk:** Ez a kód az általunk írt programból keletkezik.

3) Felhasznált előfordított függvények kódja: A fordítóhoz tartozik egy függvénykönyvtár, ami előre megírt függvényeket tartalmaz. Ezen függvények nagy többségét platformtól függetlenül minden fordítói csomag tartalmazza, ilyen például a printf() függvény is.

A Microchip C30 fordítói környezet tartalmaz egy alternatív indítókódot is, amelynek forrása a crt1.s állományban található. Az alternatív indítókódból hiányzik a változók inicializálása, így kevesebb programmemoriát igényel és gyorsabb lefutású mint az eredeti indítókód. Mivel az alternatív inicializáló kód nem végzi el a változó alaphelyzetbe állítását, ezért **használata veszélyes**, csak haladó programozóknak érdemes vele dolgozni. Az alternatív indítókód használatát a Project → Build Options... → Project menüpont alatt található konfigurációs ablak **MPLAB LINK30** fülén lévő **Symbols & Output** legörökítő menüt kiválasztva, a **Don't initialize data sections** jelölőnégyzet bepipálásával érhetjük el.

Most, hogy már elég sok minden tudunk a fordításról, nézzük meg a gyakorlatban is, hogy sikerül-e hiba nélkül lefordítanunk a nemrég megírt programunkat. Fordítást a Project → Build All menüpont segítségével indithatjuk el. Ebben az esetben az összes forrásállományuk újrafordul. Ha csak az utolsó fordítás után módosult állományokat szeretnénk lefordítani, akkor a Project → Make menüpontot célszerű választani. Ha sikerült hiba nélkül beírni az első kis programunkat, akkor fordítás után a **BUILD SUCCEEDED** feliratnak kell megjelennie az Output ablak **Build** füle alatt. Abban az esetben, ha piros színnel **BUILD FAILED** jelent meg az előbb említett ablakban, akkor hibát észlelt a fordító. A hiba megkeresésében is az Output ablak nyújt segítséget. Feljebb görgetve az ablakot, kék színnel találjuk meg a fordító által küldött figyelmeztetéseket és hibaüzeneteket is. Az üzenetben megjelenik a sor száma, ahol a fordító hibát talált, és a hiba rövid leírása. Ha az adott hibaüzenetre kattintunk az egérrel, akkor szövegszerkesztőben a hibaüzenet által mutatott sorra ugrik a kurzorunk.

Hibakeresés során fontos fontos értenünk a fordító lelkivilágát is. A nyelv sorfüggetlen, ezért a hiba nagyon sokszor nem ott van, ahol a fordító jelzi, hanem annak a környékén (általában felette). Például, ha a programunkból kitöröljük az egyetlen egy pontossávosszéket, akkor fordítás után a fordító nem abban a sorban fog hibát jelezni, ahol kitöröltük, hanem a következő sorban, hisz a fordító azt hitte, hogy még nincs vége az utasításnak, de a halmazzáró jelet már nem tudta az utasítás részeként értelmezni.

Sikeres fordítás után a következő kimeneti ablakkal kell találkoznunk:

```

Output
Build Version Control Find in Files
Debug build of project 'c:\PIC24\Elszo\hello.mcp' succeeded
Preprocessor symbol '_DEBUG' is defined.
Jul 11 22:34:37 2008
BUILD SUCCEEDED
  
```

10.3. ábra
Sikeres fordítás eredménye

10.3.2. Futtatás és szimuláció

Elérkeztünk az első programunk fejlesztésének utolsó pontjához, a program teszteléséhez. Első körben szimulátor segítségével fogjuk elvégezni a tesztelést. Válasszuk ki a Debugger → Select Tool → MPLAB SIM menüpontot. Ezzel a mozdulattal aktiválhatunk a fejlesztőkörnyezettel együtt feltelepített szimulátort. Következő lépés a PORTA regiszter figyelése. A View → Watch menüponttal aktiválhatjuk a változók és regiszterek értékének figyelésére alkalmas ablakot. A frissen kinyíló ablakban adjuk hozzá a PORTA regisztert, amit úgy lehetünk meg, hogy az ablak bal felső sarkában található legördülő menüből válasszuk ki az előbb említett regisztert, majd az Add SFR gombra rákattintva megjelenik a listában a PORTA regiszter értéke.

A programunkat lépésenként fogjuk futtatni. Első körben kattintsunk a programunk forráskódját tartalmazó ablakban az egér jobb gombjával a PORTA = 0xFF; utasítást tartalmazó sorra, majd a megjelenő menüből válasszuk ki a Run To Cursor parancsot. A parancs hatására a szimulátor az előbb említett sorig lefuttatja a programunkat, majd a sor végrehajtása előtt megáll. Ezt számunkra a fejlesztőkörnyezet a programsor előtt megjelenő zöld nyíl segítségével jelzi. A 10.4. ábra mutatja a futtatókörnyezetünk jelenlegi állapotát.

```
/* Az első programom */

#include <p24fj128ga010.h>

main ()
{
    PORTA = 0xFF; // PORTA-ba FF hexadecimális szám mozgatása
```

10.4. ábra
Lépésenkénti futtatás

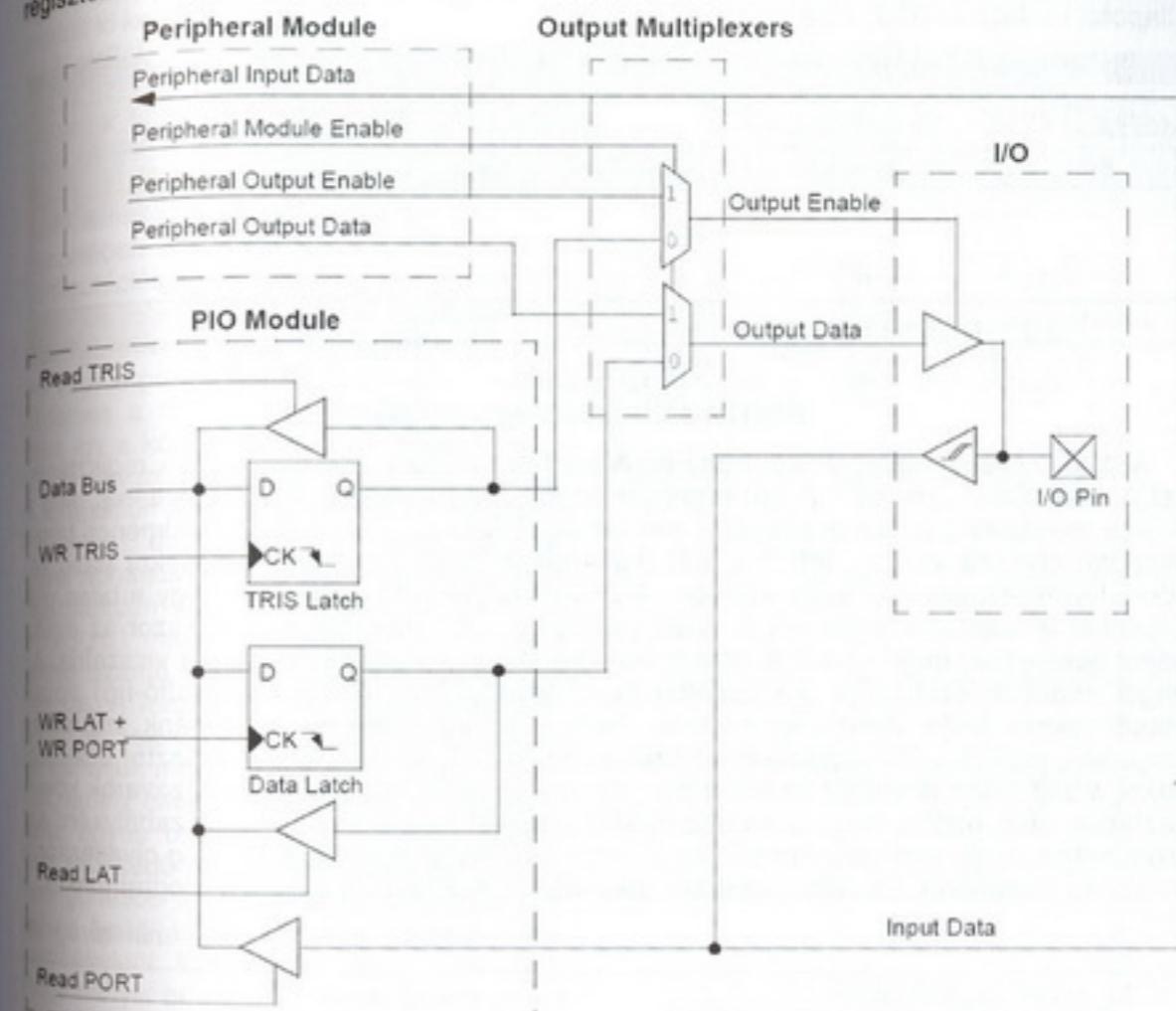
Ezek után hajtassuk végre a szimulátorral a következő utasítást. Ezt a Debugger → Step Over menüpont segítségével mondhatjuk meg a szimulátornak. Mit tapasztaltunk? Hát ez az! Semmit! A PORTA regiszter értéke nem változott meg, hisz nem állítottuk be a port irányát.

A lépésenkénti szimuláció során érdemes pár fontos menüponttal megismerned-nünk. Ezek a Debugger menüpont alatt találhatók. A Step Into egy utasításba lép be, a Step Over az utasítást végrehajtással átugorja, a Step Out a függvény hívójához tér vissza. A Debugger → Reset → Processor Reset menüpont segítségével a szimulált processzort indíthatjuk újra. A szimulációt elősegítő főbb menüpontok a fejlesztőkörnyezet eszközszorában is megtalálhatók.

10.4. PORTA BEÁLLÍTÁSA

A 10.5. ábra a PIC24FJ128GA010 mikrokontroller adatlapja 9. fejezetének (Be- és kimeneti portok) legelső ábrája, amely egy port általános lábkialakítását szemlélteti. A 16 bites mikrokontrollerek portjai 16 bitesek, de ez nem azt jelenti, hogy minden tizenhat láb kivétésre került. Az ábrán megfigyelhető, hogy minden lábhoz tartozik egy iránybit, amely az adott lábat be- vagy kimenetnek állítja. A mikrokontroller bekapsolását követően az

összes láb bemenetként működik, azaz a porthoz tartozó TRIS regiszterek bitjeinek értéke egy. Ha egy adott lábat kimenetként szeretnénk használni, akkor a porthoz tartozó TRIS regiszter adott bitjét nullába kell állítani.



10.5. ábra

A PIC24F kontrollerek általános PORT-kialakítása

Az új tudásunkat alkalmazva, módosítsuk az előző programunkat:

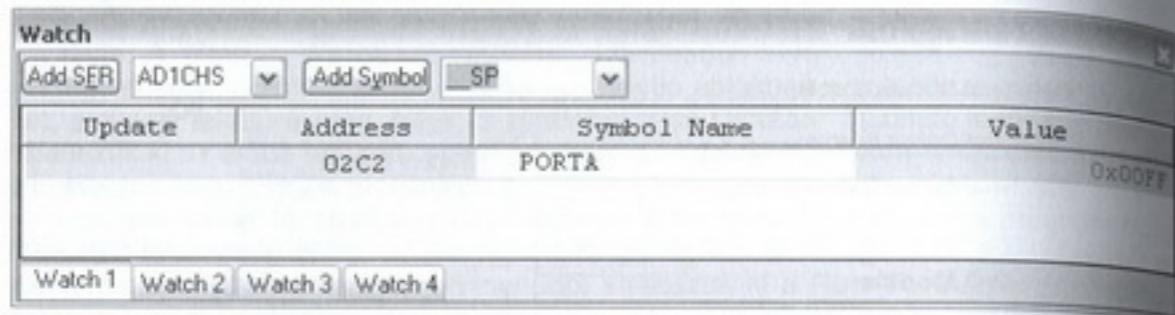
10.2. mintaprogram

```
/* Az első programom */

#include <p24fj128ga010.h>

main () // Főprogram kezdete
{
    TRISA = 0; // PORTA minden lába kimenet lesz.
    PORTA = 0xFF; // PORTA-ba FF hexadecimális szám mozgatása
```

Fordítsuk le, majd teszteljük szimulátor segítségével a programunkat. Ha minden igaz, akkor most már tényleg meg fog változni a szimulátorban a PORTA értéke. Egy adott változó vagy regiszter módosulását a szimulátor piros színnel jelzi a figyelőablakban. Ezt az állapotot mutatja a 10.6. ábra.



10.6. ábra
PORTA értékének megváltozása

A 10.5. ábrán megfigyelhető, hogy az A port-ot íráskor a PORTA és LATA regiszterekkel is meg lehet címezni. A két regiszter között a port olvasásakor van eltérés. Míg a PORTx regiszter a lábainak állapotát olvassa be, addig a LATx regiszter a kimeneti tároló állapotát olvassa vissza. Ennek a különbségnak a lábak bitenkénti elérésekor (lásd későbbi fejezetekben) van nagy szerepe. A PIC architektúrára jellemző, hogy a bites utasításokat is bájtosan hajtja végre, azaz például egy bit írásakor a processzor az egész bájtot beolvassa, majd az adott bitet módosítja, és az egész bájt tartalmát visszaírja. Az angol irodalom ezeket az utasításokat Read-Modify-Write (olvasó-módosító-író) típusú utasításoknak hívja. Abban az esetben, ha egy portot bitenként szeretnénk írni, akkor szigorúan csak a LATx regisztereit szabad használni, mert a PORTx regiszter bites írásakor a bájt teljes tartalmát az adott port lábairól olvassa be, ami villamos zavarok következtében nem biztos, hogy a korábban kiírt értékkal lesz egyenlő. Ökölszabályként azt mondhatjuk, hogy port írásakor minden a LATx regisztereit használjuk, míg olvasáskor a PORTx regisztereit. Ez előbbi szabályt alkalmazva módosítuk az előző programunkat.

10.3. mintaprogram

```
/* Az első programom */

#include <p24fj128ga010.h>

main ( )           // Főprogram kezdete
{
    TRISA = 0;      // PORTA mindegyik lába kimenet lesz.
    LATA = 0xFF;    // PORTA alsó nyolc bitje egyes lesz.
}
```

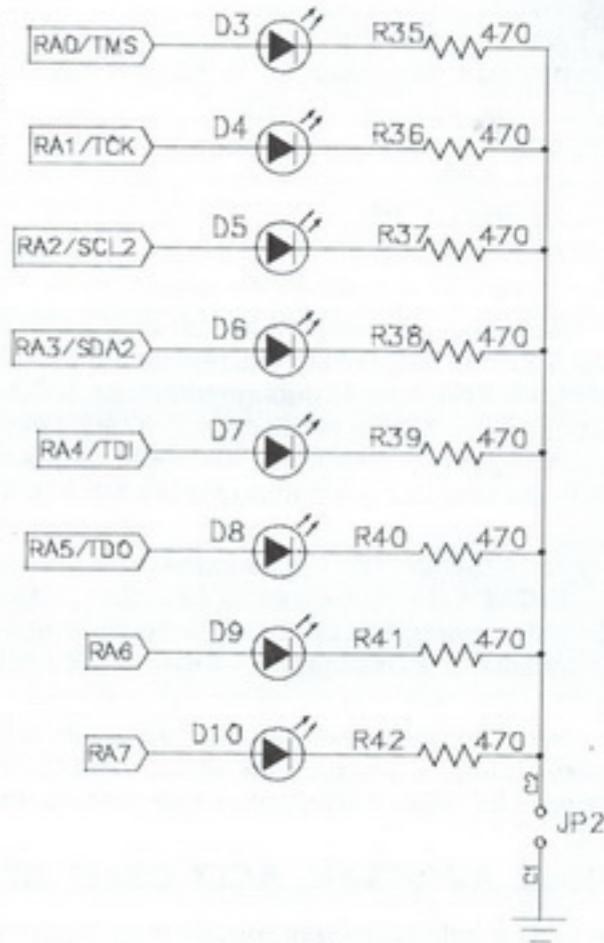
10.4.1. És a LED-ek felvillannak...

A fejezet elején nem véletlenül választottam a PIC24FJ128GA010 típusú IC-t. Az Microchip cég által forgalmazott Explorer 16 Development Board fejlesztőpanelben is az előbb említett mikrokontroller-típus található. A 10.7. ábra az Explorer 16 felhasználói kézikönyvből származik, és a fejlesztőpanelen található nyolc, egymás melletti LED kapcsolási rajzát mutatja. A kapcsolási vázlaton látható, hogy az A port alsó nyolc helyi értékű lábán egy LED és vele sorasan egy ellenállás található, amelynek a másik vége egy jumperen keresztül a földre van kötve. A LED-ek aktív egyre világítanak.

Egy apró módosítást kell elvégeznünk a programon, hogy az összes LED világítson. A Microchip 16 bites kontrollerei JTAG debugger modult is tartalmaznak, ami az A port egyes lábaira van kivezetve. Ahhoz, hogy az A port összes lábat tudjuk használni, a JTAG modult ki kell kapcsolnunk.

A pic24F szériának kettő darab konfigurációs szava van, amelyek értékét a mikrokontroller felprogramozása alatt lehet beállítani. A PIC24FJ128GA010 mikrokontroller konfigurációs szavai egyes bitjeinek jelentéséről a mikrokontroller adatlapjának 23.1 fejezetében olvashatunk részletesebben. A konfigurációs biteket értékét az MPLAB IDE felületén lehet beállítani, a **Configure → Configuration Bits...** menüpont segítségével vagy a programunk forrásállományában. Érdemes a második megoldást használni, mert így a kontrollerben lévő program és a hozzá tartozó konfiguráció egy helyen van. A forrásállományban a konfigurációs biteket _CONFIG1 és _CONFIG2 makrók segítségével adhatjuk meg (a makrókról a későbbi fejezetekben lesz szó részletesebben). A _CONFIG1 makró az egyes, a _CONFIG2 a kettes számú konfigurációs szó értékét állítja be. A makróban az egyes bitek állapotát zárójelek között, „&” jelekkel elválasztva adhatjuk meg. Beállítani csak azt a bitet kell, amely eltér az alapértelmezett értéktől, azaz egyes bitérték helyett nullás értékre van szükségünk. Mindazonáltal az összes konfigurációs lehetőség áttanulmányozásával és beállításával később, hosszas hibakeresésekkel kímélhetjük meg magunkat. Hogy melyik bit milyen névvel érhető el, az a p24fj128ga010.h fejlécállomány végén található.

- A programunk esetén a következő beállításokat érdemes elvégezni:
 - JTAG Port kikapcsolása**, ha ICD2-t vagy egyéb ICSP interfészen keresztül kapcsolódó fejlesztőrendszeret (PICkit2, ICD3, RealICE) használunk.
 - Watchdog Timer kikapcsolása**, amíg programfejlesztésünk kezdeti stádiumban van.
 - A külső oszcillátor engedélyezése**, mert az Explorer 16 fejlesztőpanelen egy 8 MHz-es oszcillátor van elhelyezve az OSC1 és az OSC2 lábaknál.
 - A külső oszcillátor típusát nagy sebességrére (HS) állítani** (4 MHz-nél gyorsabb oszcillátorok esetén)
- Ha az előbb felsorolt beállításokat megadjuk a programunkban, akkor a következő forrásállományt kapjuk:



10.7. ábra
Az A port alsó nyolc lábára kötött LED-sor kapcsolási rajza

```
10.4. mintaprogram
/* Az első programom */

#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

main ( ) // Főprogram kezdete
{
    TRISA = 0; // PORTA minden lába kimenet lesz.
    LATA = 0xFF; // PORTA alsó nyolc bitje egyes lesz.
}
```

Most már más dolgunk nincs, mint az **Explorer 16** fejlesztőpanelt az **ICD2** segítségevel a számítógépünkkel összekötni, majd a **Debugger → Select Tool** menüpont alatt az **MPLAB SIM** helyett debuggerként az **ICD2-t** kiválasztani. Az **ICD2**-vel a **Debugger → Connect** menüpont segítségével tudjuk felvenni a kapcsolatot. A lefordított programunkat a **Debugger → Program** menüpont segítségével tudjuk letölteni a mikrokontrollerbe. A futtatási lehetőségek megegyeznek a szimulátorral megismert módszerekkel.

Az Explorer 16 fejlesztőpanel újabb változatainál lehetőségünk nyílik arra, hogy PICkit™ 2 segítségével is fel tudjuk programozni és debugálni a 16 bites mikrokontroller-családjába tartozó PIM mikrokontroller-modulokat. PICkit™ 2 debugger használatához a Debugger → Select Tool menüpont alatt a PICkit 2-t kell kiválasztani.

Ha a programunkat letöltjük a mikrokontrollerbe, majd lépésenként elkezdjük futtatni, akkor ahogy a programunk utolsó utasításához érkezünk, a fejlesztőpanelen lévő nyolc darab LED elkezd világítani. Ugye nincs is ennél szébb „Hello World!” program!

10.5. AZOKNAK, AKIK CSAK AZ ASSEMBLYBEN BÍZNAK

A fejezet befejezéseként nézzük meg, hogy milyen kód fordult a kis C programunkból. Most tekintsünk el az inicializáló rész értelmezésétől (ez legyen házi feladat az igazi ínyencök...), csak az általunk írt programunkat elemezzük egy kicsit. A lefordított program assembly kódját a **View → Disassembly Listing** menüpont segítségével nézhetjük meg. Az így megnyitott ablak tartalmát a 10.8. ábra mutatja.

```
Assembly Listing
1:             /* Az első programom */
2:
3:             #include <p24fj128ga010.h>
4:
5:             main ( )
6:             {
7:                 TRISA = 0; // PORTA minden lába kimenet lesz.
8:                 LATA = 0xFF; // PORTA alsó nyolc bitje egyes lesz.
9:
10:                }
11:               return
```

10.8. ábra
A C kódból készült assembly kód

Az assembly és az eredeti C kódot összehasonlítva azt tapasztaljuk, hogy igazából semmi ördöngösségi nem történt. Az assembly esetén nagy valószínűséggel mi is így oldottuk volna meg a problémát. Persze mondhatjuk azt, hogy a **TRISA** regisztert egyenesen lehetett volna törlni a **clr trisa** utasítással, ehelyett a fordító két utasítással oldotta meg ugyanezt. Ilyenkor gondoljunk arra, hogy a fordító algoritmusok alapján dolgozik, amelyeknek a célja a biztos (helyesen működő) kód készítése. Természetesen lehetőségünk van a kód optimalizálására is. A fordító beállításánál, a **Project → Build Options...** → **Project** Menüpont segítségével megnyíló ablak **MPLAB C30** füle alatt található **optimalizációs** beállításoknál, a legörökölő menüben lehet kiválasztani az optimalizációs beállításokat. A kódoptimalizáció szintjét **s**-re állítva már a fordító is végrehajtja az általunk észrevett egyszerűsítési lehetőséget. A 10.9. ábra a bekapcsolt kódoptimalizációval lefordított programlistát mutatja. Azt azért megjegyezném, hogy a fejlesztés során érdemes a kódoptimalizációt kikapcsolni, mert ha valamilyen oknál fogva szükségünk van a C programból készített gépi kód értelmezésére, akkor a nem optimalizált kód könnyebben értelmezhető.

```
Disassembly Listing
1:             /* Az első programom */
2:
3:             #include <p24fj128ga010.h>
4:
5:             main ( )
6:             {
7:                 TRISA = 0; // PORTA minden lába kimenet lesz.
8:                 LATA = 0xFF; // PORTA alsó nyolc bitje egyes lesz.
9:
10:                }
11:               return
```

10.9. ábra
Fordítás eredménye a kódoptimalizáció segítségével

11. EGY KIS MATEMATIKA

11.1. VÁLTOZÓK

Aki eddig assembly nyelvben programozott, a változót úgy ismerte, mint egy adott nagyságú memóriaterületet. Ha egy assembly programozónak szüksége van arra, hogy elároljon egy számot a memóriában, akkor a szám nagyságától függően általában egy- vagy kétbájtos memóriaterületet foglalt le vagy definiált. Az assembly programozók csapata is kétfelé oszlik, az egyik fele az előbb említett esetben csak azt mondta meg a fordítónak, hogy két bajtra van szüksége, és egy adott névvel szeretné tovább ezt a két bajtot hívni, a másik fele a csapatnak azt is megmondta a fordítónak, hogy mondjuk 800h-tól kezdődően kerüljön lefoglalásra a memóriarész.

- A C nyelv a típusos nyelvek családjába tartozik. A család két nagy részre bontható:
- Gyengén típusos nyelvek:** Ilyen nyelv például a BASIC és nagyon sok script nyelv. Ezekben a nyelvekben a változó típusát az első használat határozza meg. Ha egy változóba először számot írunk, akkor szám lesz, ha egy szöveget, akkor pedig karakterlánc. Ezekben a nyelvekben a típusok közötti átváratás is nagyon egyszerű. Kezdők számára ezek a nyelvek könnyen elsajátíthatók, de bonyolult programok esetén komoly hibákat rejthetnek az automatikus konverziók.
 - Erősen típusos nyelvek:** Ide tartozik a C nyelv is. Ezknél a nyelvenél az első használat előtt, a változó létrehozásakor (deklarációjákor) meg kell adni a fordítónak, hogy milyen típusú lesz az adott változó. A változótípusok közötti átváratás lehetséges, de bizonyos szigorú megkötelesek mellett. Az eredeti ANSI-C szabvány még azt is megadja, hogy a függvényben használt változók deklarációja csak a függvény elején történhet meg.

Mielőtt rátérünk a C nyelvben használható változótípusokra, ismerkedjünk meg a változók három alapvető jellemzőjével (tulajdonságával):

- Tartomány:** Azon értékek halmaza, amelyet az adott változó felvehet.
- Művelet:** A változótípushoz tartozó műveletek. Például míg az egész számok körében értelmezett a maradékos osztás, a lebegőpontos számok körében nincs értelmezve.
- Reprezentáció:** Ábrázolási mód, bitkombináció. Ennek a tulajdonságnak főleg akkor van szerepe, ha különböző nyelvek vagy eszközök közötti adatcserét kell elvégezni.

11.1.1. Egész számok

A 11.1. ábrán látható táblázat a Microchip C30 fordító felhasználói kézikönyvének ötödik fejezetéből származik. A táblázat az egész számok tárolására alkalmas változókat foglalja össze. A fordítók nagy többsége hasonló változókat használ, de főleg a mikrokontrolleres világban, érdemes megnézni az adott fordító által alkalmazott változótípusokat és azok bithosszúságát. Az egyes mikrokontroller-típusokra készített implementáció által használt változótípusok a mikrokontrollerek architekturális sajátosságainak következtében eltérhetnek az eredeti ANSI-C szabványtól.

A táblázatból több minden megfigyelhető. minden változónak létezik előjeles (`signed`) és előjel nélküli (`unsigned`) változata. Alapértelmezett az előjeles változat, azaz ha nem írjuk ki a változó deklarációjákor a változótípus előtt a `signed` vagy `unsigned` előtagot, akkor a fordító a változót előjelesnek fogja tekinteni.

Az előjel nélküli egész számokat a nyelv egyszerű bináris számokként ábrázolja, az előjeles számok tárolására pedig kettes komplemensű számábrázolást alkalmaz.

Ha jobban megfigyeljük a táblázatot, akkor látjuk, hogy az int és sort változótípusok szinonimák, ugyanolyan nagyságúak. Ez az állítás a 32 bitnél kisebb architektúrákhoz készült fordítókra igaz. A 32 bites fordítóknál a sort típusú változók 16 bitesek, de az int típusú változók már általában 32 bit hosszúságúak.

| Típus | Méret [bit] | Min | Max |
|---------------------------------|-------------|------------------|--------------------|
| char, signed char | 8 | -128 | 127 |
| unsigned char | 8 | 0 | 255 |
| short, signed short | 16 | -32768 | 32767 |
| unsigned short | 16 | 0 | 65535 |
| int, signed int | 16 | -32768 | 32767 |
| unsigned int | 16 | 0 | 65535 |
| long, signed long | 32 | -2 ³¹ | 2 ³¹ -1 |
| unsigned long | 32 | 0 | 2 ³² -1 |
| long long**, signed long long** | 64 | -2 ⁶³ | 2 ⁶³ -1 |
| unsigned long long** | 64 | 0 | 2 ⁶⁴ -1 |

^{** ANSI-89 kiterjesztés}

11.1. ábra

Egész számok tárolására alkalmas változók

Változó létrehozásakor (deklaráció) meg kell adni a változó típusát, majd a változó nevét és esetleges kezdőértékét. Abban az esetben, ha egyfajta típusú változóból több példányra is szükségünk van, akkor lehetőségünk nyílik arra, hogy a változó típusa után visszavével elválasztva több változónevet is megadhassunk. A változók elméletileg tetszőleges hosszúságúak lehetnek, de fontos, hogy betűvel vagy aláhúzás karakterrel (_) kell kezdődniük. Ebből következik, hogy változó neve számjeggyel nem kezdődhet, de utána már tartalmazhat számjegyet is. Még egyszer fontos megemlíteni, hogy a nyelv kis- és nagybetűrézékeny, az alma és az Alma elnevezésű változók teljesen függetlenek egymástól. A változók nevei csak az angol ábécé betűit tartalmazhatják, így ékezes betűket ne használunk!

Nézzünk pár példát változó deklarálásra:

```
char a; // 8 bites előjeles változó
signed int egeszSzam; // 16 bites előjeles változó
unsigned long j; // 32 bites előjel nélküli változó
int szam = 50; // Kezdeti értékkadás
unsigned int a, b, c; // Több változó deklaráció
char x=5, y=6, z=7; // Több változó deklaráció értékkadással
```

Abban az esetben, ha egy változónak a változó deklarációjákor nem adunk kezdőértéket, akkor a változó létrehozásakor csak a memória foglalás történik meg, alapértelmezett értékkadás nem. Ilyen esetben a változó kezdeti értéke kisszámlíthatatlan lesz, az általa használt memóriaterület aktuális értékét veszi fel. Mielőtt egy változót először használunk, ne felejtsünk el kezdőértéket adni neki!

11.1.2. Változók a memoriában

A Microchip C30 fordító a több bájtból álló változókat „a legkisebb helyi érték elől” (little endian) bájtsorrend szerint tárolja a memoriában. A legkisebb helyi értékű bájt a legkisebb memóriacímet kapja, a legnagyobb helyi értékű bájt pedig a legnagyobb memóriacímet. A 11.2. ábra egy long típusú változó 0x100 memóriacímtől kezdődő memóriaterületen történő elhelyezését mutatja. A változó által reprezentált érték: 0x12345678.

| | | | |
|-------|-------|-------|-------|
| 0x100 | 0x101 | 0x102 | 0x103 |
| 0x78 | 0x56 | 0x34 | 0x12 |

11.2. ábra

0x12345678 értékű, long típusú változó elhelyezkedése a memoriában

Az előbbi változó két munkaregiszterben történő elhelyezése a következőképpen alakul:

| w4 | w5 |
|--------|--------|
| 0x5678 | 0x1234 |

11.3. ábra

0x12345678 értékű, long típusú változó elhelyezkedése a w4 és w5 munkaregiszterekben

11.1.3. Lebegőpontos valós számok

A C nyelv nemcsak egész számok tárolására alkalmas változótípusokkal rendelkezik, hanem lehetőséget biztosít valós számok tárolására is, lebegőpontos számábrázolás segítségével. A lebegőpontos számábrázolásnál a számot ábrázoló bitkombináció 3 részre bontható: előjelbit, mantissa (a szám alapja), karakterisztika (kitevő). A számábrázolás hasonlít a tízes számrendszerben használt normalizált alakhoz, csak jelen esetben tízes helyett kettes számrendszerben kell értelmezni a számokat. Nagyon sokáig a fordítók különböző módon tárolták a lebegőpontos számokat, manapság szinte az összes fordító az IEEE-754/1985 ajánlást alkalmazza, nincs másképpen a C30 fordító esetén sem. A 11.4. ábra a Microchip C30 fordító által használt lebegőpontos változótípusokat foglalja össze.

| Típus | Méret [bit] | Kitevő (exponens) | | Normalizált alakban | |
|--|-------------|-------------------|-------|---------------------|------------|
| | | E min | E max | N min | N max |
| float | 32 | -126 | 127 | 2^{-126} | 2^{128} |
| double (fordítói beállítástól függő, bithosszúsága egyenlő lehet a long double-lal is) | 32 | -126 | 127 | 2^{-126} | 2^{128} |
| long double | 64 | -1022 | 1023 | 2^{-1022} | 2^{1024} |

11.4. ábra

Lebegőpontos számok tárolására alkalmas változók

Ha egy új C fordítóval kezdünk dolgozni ugyanúgy, mint az egész számok esetén, a lebegőpontos számok esetében is érdemes megnézni az adott implementáció által használt változótípusokat és azok bithosszúságait. Az egész számok ábrázolására alkalmas változótípusok esetén a bithosszúság a legkisebb és a legnagyobb ábrázolható számot

határozza meg. Lebegőpontos számábrázolás esetén a bithosszúság növelésével nemcsak a maximális számábrázolási képesség határa tolódik ki, hanem egyben a változó pontossága is nő. Lebegőpontos számábrázolást használó változókat hasonlóképpen kell létrehozni, mint az egész számok tárolására alkalmas társaikat kellett:

```
float valtozo = 1.414214;
long double nagyobbValtozo;
```

Ha lebegőpontos konstans számot szeretnénk írni a programunkban, például kezdőértéket szeretnénk adni a változónknak, akkor a tizedesjelet amerikai írásmód alapján, ponttal kell jelölnünk.

11.2. OPERÁTOROK

Most, miután megismerkedtünk az alapvető változótípusokkal, és megtanultunk változót létrehozni, ismerkedjünk meg azokkal az alapvető operátorokkal, amelyekkel módosítani tudjuk a létrehozott változóink értékeit.

Mielőtt belemennénk az egyes konkrét operátorok működésébe, nézzük meg az operátorok csoportosítását:

- Az operandusok száma szerint léteznek:
 - egy-,
 - kettő- vagy
 - háromoperandusú operátorok.
- Az operátorok típusuk szerint lehetnek:
 - elválasztást végző,
 - aritmetikai,
 - relációs,
 - logikai,
 - bitműveletet végző és
 - speciális operátorok.
- Jellegük szerint az operátorok lehetnek:
 - prefix és
 - postfix operátorok.

11.3. ARITMETIKAI OPERÁTOROK

Az aritmetikai operátorok egyszerű, alapvető matematikai műveletek elvégzésére alkalmasak. Az aritmetikai operátorok családjába a következő operátorok tartoznak:

+ összeadás,
- kivonás,
* szorzás,
/ osztás,
% maradékképző (modulo),
= értékkopírozás és az
- egyoperandusú mínusz operátor.

Az összeadás (+), kivonás (-), szorzás (*) operátorok a matematikában megszokott módon működnek, kiértékelésük balról jobbra történik. Az osztás (/) operátor egész számok körében csak az osztás eredményének egész részével tér vissza, míg lebegőpontos számok esetén az osztás tört részét is kiszámolja. Ahogy már azt a változók bevezetésénél megemlíttetük, a maradékképző (%) operátor csak az egész számok körében értelmezett művelet, eredménye az osztás maradékával egyenlő.

Az értékadó (=) operátor a bal oldalán álló változó értékét egyenlővé teszi a jobb oldaláról kapott értékkel. Ha egy változó vagy kifejezés előtt egy minuszsel (-) áll egymáshoz, akkor az adott változó vagy kifejezés értékét negatív előjellel adja tovább.

A következő kód részlet az összeadó és az értékedő operátor használatát mutatja be:

```
int a=3, b=4, c;
c = a + b; // A c változó értéke 3+4=7 lesz.
```

11.4. ELVÁLASZTÁST VÉGZŐ OPERÁTOROK

Az elválasztást végző operátorok közül már megismertünk párral. Már ismerjük a logikai blokkhatárt jelző operátort {}, amely kijelöli a fordító számára az egy logikai egységek névét, mint például egy függvény törzsét.

Már találkoztunk a vessző operátorral is. A vessző operátort több változó egy utasítással történő létrehozásakor használtuk (11.1. fejezet). A vessző operátort nemcsak arra lehet használni, hogy ugyanolyan változótípusból több változót hozzunk létre egy utasítással, hanem például matematikai műveleteket is elválaszthatunk vele. Nézzük a következő kis példát:

```
int a, b, c=5, d=6;
a=c+d, b=c*d;
```

Nézzük meg ugyanezt a példát vessző operátor használata nélkül:

```
int a;
int b;
int c=5;
int d=6;
a=c+d;
b=c*d;
```

Fontos tudni, hogy a vessző operátor kiértékelése balról jobbra történik. Ennek a tulajdonságának akkor van nagy szerepe, ha a vesszővel elválasztott utasítások hatással vannak egymásra. Nézzük erre is egy példát:

```
int a=1, b=2, c=3;
a=b, b=c;
```

Mennyi lesz az a, b, c változók értéke a második sor végrehajtása után? Bontsuk ki a kifejezést! A következő példarészlet az előbbi példa vessző operátor nélküli változata. A példa mellett, megjegyzésként, egy táblázatot találunk. A táblázat egyes oszlopai a változók értékeinek változását mutatják, az adott sor végrehajtása után. A megváltozott értékű változó értéke kiemelten van jelölve. Ha egy változó értéke még nincs megadva, akkor az értékét egy kérdőjel helyettesíti, utalva arra, hogy a valóságban sem tudjuk a nem inicializált változók értékeit.

```
main() {
    int a=1; // 1 | ? | ?
    int b=2; // 1 | 2 | ?
    int c=3; // 1 | 2 | 3
    // ---+---+---
    a=b; // 2 | 2 | 3
    b=c; // 2 | 3 | 3
}
```

A C nyelv szabadsága sok, szinte fejtörő bonyolultságú kifejezésre ad lehetőséget. A C nyelvet legjobban a mesterszakács késéhez lehet hasonlítani: aki tudja kezelni, az csodára képes, de a kezdő szakács egy óvatlan mozdulattal a saját kezét is megvághatja vele. Kerüljük a bonyolult kifejezéseket, hisz azokkal csak a saját magunk és munkatársaink életét keserítjük meg feleslegesen.

Folytatva az előző mintapéldát, gondolkozzunk el azon, hogyan lehet megoldani azt, hogy egy utasítással az a és b változó a c változó értékével legyen egyenlő? Előbb láttuk, hogy ha vesszővel választottuk el a kifejezéseket, akkor a három változó értéke nem lett egyforma. Két megoldás is van a kérdésre. Vagy megcséréljük az előző mintapélda utolsó két sorát, vagy egy sokkal elegánsabb megoldást alkalmazunk:

```
a=b=c;
```

Ha már kiértékelési irányokról beszélünk, akkor érdemes megfigyelni, hogy az értékedő operátor kiértékelése jobbról balra történik. A kiértékelés irányának oka az, hogy a bal oldali változó értékét a fordító csak akkor tudja módosítani, ha az értékedő operátor jobb oldalán álló kifejezést már kiértékelte.

Az elválasztást végző operátorok családjába már csak két olyan operátor tartozik, amelyeket még nem ismertünk meg. Az egyik a tömb indexelését végző operátor, de ennek bemutatásával majd később foglalkozunk, majd a tömbök ismertetésével együtt ismertünk meg a használatával.

A zárójel () operátor gyakorlatilag megegyezik a hagyományos matematikai zárójellel. A zárójelekkel csoportosított műveletek kiértékelési sorrendje megegyezik a matematikában használt kiértékelési sorrenddel. Zárójelek segítségével összetett matematikai kifejezéseket adhatunk meg, mint ahogyan azt a következő példa is mutatja:

```
a = (b+5) * (c-2);
```

Az a változó értéke a b változó ötvelével növelt és a c változó kettővel csökkentett értékének szorzatával lesz egyenlő.

11.5. INKREMENTÁLÓ ÉS DEKREMENTÁLÓ OPERÁTOR

Aki már programozott valamelyen nyelven, az tudja, hogy a programokban nagyon sokszor fordul elő, hogy egy változó értékét eggyel növelni vagy csökkenteni kell. Ennek a műveletnek a fontosságát mutatja, hogy szinte minden assembly nyelvben megjelenik az inkrementáló (inc) és a dekrementáló (dec) utasítás. A C nyelv fejlesztői is fontosnak tartották egy változó értékének eggyel történő növelését vagy csökkentését, ezért a feladat ellátásához két új operátor vezettek be.

Ellentétben a többi aritmetikai operátorral (az értékadó operátoron kívül), ahol a változó értékét csak felhasználjuk, de nem változtatjuk meg, az inkrementáló és dekrementáló operátoronál egyenesen az adott változó értéke növekszik vagy csökken eggyel.

Nézzük meg, hogy pontosan miről is van szó:

```
i++; // Megegyezik az i=i+1; utasítással
```

```
i--; // Megegyezik az i=i-1; utasítással
```

A nyelv fejlesztői nem álltak meg az egyszerű növelésnél vagy csökkentésnél. Az operátornak két változatát vezették be:

- Előtagként használható (**prefix**) operátort. Ebben az esetben az inkrementáló vagy a dekrementáló operátor először megnöveli (++) vagy csökkenti (--) a jobb oldalon álló változó értékét, majd a változó megváltoztatott értékét továbbadja a többi operátornak feldolgozásra.

// Az utasítás végrehajtási sorrendje:

```
c = ++a; // (1.) a = a + 1;
           // (2.) c = a;
```

- Utótagként használható (**postfix**) operátort. Ebben az esetben az inkrementáló vagy a dekrementáló operátor először továbbadja a bal oldalon álló változó értékét a többi operátornak feldolgozásra, majd utána növeli (++) vagy csökkenti (--) a változó értékét.

// Az utasítás végrehajtási sorrendje:

```
c = a++; // (1.) c = a;
           // (2.) a = a + 1;
```

Az előbb megtanult két szabályt alkalmazva, játszunk egy kicsit! Mennyi lesz a következő mintapéldában az a, b és c változó értéke a főprogramunk végén?

11.2. mintaprogram

```
main ( )
{
    int a, b, c;
    a = 5;
    b = 6;
    c = ++a+b++;
}
```

Igazából az utolsó sor jelent fejtörést a számunkra, de abban az esetben már nem, ha azt is az elemeire szedjük. A következő forráskód az előző programmal megegyező működésű, azzal a különbséggel, hogy az utolsó utasításban lévő operátorok a végrehajtási sorrendjük szerint külön sorba kerültek. A vessző operátoromál már használt táblázatot, a változók értékeinek változásáról, a programkód megjegyzéseként itt is megtalálhatjuk.

11.3. mintaprogram

```
main ( )          // a | b | c
{
    // -----+
    int a, b, c; // ? | ? | ?
    a = 5;        // 5 | ? | ?
    b = 6;        // 5 | 6 | ?
```

```
++a;           // ---+---+---
               // 6 | 6 | ?
c = a + b;   // 6 | 6 | 12
b++;          // 6 | 7 | 12
```

Aki nem hiszi el, nézze meg a szimulátor segítségével a változók értékeit. A bátrabbak pedig nyugodtan próbálják meg értelmezni a fordító által készített assembly kódot is. Ez jó bemelégités a fejezet hátralévő szakaszához, ahol a fordító által generált assembly kódokkal fogunk részletesebben megismernedni.

11.6. A MEGFELELŐ VÁLTOZÓTÍPUS KIVÁLASZTÁSA

A változótípusok megismerése még kevés ahhoz, hogy megfelelő hatékonysággal használjuk a változókat. Miről is van szó? Ha azt szeretnénk, hogy a jövendőbeli programunk minden processzoridővel, minden operatív memóriával gazdaságosan bánjon, akkor nagyon óvatossan kell kiválasztanunk a megfelelő változótípust. Nézzünk egy egyszerű mintapéldát arra, hogy megértsük a probléma lényegét. Legyen egy nagyon egyszerű feladatunk: Szorozzuk össze két változó értékét, és azt tároljuk le egy harmadik változóba. A kezdőértékek megválasztásánál ügyeljünk arra, hogy a szorzás eredménye ne csorduljon túl (azaz beleférjen az adott változótípus által meghatározott szélső értékek közé).

11.6.1. A legkisebb változótípus

Próbákonban az első programunkban a lehető legkisebb memóriaterületet igénylő, egybájtós (char) változótípust használjuk. Az előjel nélküli változatot (unsigned) azért érdemes választani, mert így pozitív számok esetén nagyobb az ábrázolható szám maximum (0x00 – 0xFF). Hozzunk létre egy új projektet, és konfiguráljuk be úgy, mint ahogy azt az első projektünkben tettük (10.1. fejezet), azzal a különbséggel, hogy most nevezzük el a forrásállományunkat szam.c-nek. Végezetül másoljuk bele a következő kis programot:

11.4. mintaprogram

```
unsigned char a, b, c;

main ( )
{
    a = 0x0F;
    b = 0x10;
    c = a*b;
```

Fordítsuk le a programot a fordítási optimalizációk kikapcsolásával, majd nézzük meg a disassembly ablakban a fordítás eredményeként kapott gépi kódot. A könnyebb kódértelezés érdekében az egyes assembly sorok mellett megjegyzésben megtalálható az adott utasítás által végzett feladat működésének leírása is.

| | | |
|----------------------|--|---------------------------------|
| a = 0x0F; | | // 0xf érték mozgatása a W0-ba |
| mov.b #0xF,0x0000 | | // W0 mozgatása az a változóba |
| b = 0x10; | | // 0x10 érték mozgatása a W0-ba |
| mov.b #0x10,0x0000 | | // W0 mozgatása a b változóba |
| mov.b 0x0000, 0x0801 | | |
| c = a*b; | | |

```

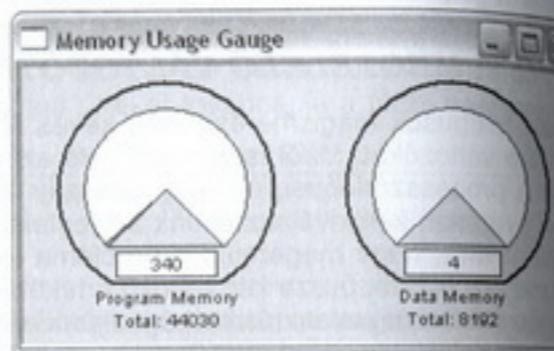
mov.w #0x800,0x0002      // a változó mozgatása a W1-be
mov.b [0x0002],0x0002
mov.b 0x0801,0x0000      // b változó mozgatása a W0-ra
mul.ss 0x0002,0x0000,0x0000 // W0 = W1*W0
mov.b 0x0000,0x0802      // W0 mozgatása a c változóba

```

A programunk három bajtot használt el az adatmemoriából, mert három darab egybájtos változót deklaráltunk. Gyakorlatban, mivel a 16 bites PIC mikrokontrollerek adatbusza 16 bit szélességű, így a három bajtot négy bajtra, jobban mondva két szóra egészítette ki a linker.

Fordítás után az adat- és a programmemória foglaltságát mi is meg tudjuk nézni a **View → Memory Usage Gauge** menüpont segítségével. Az említett menüpont egy ablakot nyit meg, ahol két, az autók sebesség- vagy fordulatszámmérőjéhez hasonló ábra jelenik meg.

Az egyik „kijelzőről” a felhasznált programmemória, míg a másikról a felhasznált adatmemória olvasható le. A programunk lefordítása utáni memóriaterhelést a 11.5. ábra mutatja.



11.5. ábra
Felhasznált memória

11.6.2. Egy lépéssel feljebb

Következő lépésként módosítuk az előző programunkat. Cseréljük le az `unsigned char` változótípust `unsigned int`-re. Az új programunk következőképpen alakul:

```

unsigned int a, b, c;

main ( )
{
    a = 0x0f;
    b = 0x10;
    c = a*b;
}

```

Fordítsuk le az új programunkat, majd nézzük meg a disassembly ablakban a kapott assembly kódot.

```

a = 0x0f;
mov.w #0xf,0x0000      // 0xf érték mozgatása a W0-ra
mov.w 0x0000,0x0800      // W0 mozgatása az a változóba
b = 0x10;
mov.w #0x10,0x0000      // 0x10 érték mozgatása a W0-ra
mov.w 0x0000,0x0802      // W0 mozgatása a b változóba
c = a*b;
mov.w 0x0800,0x0002      // a változó mozgatása a W1-be
mov.w 0x0802,0x0000      // b változó mozgatása a W0-ra
mul.ss 0x0002,0x0000,0x0000 // W0 = W1*W0
mov.w 0x0000,0x0804      // W0 mozgatása a c változóba

```

Az előző disassembly kódolt nem kaptunk különlegesebben más kódot. Az egyik nagy változás az, hogy bájtos (`mov.b`) adatmozgató utasítások helyett szavas (`mov.w`) adatmozgató utasítások jelentek meg. A változók memóriacímei is megváltoztak, egy változó egy bajt helyett két bajtot foglal el, így a teljes adatmemória-használatunk hat bajtra módosul.

Még egy apróság is észrevehető a két kód között. A `char` változótípussal fordított kódunk egy felesleges „indirekciós varázslatot” hajt vége, míg az `int` változótípussal a szorzás előkészítésénél lévő adatmozgatás sokkal tisztább. Félelmet ne esétek, a fordító a `char` változótípussal is helyes kódot fordít, csak a kelleténél kicsit bonyolultabban. Ebből is látszik, hogy a fordítót is emberek készítik. Gondolunk csak bele: mi is mindig a legegyszerűbb utat választjuk... ? ☺

11.6.3. Irány a long!

Következő lépésként az `int` változótípust cseréljük le `long` típusra. Az így kapott forráskód a következőképpen néz ki:

11.5. mintaprogram

```

unsigned long a, b, c;

main ( )
{
    a = 0x0f;
    b = 0x10;
    c = a*b;
}

```

Ez a pici programkód-módosítás már jóval jelentősebb assembly kódmodosulást, jobban mondva kódnövekedést okoz. Azonban ne felejtsünk el egy „apróságot”. Jelenleg két 32 bites számot szorzunk össze 16 bites processzorral. Vallja be mindenki öszintén, ki szeret ilyen kódot gépi kódban megírni, és még az osztásról nem is beszéltünk...

```

a = 0x0f;
mov.w #0xf,0x0000
mov.w #0x0,0x0002
mov.w 0x0000,0x0800
mov.w 0x0002,0x0802
b = 0x10;
mov.w #0x10,0x0000
mov.w #0x0,0x0002
mov.w 0x0000,0x0804
mov.w 0x0002,0x0806
c = a*b;
mov.w 0x0800,0x000c
mov.w 0x0802,0x000e
mov.w 0x0804,0x0004
mov.w 0x0806,0x0006
mul.uu 0x000c,0x0004,0x0010
mul.ss 0x000c,0x0006,0x0000
mov.w 0x0012,0x0008
add.w 0x0008,0x0000,0x0008
mul.ss 0x0004,0x000e,0x0000

```

```

add.w 0x0008,0x0000,0x0008
mov.w 0x0008,0x0012
mov.w 0x0010,0x0808
mov.w 0x0012,0x080a

```

Azonkívül, hogy a programkódunk jelentősen megnőtt, a felhasznált memória is 12 bájtra növekedett, mert három darab négybájtos változót hoztunk létre.

11.6.4. És végül a 64 bit

Módosítsuk megint a programunkat, a változótípusunkat cseréljük le long long-ra.

11.6. mintaprogram

```

unsigned long long a, b, c;

main ( )
{
    a = 0x0f;
    b = 0x10;
    c = a*b;
}

```

Az így kapott assembly kódot már csak erős idegzetű bitvadászoknak ajánlom bogarázsra. A kód nézegetésekor felfedezhetjük, hogy a szorzás műveletét a fordító nem a szorzás helyén végi el, hanem egy függvényt hív meg. A függvény forráskódja megnézhető az MPLAB C30 fordító `src\libm\alkonyvtárban` található `multid3.c` állományban. A fordító fejlesztői, gondolom, azért választották ezt a megoldást, mert így hosszadalmas matematikai számolások esetén kisebb kód érhető el.

11.6.5. Elhagyjuk az egész számokat

Ha megint módosítjuk a programunkat, és az `unsigned long long` változótípust lecseréljük `float`-ra, akkor a következő forráskódot kapjuk:

11.7. mintaprogram

```

float a, b, c;

main ( )
{
    a = 0x0f;
    b = 0x10;
    c = a*b;
}

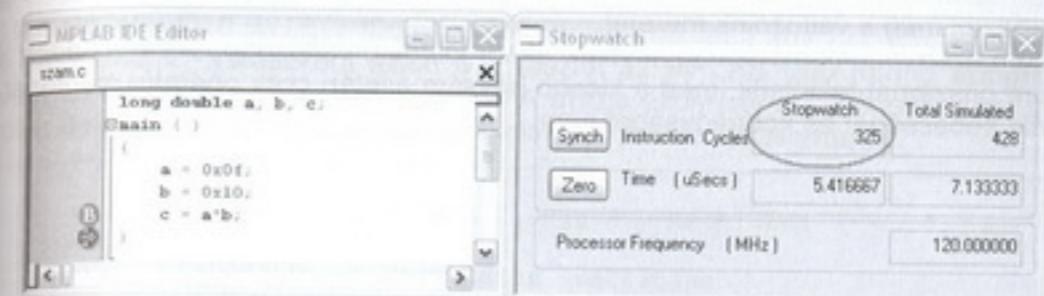
```

Fordítás után észrevehetjük, hogy a fordító hasonlóképpen járt el, mint a `long long` típus esetén, azaz a szorzást függvényhívás segítségével oldotta meg.

11.6.6. Mérési eredmények

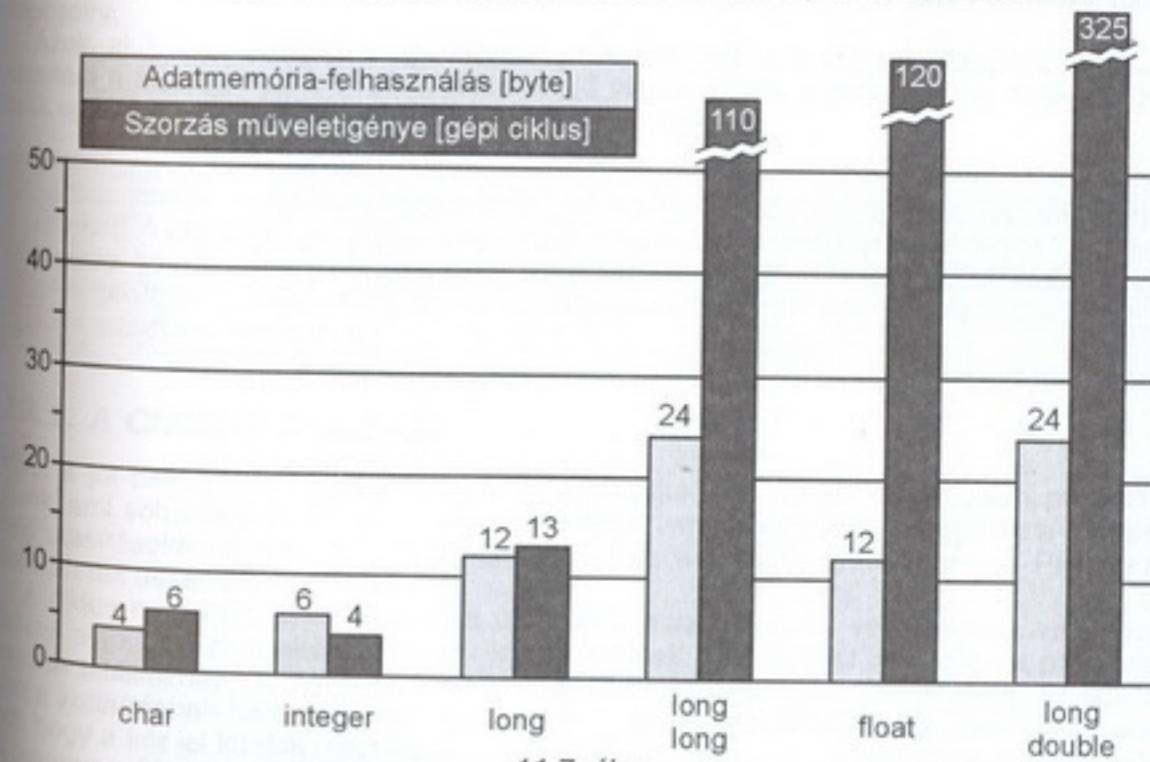
Ezt a könyvet valószínűleg főként mérnökkéletű olvasók fogják bongészni. Ebből a feltételezésből kiindulva, úgy gondolom, hogy így a fejezet végén nem árt elvégeznünk egy kis mérést. Futtassuk le az előző mintapéldákat. Nézzük meg a példákban használt három változó adatmemória-igényét és a szimulátor stopperórája (**Debugger** – **StopWatch**) segítségével a példában használt szorzás (`c = a*b;`) utasítás műveletigényét is. A szorzás utasítás műveletigényét két egymás utáni sorban elhelyezett töré-

ponttal célszerű elvégezni. Kettőt kattintva egy adott sorra a sor elején megjelenik egy **B** betű. Ez azt jelenti, hogy töréspont kerül az adott sor végrehajtása elő. Futtatás alatt, ha a szimulátor (és a fizikai debuggerek is, mint például az ICD2) törésponthoz érkezik, akkor az adott sort már nem hajtja végre, felfüggeszti a program futtatását. Az adott sorra újból kettőt kattintva a töréspont eltávolítható. A 11.6. ábra a szimuláció során készült kép, amely a forrásállományt tartalmazó ablak és a szimulátor stopperórája ablakának tartalmát mutatja.



11.6. ábra
Töréspontos futtatás

A 11.7. ábra a könyv írásakor mért eredményeket szemlélteti: részben a memóriaigényt, részben pedig a szorzó utasítás végrehajtásának gépi ciklusban mért idejét. Megfigyelhető, hogy a változó nagyságának növelésével a szorzás művelet végrehajtási ideje exponenciális jelleggel nő. A teszteléskor a mintaprogramok optimalizáció nélkül, az MPLAB C for PIC24F / PIC24H / dsPIC30 / dsPIC33 v3.10 fordító(k) segítségével lettek lefordítva.



11.7. ábra
Adatmemória-felhasználás (byte) és a szorzás műveletigénye (gépi ciklus) a változótípusok függvényében

11.6.7. Mikor melyik változótípust érdemes választani?

Erre a kérdésre mindenkor a program írójának (tervezőjének) kell a választ megadnia. Nézzünk pár szabályt arra, hogy milyen szempontok alapján tudjuk a változónkat kiválasztani:

- Ha az adott feladat megoldható az egész számok körében, akkor ne használunk lebegőpontos számokat.
- Nézzük meg, hogy mekkora számokat kell tárolnunk, a matematikai műveleteinknek milyen számtartományban várhatók az eredményeik. Ennek a tudásnak a birtokában határozzuk meg a változóink méretét.
- Ha nem szenvedünk memóriahiányban, akkor kiinduló változótípusként az adott architektúra bitszámát tekintsük, mert a kisebb bitszám esetén csak operatív memóriát spórolunk, sebességet és kódnagyságot nem.

12. KÖRBE-KÖRBE

Az éles szemű olvasó az első program (10. fejezet) tanulmányozása közben feltehette a kérdést: Mi történik az után, hogy a főprogramunkban a `main()` függvényünk utolsó sorára is végrehajtásra került? Mi történik az után, hogy a programunk kilép a `main()` függvényből?

Ha visszalapozunk a 231. oldalra, akkor az ott található disassembly ablakban (10.8. ábra) látható lefordított `main()` függvényünk utolsó utasítása egy `return`. Hová tér viszszá a programunk? Válaszként álljon egy részlet az MPLAB C30 fordító `src\pic30\` alkönyvtárból található `crt0.s` állományából. A `crt0.s` állomány a C kódunk előtt, a mikrokontroller bekapsolását követően lefutó inicializáló kód forrásállománya.

```
1:    call _main           ; a main() függvény meghívása
      .pword 0xDA4000     ; Szimulátor leállítása
      reset                ; Processzor újraindítása (reset)
```

Az assembly állományból látható, hogy a `main()` függvényünk utolsó utasításának végrehajtása után a program visszatér az inicializáló kódhoz, ami újraindítja a kontrollert. Ez hasonlít ahhoz a megoldáshoz, mikor egy egyszerű program befejezése után újraindítjuk az otthoni számítógépünket. Ahogy ezt érezzük, az első programunkban nem a legelegánsabb megoldást választottuk, mert **egy mikrokontrolleren futó program soha nem ér véget!** Ha kikapcsoljuk a saját mobiltelefonunkat, akkor sem ér véget benne a program, mert általában egy gomb hosszabb lenyomásával vissza tudjuk kapcsolni.

Azok, akik assemblyben programoznak, már mondják is a választ: Tegyük egy `goto` utasítást a főprogramunk végére, azzal majd visszaugrunk a programunk elejére. Ezzel csak egy probléma van, a C nyelvben nincs `goto` utasítás.

Jobban mondva létezik a nyelvben a `goto` utasítás, de használata nagyon nem ajánlott! A címkére való ugrás a nyelvben történelmi okokból maradt benne. A manapság használt összes tervezési módszer elveti a `goto` utasítás használatát, így mi is kerüljük a használatát! Szinte az összes algoritmus megvalósítható `goto` utasítás használata nélkül is.

12.1. A CIKLUS FOGALMA

Mivel a jól bevált `goto` utasítást nem használhatjuk, így azzal, hogy olyan programot írunk, ami soha nem ér véget, új nyelvi elemekre van szükségünk, mégpedig a cikluskezelő utasításokra. A ciklusutasítás fogalmát kevés assembly nyelv ismeri, a PIC mikrokontrollerek assembly nyelve sem.

A ciklus nem más, mint egy adott utasításhalmaz többszöri végrehajtása valamelyen feltétel alapján. Ezt a feltételt hívjuk ciklusfeltételnek. Ciklusokkal nem csak a programozásban találkozhatunk. Aki már énekel vagy játszott valamelyen hangszeren, az találkozott a kottaírásban használt ismétlöjelékekkel `|| :||`. Az ismétlöjelek arra utasítják az olvasót, hogy a két jel közötti részt egyszer meg kell ismételnie. A ciklusvezérlő utasításokkal a programunkban mi is elhelyezhetünk ismétlöjelket, de az ismétlés számát mi határozuk meg, a ciklusfeltétel segítségével.

- A C nyelv háromfajta ciklust definiál:
- Előírás, egyszerű ciklus: `while`

- Elől tesztelő, lefutó típusú ciklus: `for`
- Hátul tesztelő, egyszerű ciklus: `do-while`

Az elől és hátul tesztelő ciklusszervezés között a különbség az, hogy az elől tesztelő ciklusok esetén a ciklusfeltételt a program a ciklusba lépés előtt megvizsgálja, és ha nem teljesül, akkor a ciklus tartalmát (ciklusmagot vagy ciklustörzset) egyszer sem hajtja végre. A hátul tesztelő ciklusok esetén a ciklus tartalma egyszer biztosan lefut, majd csak az első futás után történik meg a ciklusfeltétel-vizsgálat.

A nyelvben lévő minden három ciklus feltétele úgynevezett **benntartó** feltétel. A benntartó feltétel azt jelenti, hogy a program addig **nem lép ki a ciklus vérehajtásából**, amíg a ciklusfeltétel teljesül, vagyis **igaz**.

Mielőtt a konkrét háromféle ciklusszervező utasítással megismernéknénk, nézzük meg, hogy a jövendőbeli ciklusaink ciklusfeltételét milyen operátorok segítségével adhatjuk meg.

12.2. RELÁCIÓS OPERÁTOROK

A relációs operátorok feladata az, hogy összevessék a jobb és a bal oldalukon álló kifejezések értékeit, majd a megadott feltétel alapján igaz vagy hamis értékkel térjenek vissza. Igaz vagy hamis? ... Hiszen a nyelvben nincs is logikai változó! Az előző fejezetben, amikor a változókat ismertettük, akkor a legkisebb változó is bájtos volt. Az eredeti ANSI C nyelv nem ismeri a bites változótípust. Természetesen léteznek olyan implementációk, ahol a bites változót bevezették, de az implementációk nagy többségében, beleértve a GNU C-t is, nem. Attól függetlenül, hogy az adott implementációban létezik-e bites változó vagy sem, a **logikai igaz vagy hamis egész számok körében értelmezett**.

A C nyelvben egy kifejezés

- igaz**, ha az értéke nem nulla,
- hamis**, ha az értéke nulla.

Az előbbi állításból következik az, hogy a -120, 50, -1 és 1 is igaz érték, csak a 0 számít hamisnak. A **relációs** operátorok az előbbi szabályt betartva, ha a **feltétel nem teljesül**, akkor **nullával, különben eggyel** térnek vissza. A nyelv hat relációs operátor definiál:

- `<` kisebb
- `>` nagyobb
- `<=` kisebb vagy egyenlő
- `>=` nagyobb vagy egyenlő
- `==` egyenlő
- `!=` nem egyenlő

A relációk megegyeznek a matematikában használt relációs kifejezésekkel. Egy operátorról érdemes kicsit részletesebben is megnézni: az **egyenlőség** `==` operátor. Az egyenlőség operátor könnyen összetéveszthető az **értékkadó** = operátorral, pedig a feldatuk teljes mértékben különbözik. A rossz hír az, hogy feltételvizsgálatkor a nyelv me engedi az értékkadást is. Például ha egy ciklusfeltételben két egyenlőségjel helyett csak egy egyenlőségjelet teszünk ki, akkor a programunk hiba nélkül lefordul, maximum csak figyelmeztetést kapunk a fordítótól. A program a feltételkifejezés futása közben értékvizsgálat helyett a bal oldalon álló változó értékét az értékkadó operátor, a jobb oldalon álló kifejezés értékére fogja módosítani, és a feltétel teljesülése annak a függvényében lesz igaz vagy hamis, hogy a változó új értéke különbözik-e nullától vagy nem. Ezért feltételek írásakor minden győződjünk meg arról, hogy **egyenlőségvizsgálatkor két egyenlőségjelet raktunk-e ki!**

12.3. LOGIKAI OPERÁTOROK

Az előbb megismert relációs operátorokkal csak egy feltételt tudunk megadni. Mi van abban az esetben, ha több feltételre van szükségünk? Például ha egy ciklusfeltételben a változónak ötnél nagyobbnak és tíznél kisebbnek kell lennie. Azt hogyan tudjuk megadni? Az ilyen problémák leírására nyújtanak megoldást a logikai operátorok. A C nyelv, a matematikából jól ismert három alapvető logikai műveletet definiálja:

- `&&` logikai **és**
- `||` logikai **vagy**
- `!` logikai **nem**

A logikai operátorok, a relációs operátorokhoz hasonlóan, az egész számok körében dolgoznak, és **eredményük 0, ha hamis az állítás, vagy 1, ha igaz az állítás**. A 12.1. ábra a logikai operátorok eredményeit foglalja össze.

| a | b | a && b | a b | !a |
|-------|-------|--------|--------|-------|
| hamis | hamis | hamis | hamis | IGAZ |
| hamis | IGAZ | hamis | IGAZ | IGAZ |
| IGAZ | hamis | hamis | IGAZ | hamis |
| IGAZ | IGAZ | IGAZ | IGAZ | hamis |

12.1. ábra
Logikai operátorok eredményei a és b kifejezések függvényében

A logikai nem operátor egyoperandusú, például az ötnél nem nagyobb számokat a következőképpen is megadhatjuk: `!(szam > 5)`, ami matematikailag megegyezik a `szam <= 5` kifejezéssel. A fent említett feltételt az újonnan tanult operátorainkkal a következő módon tudjuk megadni:

`szam > 5 && szam < 10`

A jobb átláthatóság kedvéért ajánlatos a zárójelek használata. A következő kifejezés megegyezik az előbbi kifejezéssel:

`(szam > 5) && (szam < 10)`

Egy érdekkességet érdemes megemlíteni a logikai operátorok feldolgozásával kapcsolatban. A logikai operátorok kiértékelése balról jobbra történik, de csak addig, amíg a feltétel nem válik **vagy esetén igazzá**, illetve **esetén hamissá**. Az előző példában, ha a `szam` változó értéke 3, akkor a kifejezés első fele nem teljesül, ezért az **és** művelet második felét már meg sem vizsgálja a fordító. Ennek a tulajdonságának akkor van jelentősége, ha például `++` vagy `--` operátorokat használunk a feltételen, mert ilyenkor a feltételektől függően kerülnek vérehajtásra, ezért kerüljük az ilyen bonyolult kifejezések használatát!

12.4. WHILE CIKLUS

A **while** utasítás a legegyszerűbb, elől tesztelő ciklusszervező utasítás. A **while** utasítás után zárójelek között kell megadni a ciklusfeltételt, majd halmazzárójelek között a ciklus magját. Abban az esetben, ha csak egy utasítást tartalmaz a ciklusmag, akkor lehetőség van a halmazzárójelek elhagyására. Ilyen esetben, a **while(...)** utasítás után, csak a

ciklus törzsét képző egy utasítást kell leírmunk, pontosvesszővel lezárva. Másképpen megfogalmazva, ha a `while(...)` utasítás után nem teszünk halmazzárójelet, akkor a ciklus törzse csak a következő utasítás lesz. Ezért ajánlatos egy utasításos ciklustörzset is halmazzárójelek közé elhelyezni, mert így később nem lesz probléma akkor, ha újabb utasításokkal bővítjük ki a már elkészített ciklustörzset.

A nyelvben létezik az üres utasítás fogalma is, ami nem áll másból, mint egy pontosvesszőből. Ha a `while` utasítás után pontosvesszöt teszünk, akkor azt hiszi a fordító, hogy üres ciklustörzzel szeretnénk a ciklusunkat használni, így az utána lévő programrész, abban az esetben is, hogy ha halmazzárójelek között van, nem kerül bele a ciklustörzsbe! Ezért a `while`, majd a későbbiekben bemutatásra kerülő `for` ciklusutasítás után soha se tegyünk pontosvesszöt, kivéve, ha üres ciklusmagot szeretnénk használni.

Most, hogy már mindenki elrettent a ciklusok használataitól, jöjjön a `while` ciklus általános alakja és a hozzá tartozó folyamatábra (12.2. ábra).

```
while( kifejezés )
{
    ciklustörzs;
}
```

Nézzünk egy rövid példát a `while` ciklus alkalmazására. Számoljuk ki az első tíz pozitív egész szám összegét. Természetesen ezt a számítani sor összegképlettel is ki tudjuk számolni, de azért ellenőrizzük, hogy igaz-e a képlet:

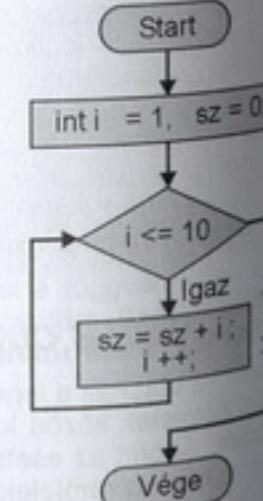
$$S_{10} = 10 \cdot \frac{1+10}{2} = 55$$

Nézzük meg először a programunk megvalósítását, majd a folyamatábráját is. (12.3. ábra).

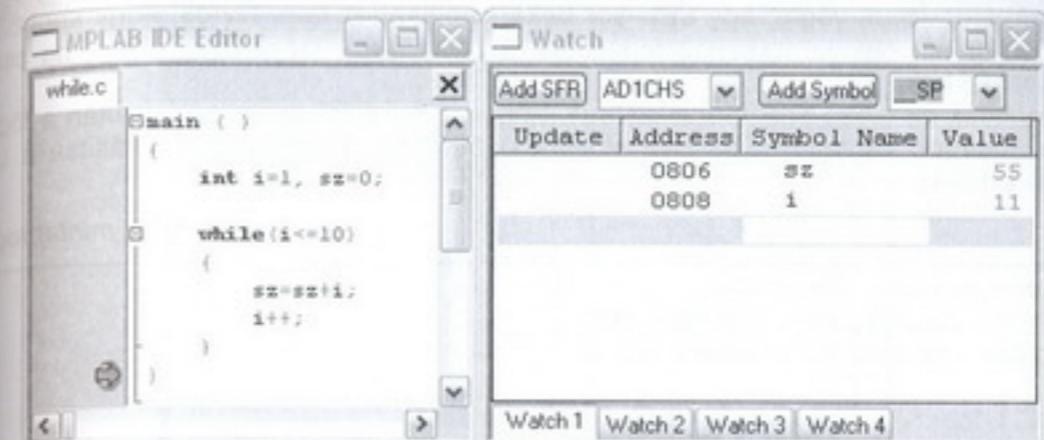
12.1. mintaprogram

```
main ( )
{
    int i=1, sz=0;

    while(i<=10)
    {
        sz=sz+i;
        i++;
    }
}
```

12.3. ábra
Az első tíz pozitív egész szám összegét kiszámoló algoritmus folyamatárája, while ciklussal12.2. ábra
While ciklus folyamatábraja

Ha a programunkat beírtuk és lefuttattuk, akkor a 12.4. ábra eredményét kell kapnunk. Mégis igaz a képlet...?!



12.4. ábra

A `while` ciklus működését bemutató program szimulációjának eredménye

Az `sz` változó eredménye tényleg annyi, mint amennyire számítottunk. De mi van az `i` változóval? Miért 11 az eredménye, hiszen az első 10 szám összegét számoltuk ki? A választ úgy kaphatjuk meg, ha lépésenként hajtjuk végre a programunkat. Gondoljunk csak bele: A ciklusfeltételünk `i<=10`, azaz `i` változó 10-es értékénél még egyszer lefut a ciklusmag, hozzáadja az `i` értékét az `sz` változóhoz, így kapjuk meg az 55-öt, majd meg-növeljük az `i` változó értékét egyelőre, 11-re, ami már a ciklusfeltételt nem fogja kielégíteni, így a program kilép a ciklusból.

Most térdünk vissza a 10.3. fejezetben megtervezett programunk utolsó változatához, és egészítsük ki a programot úgy, hogy a program soha ne érjen véget. Ahhoz, hogy a főprogramban lévő ciklusból ne tudjon kilépni a programunk, úgy kell megadnia a ciklus-feltételt, hogy az mindig igaz legyen. Ennek az egyik legegyszerűbb megoldása, ha a feltételbe fix számot írunk, mondjuk, az 1-et. Az ilyen típusú ciklusokat végtelen ciklusoknak nevezzük, ezzel is jelezve, hogy a ciklusmag végtelenszer fog végrehajtódni. Ha belegondolunk, a minden nap gyakorlatban is a programjaink nagy része inicializálás után egy végtelen ciklusban tölti az idejét. Végezzük el még egy apró változtatást az eredeti "Szia világ!" programon: Csak az A PORT alsó nyolc lábát állítsuk be kimenetnek, a többi lábat hagyjuk meg bemenetnek, hogy biztosan ne okozzunk a nem használt lábakon keresztül rövidzárlatot az Explorer 16 fejlesztőpanelen.

12.2. mintaprogram

```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

main ( ) // Főprogram kezdete
{
    while(1) // Végtelen ciklus
    {
        TRISA = 0xFF00; // PORTA alsó nyolc lába kimenet lesz.
        LATA = 0xFF; // PORTA alsó nyolc bitje egyes lesz.
    }
}
```

12.4.1. Egy kis dinamizmus

Következő lépésként alakítsuk át a programunkat úgy, hogy a programunk villogtassa a LED-eket. Gondoljuk végig: egy LED-sor bekapcsolásához nem biztos, hogy kontrollerre van szükségünk, egy jól használt tápellátás is csodákra képes, főleg akkor, ha csak a LED-eket kell bekapcsolnunk.

Módosítsuk az előző programunkat úgy, hogy az A PORT inicializálása után a főprogramban található végtelen ciklusban először 0x0000-ra, majd 0x00FF-re állítsa a LATA regiszter értékét.

12.3. mintaprogram

```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

main ( )           // Főprogram kezdete
{
    TRISA = 0xFF00; // PORTA alsó nyolc lába kimenet lesz.

    while(1)
    {
        LATA = 0x0000; // PORTA összes bitje nulla lesz.
        LATA = 0x00FF; // PORTA alsó nyolc bitje egyes lesz.
    }
}
```

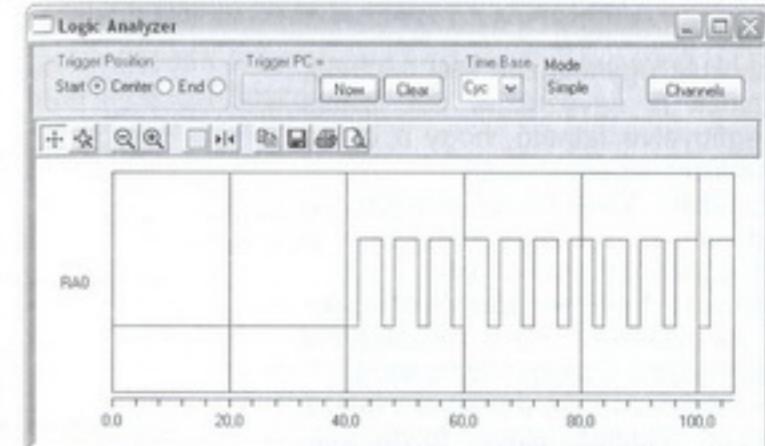
Teszteljük le a programunkat. Első körben a teszteléshez a szimulátorról érdemes használni, azon belül is az automatikus, lépésenkénti programfuttatást (animálást). Ezt a funkciót a **Debugger → Animate** menüpont alatt találjuk meg. Animálás alatt a program futtatása lépésenként történik úgy, hogy a megfigyelt változók értékei minden lépés után frissülnek.

A villamosmérnök egyik kedvenc eszköze az oszcilloszkóp. Szeretjük látni a villamos jeleinek értékeit, az idő függvényében. Tudjuk, egy jó ábra száz szónál is többet jelent... Jelenítsük meg a szimulátorunk segítségével mi is az A PORT egy lábának kimeneti változását! Ehhez a művelethez hívjuk a logikai analizátor segítségét. Mielőtt az analizátort kinyitnánk, engedélyezni kell a nyomkövetés funkciót. A **Debugger → Settings...** segítségével kinyíló ablak **Osc/Trace** füle alatt pipáljuk be a **Trace All** funkciót. Ha már ki van nyitva a konfigurációs ablak, akkor érdemes beállítani a szimulátorban használt processzorfrekvenciát is 8 MHz-re.

Ha az oszcillátor áramkör négyeszeres frekvenciaszorzó (PLL) funkcióját is bekapcsoljuk, akkor 8 MHz-es oszcillátor használata esetén 32 MHz-es processzorfrekvencia érhető el. Ilyen esetben a szimulátor frekvencia-beállítását is 8 MHz-ről 32 MHz-re kell átállítani.

Miután bekapcsoltuk a nyomkövetés funkciót, már elérhetővé vált a logikai analizátor, amit a **View → Simulator Logic Analyser** menüpont segítségével tudunk elindítani. A kinyíló ablakban **Channels** gomb megnyomása után egy újabb ablak nyílik meg. A frissen kinyitott ablakba a bal oldalon lévő listából keressük ki az RA0 lábat, majd az **Add ⇒** gombra kattintva adjuk hozzá az ábrázolandó jelekhez. Az **OK** gombra kattintás után az analizátor ablakában bal oldalon megjelent az RA0 felirat.

Most már csak egy feladatunk maradt hátra: indítsuk el a programunk szimulációját az animálás funkció segítségével! Ha minden jól hajtottunk végre, akkor a saját szimulátorunk ablakában a 12.5. ábra által mutatott ablakhoz hasonló eredményt kell látnunk.

12.5. ábra
Logikai analizátor

12.4.2. A villogó LED-ek

Most újból vegyük elő az **Explorer 16** fejlesztőeszközünket, és töltük le a programunkat. Elölne ne felejtse el a debugger menüben az MPLAB SIM helyett az általunk használt fejlesztőrendszer, jelen esetben az MPLAB ICD 2-t kiválasztani. Indítsuk el a programunkat az automatikus lépésenkénti programfuttatással (animálással)! Ha minden igaz, akkor egy kis idő elteltével bekapcsolódnak a fejlesztőpanelen lévő LED-ek, majd kialszanak, és így tovább.

Mi van abban az esetben, ha a programunkat nem animálva, hanem a **Debugger → Run** menüpont segítségével valós időben futtatjuk? Az eredmény egy fél fényerővel világító LED-sor lesz, mert a mikrokontroller 1,5 µs alatt kapcsolja be és ki a LED-eket (másodpercenként több mint 66 ezerszer). Ezt a sebességet a szemünkkel nem tudjuk követni, ezért látjuk gyengébb fényerővel világítani a LED-sort.

Lassítsuk a LED-ek villogását úgy, hogy az a szemünkkel is látható legyen. Ehhez most egy szoftveres késleltetést fogunk elkészíteni, ami nem más, mint egy ciklus, ami elég sokszor fut le ahhoz, hogy a processzor lefoglalja, így a LED-ek be- és kikapcsolása között több idő teljen el.

A szoftveres késleltetés nem a legszebb megoldás, mert a késleltetés alatt a mikrokontroller nem tud másra figyelni. Nem beszélve arról, hogy az elkészítésre váró programok elemes táplálás esetén igazi elemfogyasztó program lesz... Később, a 17. fejezetben majd lecseréljük a késleltetésünket hardveres, megszakításalapú megoldásra.

A késleltető ciklus elkészítéséhez egy új ciklusutasítást fogunk használni, a **for** ciklust.

12.5. FOR CIKLUS

A **for** ciklus számlálóvezérelt, elől tesztelő, lefutó típusú ciklus. A BASIC nyelv elterjedésével a **for** ciklus is elterjedt. A manapság használatban lévő szinte összes magas szintű nyelvben megtalálható ez a ciklustípus. A C nyelvben implementált **for** ciklus nagyon kölcsönös eszköz. A ciklusutasítás paramétereinek három kifejezés adható meg, az initializáló, a feltétel és a léptető kifejezés. A ciklus általános alakja és folyamatábrája (12.6. ábra) a következőképpen néz ki:

```
for( inicializáló_kifejezés; feltétel_kifejezés; léptető_kifejezés )
{
    ciklustörzs
}
```

A **for** ciklust szokták úgynevezett elől tesztelő, hátul növelő ciklusnak is nevezni. A **for** ciklus folyamatábráját megfigyelve látható, hogy a ciklus végrehajtása a következő lépésekkel áll:

- Először az inicializáló kifejezéssel kezdődik a ciklus végrehajtása. Az inicializáló kifejezés a ciklusfeltételtől függetlenül biztosan lefut.
- Mielőtt a végrehajtás belépne a ciklustörzsbe, megvizsgálja a ciklusfeltételit. Ha a ciklusfeltétel igaz, akkor a végrehajtás a ciklustörzsre kerül.
- A ciklustörzs végrehajtása után a program futása a léptető kifejezés feldolgozásával folytatódik, majd a végrehajtás visszatér a ciklusfeltételhez.
- A ciklus végrehajtása akkor fejeződik be, ha a ciklusfeltétel hamissá válik.

Nézzük meg a **while** ciklusnál használt példának, vagyis az első tíz pozitív egész szám összegét kiszámoló algoritmusnak a **for** ciklussal történő megvalósítását és folyamatábráját (12.7. ábra).

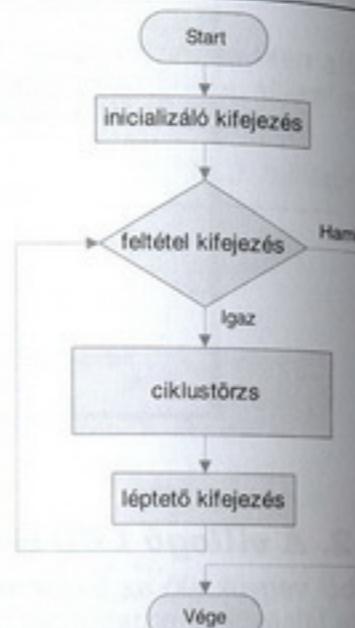
12.4. mintaprogram

```
main ( )
{
    int i, sz;
    for(i=1, sz=0; i<=10; i++)
    {
        sz=sz+i;
    }
}
```

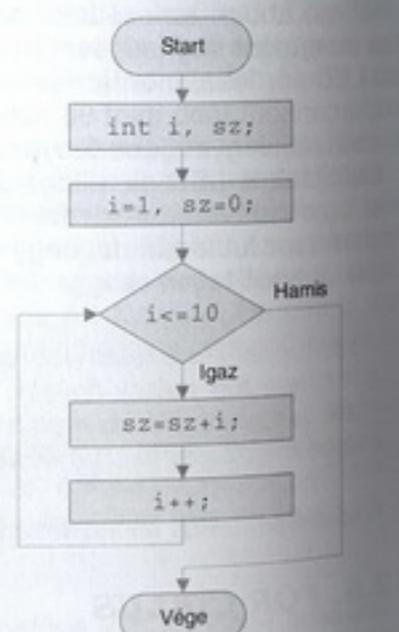
12.7. ábra

Az első tíz pozitív egész szám összegét kiszámoló algoritmus folyamatábrája, **for** ciklussal

A **for** ciklusutasításában az inicializáló és a léptető kifejezést is el lehet hagyni. Ha mind a két kifejezést elhagyjuk, akkor **while** típusú ciklust kapunk: **for(; ;)**. Abban az esetben, ha a ciklusutasítás minden három paraméterét üresen hagyjuk, akkor végtelen ciklust definiálunk.



12.6. ábra
For ciklus folyamatábrája



```
for( ; ; )
{
    // Végtelen ciklus
}
```

Most, hogy megismerkedtünk a **for** ciklussal, alkalmazzuk késleltetőként. Vegyük egy változót, amelynek értékét számoltassuk el **for** ciklus segítségével 0-tól 30 000-ig. A ciklus magjába hegyezzünk el egy **nop** (*No Operation*) assembly utasítást. A **nop** utasítást a p24fj128ga010.h fejlécállományban előre definiált **Nop()** makró segítségével tudjuk beilleszteni. (Az *inline assembly használatával* a 17.5. fejezetben fogunk részletesebben foglalkozni.) A helyes működés érdekében a késleltető ciklusunkat a LED-ek kikapcsolása és bekapcsolása után is be kell illesztenünk.

12.5. mintaprogram

```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

main ( )                                // Főprogram kezdete
{
    int ido;                            // Késleltetéshez használt változó
    TRISA = 0xFF00;                     // PORTA alsó nyolc lába kimenet lesz.

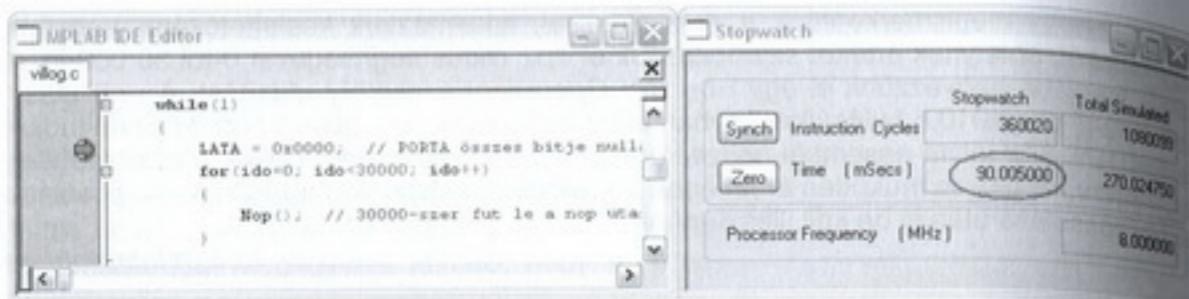
    while(1)
    {
        LATA = 0x0000;                  // PORTA összes bitje nulla lesz.
        for(ido=0; ido<30000; ido++)
        {
            // Szoftveres késleltetés:
            Nop();                      // 30000-szer fut le a nop utasítás.
        }

        LATA = 0x00FF;                  // PORTA alsó nyolc bitje egyes lesz.

        for(ido=0; ido<30000; ido++)
        {
            // Szoftveres késleltetés:
            Nop();                      // 30000-szer fut le a nop utasítás.
        }
    }
}
```

Az új programunknak, a fejlesztőpanelen lévő mikrokontrollerre letölve és ott elindítva, már szemmel látható villogást kell eredményeznie.

Milyen periódusidővel villognak a LED-ek a frissen elkészített programunkban? A válaszhoz hívjuk újból segítségül a szimulátort és a hozzá tartozó stopperórát. Helyezzünk el egy töréspontot a **LATA = 0x0000;** sor elő. A szimulátorban futassuk le idáig a programunkat, majd törljük a stopperóránkat a **Zero** gomb segítségével, és indítsuk el újból a futtatást. Amikor megint megáll a programunk futása, a stopperórán látni fogjuk, hogy mennyi idő telt el a két LED-sor be- és kikapcsolása között, azaz mennyi ideig tart egy teljes periódus. A könyv írásakor elvégzett mérés alapján durván 90 ms ideig tart egy periódus, azaz egy másodperc alatt körülbelül 11-szer villan fel a LED-sor. Érdemes megnézni a 12.8. ábra stopperóraablakában lévő számok közül az utolsó törlés után eltelt utasítások számát is, ami 360 020!



12.8. ábra
Késleltetés mérése

A játék kedvéért érdemes megnövelni vagy csökkenteni a késleltetési időt, vigyázva arra, hogy ne lépjük túl az adott változótípus számábrázolási képességeit, mert különben elég nagy meglepetések érhetnek minket. Például mit tapasztalunk akkor, ha átírjuk a két `for` ciklus feltételében szereplő számot 40000-re? ... Miért fagyott ki a programunk? ... El tudja érni egy előjeles `int` típusú változó értéke a 40000-et?

Ezenkívül érdemes még egy konfigurációs változtatást is kipróbálni. Mi történik akkor, hogy ha bekapcsoljuk az oszcillátor négyeszeres frekvenciaszorozóját. Tényleg gyorsabb lesz a végrehajtás? ... Próbaként módosítsuk a második konfigurációs szóban a konfigurációs biteket, cseréljük le az `FNOSC_PRI` beállítást `FNOSC_PRIPLL`-re. Ezzel bekapcsoltuk a négyeszeres frekvenciaszorozót.

```
_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRIPLL )
```

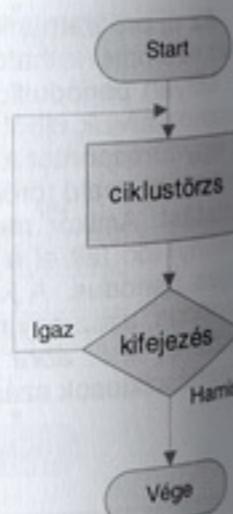
Mit tapasztalunk, ha újra letöljük, majd elindítjuk a programunkat? ... Valóban megyorsult a program végrehajtása. A négyeszeres szorzó a 8 MHz-es kvarcoszcillátorunk jeléből 32 MHz-es órajelet állított elő, így az egy másodperc alatt maximálisan végrehajtható utasítások száma 16 millióra ugrik.

12.6. DO-WHILE CIKLUS

A fejezet végén térünk vissza egy kicsit az elmélethez, és ismerkedjünk meg harmadik ciklusvezérlő utasításunkkal, a do-while ciklussal. Ez a ciklus a while ciklus testvére, csak itt a feltételvizsgálat a ciklustörzs után történik. A ciklus általános alakja és folyamatábrája (12.9. ábra) a következő:

Mint ahogy azt már említettük, a hátul tesztelő ciklusoknál a ciklustörzs biztosan lefut egyszer. Hátul tesztelő ciklusokat akkor érdemes használni, ha a feltétel olyan információtól függ, amit a ciklustörzs szerez meg. Ilyen lehet például a kilépés gomb értéke egy menükezelésben. Fontos megjegyezni, hogy ebben az esetben a while feltétel után pontosvesszöt kell tennünk, utalva arra, hogy ebben az esetben nem cikluskezdetet jelent az utasítás, hanem a do utasítás lezárását.

Nézzük meg a szokásos első tíz pozitív egész szám összegét számoló algoritmusunk megvalósítását hátul tesztelő ciklussal is (12.10. ábra).



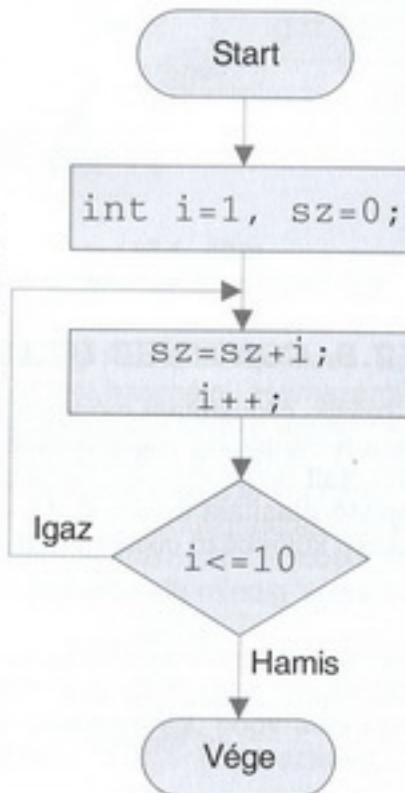
12.9. ábra.
Do-while ciklus
folyamatábrája

```
main ( )
{
    int i=1, sz=0;
    do
    {
        sz=sz+i;
        i++;
    } while(i<=10);
}
```

```
do
(
    // Végtelen ciklus
)
while(1);
```

Természetesen a do-while ciklussal is lehetőségünk nyilván a végtelen ciklus definíálására, habár ez nem igazán elterjedt megoldás. A végtelen ciklus definíálásának módszere megegyezik a while típusú ciklusnál használttal, azaz a feltételkifejezésben nullától különböző egész számot kell beírni.

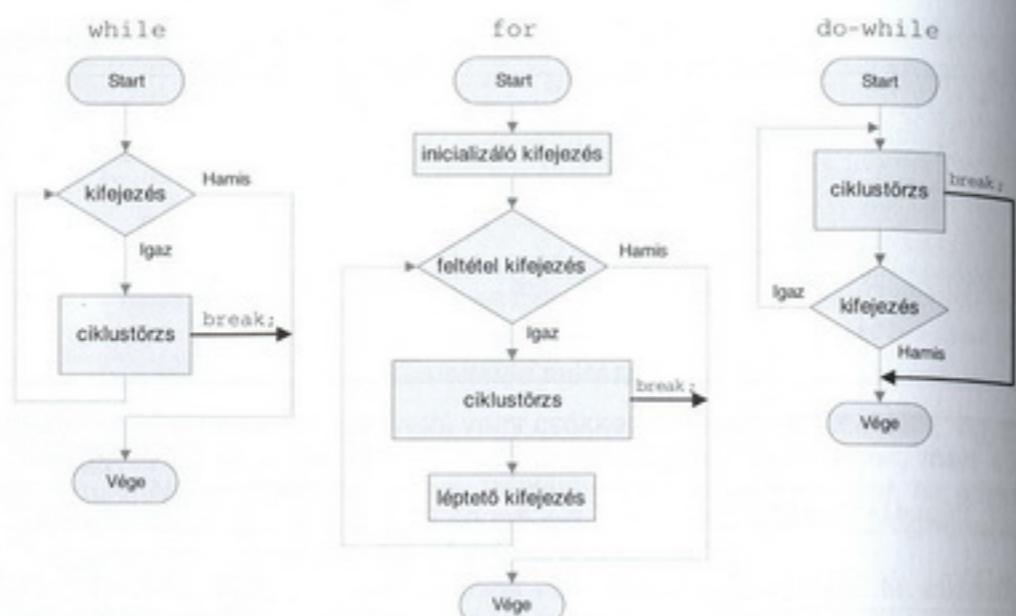
12.6. mintaprogram



12.10. ábra
Első tíz pozitív egész szám összegét kiszámoló algoritmus folyamatábrája do-while ciklussal

12.7. BREAK UTASÍTÁS

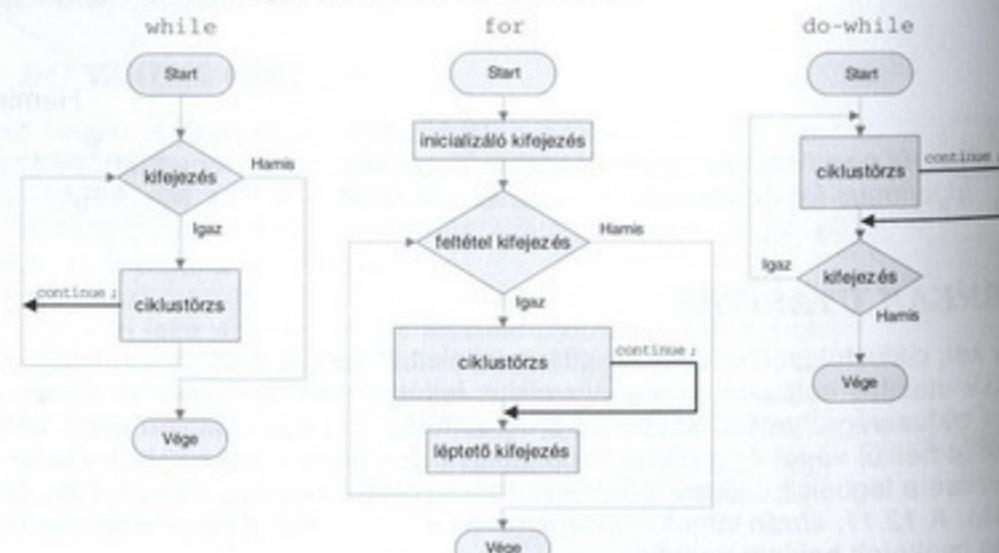
A C nyelv két, ciklusfutást módosító utasítást definiált. Az egyik ilyen a `break` utasítás. A `break` utasítás hatására az aktuális ciklus feltétel nélkül befejeződik. Ez azt jelenti, hogy ha a ciklus végrehajtása közben a programfutás `break` utasításra kerül, akkor feltételvizsgálat nélkül véget ér a ciklus feldolgozása. Egybeágazott ciklusok esetén a ciklus befejezése a legbelő ciklusra értelmezendő, azaz ilyenkor egy ciklussal feljebb lép a végrehajtás. A 12.11. ábrán látható folyamatábrák a `break` utasításnak a három különböző ciklusra gyakorolt hatását mutatják.



12.11. ábra
A break utasítás hatása a három ciklusra

12.8. CONTINUE UTASÍTÁS

A másik, ciklusfutást módosító utasítás a continue utasítás. A continue utasítás hatására a ciklusmag végrehajtása félbeszakad, és a végrehajtás újból a ciklus „fejére” ugrik, ami alatt do és do-while ciklusok esetén a feltételvizsgálatot, for ciklus esetén pedig a léptető utasítást értjük. A 12.12. ábrán látható folyamatábrák a continue utasításnak a három különböző ciklusra gyakorolt hatását mutatják.



12.12. ábra
A continue utasítás hatása a három ciklusra

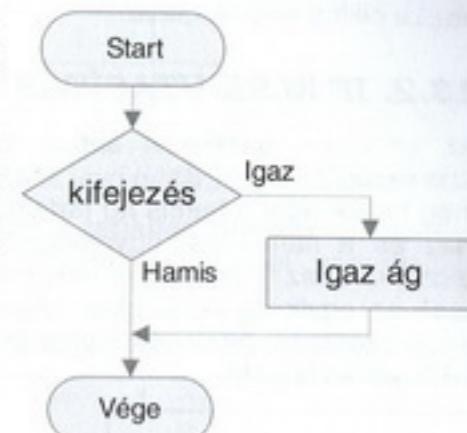
A ciklusfutást módosító utasításokat általában valamilyen feltétel alapján vagy végrehajtjuk, vagy nem. Azzal, hogy miként tudunk megadni valamilyen feltétel alapján elágazást a nyelvben, a következő fejezetben fogunk foglalkozni.

13. AMIKOR DÖNTENI KELL

13.1. IF UTASÍTÁS

Az if feltételes utasítás arra szolgál, hogy a paramétereknél szereplő feltételkifejezés kiértékelését követően a mögötte álló utasítást vagy utasításblokkot végrehajtsa vagy kihagyja. Ha a feltételként megadott kifejezés kiértékelése nullával tér vissza, azaz hamis, akkor az if utasítás után következő utasítást vagy utasításblokkot kihagyja, ellenkező esetben végrehajtja. Az if utasítás általános alakja és folyamatábrája a 13.1. ábrán látható.

```
if( kifejezés )
{
    igaz_ág;
}
```



13.1. ábra
If szerkezet folyamatábrája

Ahogy a ciklusoknál, itt is lehetőségünk van arra, hogy halmazzárójelek közé foglalt utasításblokk helyett egy utasítás szerepeljen az igaz ágban (ilyenkor nincs szükség halmazzárójelekre). Egy utasítást tartalmazó igaz ág esetén is ajánlatos halmazzárójeleket használni. Halmazzárójelek használatával elkerülhetjük azt a hibát, hogy az igaz ág későbbi bővítésekor elfelejtjük a halmazzárójeleket kirakni, így a feltételezett igaz águnk első utasítása fog csak az if utasítás feltétele alapján lefutni, míg a többi utasítás feltételelő függetlenül minden le fog futni.

Az if utasítás után, hasonlóan a 12.4. fejezetben tárgyalt while ciklusutasításhoz, ne tegyük pontosvesszöt, mert akkor az igaz ág csak egy üres utasításból áll, s ebből következik az, hogy az utána következő utasítások, feltételelő függetlenül, minden végrehajtásra kerülnek.

Az előző fejezetben megismertük a break ciklusfutást módosító utasítást, most nézünk egy példát a használatára. Használjuk arra az if utasításunkat, hogy a break utasítás segítségével befejezzén egy végtelen ciklust. A feladat legyen az, hogy addig tölgön egy ciklusban a program, amíg a ciklusváltozó értéke el nem éri a 10-es számot

13.1. mintaprogram

```
main ()
{
    int i=1;

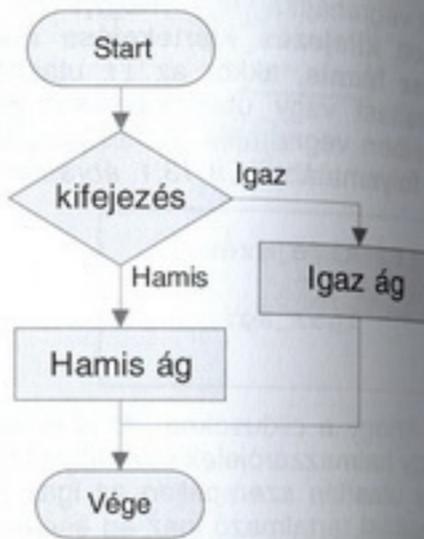
    While(1)
    {
        if( i == 10 )
        {
            break;
        }
        i++;
    }
}
```

Ha leteszteljük a programunkat a szimulátor segítségével, akkor látni fogjuk, hogy amíg az *i* változó nem veszi fel a 10-es értéket, addig az *if* utasítás nem engedi be a vezérlést az igaz ágába. Ahogy az *i* változó eléri a 10-es értéket, az *if* utasítás feltétele teljesül, így végrehajtásra kerül a *break* utasítás, aminek hatására azonnal befejeződik a *while* ciklus végrehajtása.

13.2. IF-ELSE UTASÍTÁS

Az *if-else* szerkezet abban különbözik az *if* szerkezetetől, hogy az *else* utasítás segítségével meg tudjuk adni a hamis ág tartalmát is. A két ág, az igaz és a hamis ág, úgynevezett konkurens ágak lesznek, azaz egy lefutás alkalmával biztos, hogy csak az egyik ág kerül csak végrehajtásra. Az *if-else* szerkezet általános alakja és folyamatábrája a 13.2. ábrán látható.

```
if( kifejezés )
{
    igaz_ág;
}
else
{
    hamis_ág;
}
```



13.2. ábra
Az if-else szerkezet folyamatábrája

Az *if-else* szerkezet bemutatására egy kis „partihangulatot” idéző futófényt fogunk elkészíteni. Ahhoz, hogy a futófényt elkészítsük, szükségünk van a bitoperátorok használatára is. Ezért mielőtt a programot elkészítenénk, ismerkedjünk meg a bitoperátorokkal is.

13.3. BITOPERÁTOROK

A bitoperátorok az egész számok körében értelmezett operátorok. A bitoperátorok feladata az, hogy az egész számot ábrázoló változók között bitenként végezzen el logikai műveleteket. A C nyelv a következő bitoperátorokat ismeri:

- & bitenkénti **és**
- | bitenkénti **vagy**
- ^ bitenkénti **kizáró vagy**
- ~ bitenkénti **tagadás**
- Bitenkénti eltolás (siftelés):
 - << adott bittel eltolás balra
 - >> adott bittel eltolás jobbra

A 13.3. ábra a négy alapvető bitoperátor működését foglalja össze. Az **és**, **vagy** és a **kizáró vagy** operátor kétoperandusú operátor. A bitenkénti **tagadás** operátor egyoperandusú operátor. A **tagadás** operátor eredménye az operátor jobb oldalán álló szám bitenként negáltja lesz. Az **és**, **vagy** és a **kizáró vagy** operátorok eredménye, az adott operátor jobb és bal oldalán álló két szám megegyező helyi értékű bitjeire vonatkoztatott logikai művelet eredménye lesz.

| a | b | a & b | a b | a ^ b | ~a |
|---|---|-------|-------|-------|----|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 |

13.3. ábra

Bitoperátorok eredményei ugyanazon a helyi értéken lévő bitekre vonatkoztatva

A 13.2. mintapélda a bitenkénti **és** operátor, illetve a **tagadás** operátor működését mutatja be. A mintapélda megjegyzései a sorokban található kifejezések végrehajtása közben módosuló változók értékeit mutatják.

13.2. mintaprogram

```
main ( )
{
    unsigned char a,b,c,d; // BIN | HEX
    // -----
    a = 0b01010101; // a = 0101 0101 | 0x55
    b = 0b00001111; // b = 0000 1111 | 0x0F
    // -----
    c = a & b; // c = 0000 0101 | 0x05
    // -----
    d = ~c; // d = 1111 1010 | 0xFA
}
```

A logikai és bitoperátorok szimbólumai hasonlítanak egymásra, ezért könnyen összekeverhetők. Amíg a logikai „**és**” és „**vagy**” operátorok esetén két műveleti jelet használunk (**&&**, **||**), addig a bitoperátorok esetén csak egyet (**&**, **|**). Figyeljünk minden arra, hogy melyik operátorot akarjuk használni, a két operátorcsalád egészen más eredményt ad. Amíg a bitoperátorok bitenként végzik el az adott logikai műveletet, addig a logikai operátorok eredménye 0 vagy 1 értékű egész szám lesz.

A bitenkénti eltolás operátorok az operátor bal oldalán álló kifejezés értékét a jobb oldalon álló kifejezés értékével megegyező bittel jobbra (**>>**), vagy balra (**<<**) eltolják. Az eltolás eredményeként kieső bitek elvesznek. Balra történő eltolás esetén a beérkező bitek nullák lesznek. A jobbra történő eltolás esetén az előjel nélküli (*unsigned*) típusú változók esetén a beérkező bitek 0-k lesznek, míg előjeles számok esetén (*signed*) az előjelbit lép be. (A kettes komplementus számábrázolásból következik, hogy ha az eredeti szám nulla vagy annál nagyobb volt, akkor 0, különben 1 fog belépni.) A következő mintapéldán végigkövethetők a jobbra és balra történő eltolás eredményei. A jobb átláthatóság kedvéért a belépő bitek kiemelésre kerültek.

13.3. mintaprogram

```
main ( )
{
    unsigned char a,b,c,d; // BIN | DEC
    signed char x,y,z; // -----
    a = 71; // a = 0100 0111 | 71
    b = a << 1; // b = 1000 1110 | 142
    c = b >> 1; // c = 0100 0111 | 71
}
```

```

d = c << 2;           // d = 0001 1100 | 28
// -----
x = -52;              // x = 1100 1100 | -52
y = x << 1;           // y = 1001 1000 | -104
z = y >> 1;           // z = 1100 1100 | -52
}

```

A jobbra léptetésnél belépő bitek implementációfüggőek lehetnek. Előfordulhat olyan fordító (főleg kisebb teljesítményű kontrollerekhez készített fordítók között), amely a jobbra léptetésnél is, a balra léptetéshez hasonlóan, előjeltől függetlenül nullát léptet be.

13.4. REKURZÍV ÉRTÉKADÓ OPERÁTOROK

Sok esetben szükségünk van arra, hogy egy változó értékét megnöveljük, csökkentsük, szorozzuk stb. egy számmal. Ezeket a műveleteket hívjuk rekurzív műveleteknek, mert a művelet eredménye nem egy harmadik változóban keletkezik, hanem a két bemenő változó egyikében. Ilyen művelet volt a 12. fejezetben, a ciklusok bemutatására szolgáló mintapéldákban (az első tíz pozitív egész szám összegét kiszámoló algoritmusban) szereplő `sz=sz+i;` kifejezés is. Ahogy azt már kezdjük érezni, a C nyelv nem arról szól, hogy sokat kell gépelni, ezért az ilyen típusú műveletekre is bevezettek a nyelv fejlesztői egy rövidített, rekurzív értékadó operátorcsaládot. Ezek az operátorok a következők:

| Hagyományos forma | Tömörített forma, értékadó operátorok felhasználásával |
|----------------------------|--|
| <code>a=a+b;</code> | <code>a+=b;</code> |
| <code>a=a-b;</code> | <code>a-=b;</code> |
| <code>a=a*b;</code> | <code>a*=b;</code> |
| <code>a=a/b;</code> | <code>a/=b;</code> |
| <code>a=a%b;</code> | <code>a%=b;</code> |
| <code>a=a&b;</code> | <code>a&=b;</code> |
| <code>a=a b;</code> | <code>a =b;</code> |
| <code>a=a^b;</code> | <code>a^=b;</code> |
| <code>a=a<<b;</code> | <code>a<<=b;</code> |
| <code>a=a>>b;</code> | <code>a>>=b;</code> |

13.4. ábra
Rekurzív értékadó operátorok

A rekurzív értékadó operátorok kiértékelése, hasonlóan az egyszerű értékadó operátorhoz, balról jobbra történik. Rekurzív értékadó operátorral, az előbb említett példában szereplő utasítás így írható le: `sz+=i;`

13.4.1. Egy kis „partihangulat”...

Most az if-else szerkezet bemutatásának végén beharangozott futófényt fogjuk elkezdeni. A feladat legyen a következő: Az Explorer 16 fejlesztőpanelen található mikrokontroller A portjának alsó nyolc bitjéhez csatlakozó LED-sor segítségével hozzunk létre futófényt. A futófény egyes állapotaiban mindenkor csak egy LED világítson, ami helyi érték szerint növekedjen. A legmagasabb helyi értékű LED után a legkisebb helyi értékű LED következzen.

A futófényt vezérlő programunk magját egy if-else szerkezet alkotja. Az if utasítás feltételében megvizsgáljuk, hogy jelenleg melyik LED világít. Ha a legmagasabb helyi értékű LED világít, akkor átállítjuk a legalacsonyabbra, különben egyetlen másik LED-toljuk a világító LED-et. Az aktuális LED-sorállapotot egy ledék nevezetű nyolcbites változóban tároljuk, amelynek értékét a ciklustörzs végén kirakjuk az A portra. Nézzük az előbb felvázolt elvek alapján elkészített programunkat:

13.4. mintaprogram

```

#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

main ()
{
    unsigned char ledék = 1; // futófény állapota

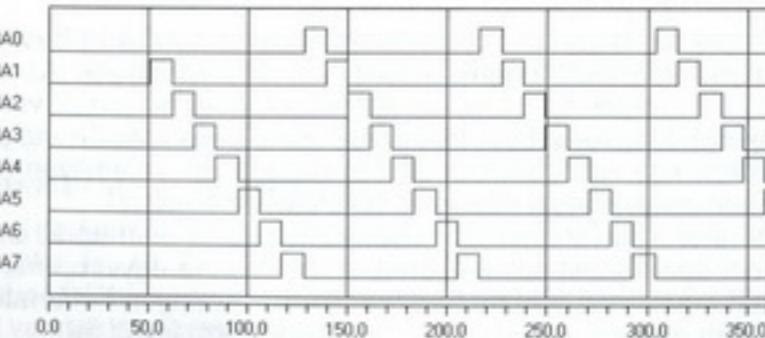
    TRISA = 0xFF00;          // PORTA alsó nyolc lába kimenet lesz.
    LATA = 0x0000;           // PORTA törlése

    while(1)                // Végtelen ciklus
    {
        if(ledék & 0x80)      // ledék változó legfelső bitje egyenlő eggyel?
        {
            ledék = 1;        // Igaz ág:
            // ledék változó legalsó bitje legyen egy,
        }
        else
        {
            ledék <<=1;       // Hamis ág:
            // ledék változó eggyel balra léptetése
        }
        LATA=ledék;           // ledék változó értékét kitessük a kimenetre
    }
}

```

Egy fontos megjegyzés még tartozik a programhoz. Az if utasítás feltételében szokatlan módon bitművelet van. Mivel egy bájt, a ledék változó, legfelső bitjének állapotáról kell információt gyűjtenünk, ezért 0x80 (binárisan 0b10000000) számmal bitenként és műveletet hajtottunk végre, más néven lemaszkoltuk. Ha a ledék változó legfelső bitjének értéke egy, akkor a maszkolás eredménye 0x80 lesz, amit az if utasítás igaznak fog tekinteni, ellenkező esetben nulla lesz, amit pedig az if utasítás hamisnak fog találni.

A szimulátorban, a logikai analizátor által megjelenített kimenetekhez adjuk hozzá az A port RA0..7 bitjeit. Futtassuk le párszor a ciklusunkat. Megjelent a futófény a kimeneten? Ha minden jól csináltunk, akkor a 13.5. ábrához hasonló kimeneti jeleket kell kapnunk.



13.5. ábra. Futófény szimulációjának eredménye

A logikai analizátor ablakában megjelenő idődiagramot a **Debugger → Processor Reset** menüpont segítségével lehet törlni. A **Debugger → MCLR Reset** hatására csak a szimulált processzorunk indul újra, a logikai analizátor nem felejt el az addigi idődiagramokat.

Most már nem maradt más dolgunk, mint beilleszteni a nemrég tanult szoftveres késleltető rutinunkat a mostani futófényprogramunkba, majd letölteni a fejlesztőpanelen lévő mikrokontrollerre. „Indulhat a parti!”

13.5. mintaprogram

```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

main ( )
{
    int ido;           // Késleltetéshez használt változó
    unsigned char ledek = 1; // futófény állapota

    TRISA = 0xFF00;      // PORTA alsó nyolc lába kimenet lesz.
    LATA = 0x0000;        // PORTA törlése

    while(1)           // Végtelen ciklus
    {
        if(ledek & 0x80) // ledek változó legfelső bitje egyenlő eggyel?
        {
            ledek = 1; // Igaz ág:
            ledek változó legalsó bitje legyen egy,
            // a többi nulla
        }
        else
        {
            // Hamis ág:
            ledek<<=1; // ledek változó eggyel balra léptetése
        }

        LATA=ledek; // ledek változó értékét kitessük a kimenetre
        for(ido=0; ido<30000; ido++)
        {
            // Szoftveres késleltetés:
            Nop(); // 30000-szer fut le a nop utasítás.
        }
    }
}
```

13.5. FELTÉTELES OPERÁTOR

Sok esetben előfordul az, hogy az if-else szerkezetet csak arra használjuk, hogy egy változónak valamelyen feltételelő függően valamelyen értéket adjunk. Az előző, futófényes példában is csak arra szolgált az if-else szerkezet, hogy a ledek változó értékét, önmaga legfelső bitjének függvényében, különböző algoritmus alapján megváltoztassuk. Az ilyen esetekre, amikor egy változónak új értéket kell adnunk valamelyen feltétel alapján, a C nyelv egy rövidített megoldással szolgál, a feltételes operátorral.

A feltételes operátor a nyelv egyetlen háromoperandusú operátora. Csak az a kérdés, hogyan lehet három operandust egy jel köré beilleszteni. Az egyik operandus az operátor jobb, a másik a bal oldalán áll, de hol áll a harmadik? Könnyen belátható, hogy egy sorban ez probléma nem oldható meg, ezért ez az operátor két jelből áll.

Az operátor általános alakja a következő:

13. fejezet: Amikor dönten kell

feltétel_kifejezés ? kifejezés_1 : kifejezés_2

Az operátor végrehajtása két lépésben történik: Először a **feltétel_kifejezés** kerül kiértékelésre. Ha a feltétel igaz, azaz nem nullával tér vissza a kifejezés, akkor

Nézzünk egy példát a feltételes kifejezés használatára. A példában négy változót deklarálunk. Az a és b változóknak kezdeti értéket adunk, míg a min és max változók értékét a programunk fogja kiszámolni úgy, hogy a min változóba az a és b változók értéke közül a kisebb, míg a max változóba a nagyobb érték fog belekerülni.

13.6. mintaprogram

```
main()
{
    int a, b, min, max; // a | b | min | max
    a = 5; // 5 | ? | ? | ?
    b = 7; // 5 | 7 | ? | ?
    min = a<b ? a : b; // 5 | 7 | 5 | ?
    max = a>b ? a : b; // 5 | 7 | 5 | 7
}
```

Következő lépésként alakítsuk át a futófényprogramunkat úgy, hogy kivesszük belőle az if-else szerkezetet, és helyette feltételes operátort használunk.

13.7. mintaprogram

```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

main ( )
{
    int ido;           // Késleltetéshez használt változó
    unsigned char ledek = 1; // futófény állapota

    TRISA = 0xFF00;      // PORTA alsó nyolc lába kimenet lesz.
    LATA = 0x0000;        // PORTA törlése

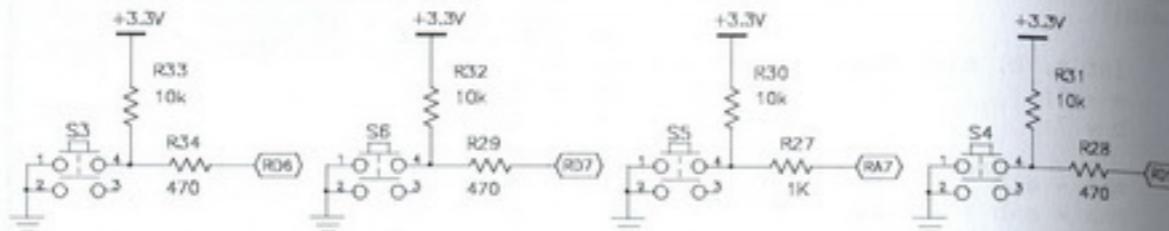
    while(1)           // Végtelen ciklus
    {
        ledek = (ledek & 0x80) ? 1 : ledek<<1; // Balra futófény 8 biten
        LATA=ledek; // ledek változó értékét kitessük a kimenetre

        for(ido=0; ido<30000; ido++)
        {
            // Szoftveres késleltetés:
            Nop(); // 30000-szer fut le a nop utasítás.
        }
    }
}
```

13.5.1. Bemeneteink is vannak

Eddig a kontroller portjainak csak a kimeneti funkcióját használtuk ki. Most ismerkedjünk meg a bemenetekkel is. Az **Explorer 16** fejlesztőszközön, közvetlenül a LED-sor alatt négy darab gombot találunk, amelyek balról jobbra haladva a kontroller **RD6**, **RD7**, **RA7** és **RD13** lábaira vannak csatlakoztatva.

A 13.6. ábra az Explorer 16 fejlesztőpanel felhasználói kézikönyvéből származik, és a négy, bemenetnek használható nyomógomb kapcsolási vázlatát tartalmazza. A kapcsolási vázlatból megfigyelhető, hogy a gombok aktív nullások. Az egyes gombok lenyomásakor, a mikrokontroller-gombokhoz csatlakozó bemeneti lábaira föld szint közelű feszültség kerül, így logikai nullát érzékel a bemenetén. Ha a gomb nincs megnyomva, akkor a kontroller adott bemeneti lábán tápfeszültség van, amit a kontroller logikai nullának érzékel. Logikai egy szintet a $10\text{ k}\Omega$ felhúzóellenállások biztosítják. Az egyes lábakkal sorba kötött $470\ \Omega$ ellenállás (RA7 lábon $1\ \text{k}\Omega$) csak védelmi funkciót látnak el. Ellentáplálás esetén (ha az adott lábat kimenetnek konfiguráljuk be) megvédi a kontroller lábait a rövidzárási áram kialakulásától. Az RA7 láb dupla funkciót lát el, bemenetként és kimenetként is használható, egy gomb és egy LED is csatlakozik a hozzá.



13.6. ábra

Az Explorer 16 négy gombjának kapcsolási vázlatá

Most készítsünk olyan futófényt, amely egy gomb lenyomásakor jobbra, felengedésekor balra fut. Az első kérdés az, hogyan tudjuk egy gomb állapotát lekérdezni. Egy láb állapotának beolvasásakor hasonló módon kell eljárni, mint láb írásakor, csak most a porthoz tartozó TRIS_x regiszternek, a port adott lábához tartozó helyi értékű bitjének értékét egyre kell állítani. A portokhoz csatlakozó érzékelők aktuális állapotát a portokhoz tartozó PORT_x regiszterek olvasásával tudjuk beolvasni. Az adott port állapotának beolvasása után más dolgunk nincs, mint a beolvasott értéket az adott bitre jellemző maszkkal bitszintű és kapcsolatba állítani. Ha a maszkolás eredménye nulla, akkor az adott lábon logikai nulla van, ha a maszkolás eredménye nullától különböző szám, akkor az adott lábon logikai egy szint található.

A C nyelv bitstruktúrák kialakítására is lehetőséget ad, amelyeket az összes SFR regiszterhez, beleértve a portregisztereket is, előre elkészítettek a C30 fordító fejlesztői. A bitstruktúrák segítségével lehetőségünk nyílik az egyes változókat bitenként is elérni. A későbbiekben mi is használni fogjuk a bitstruktúrákat, de most még nem ismerjük őket, ezért használatukat most mellőzzük. A bitmaszkolás sok esetben hasznos módszer lehet, főleg akkor, ha nincs lehetőségünk bitstruktúrák használata, mint például a hagyományos változók esetében.

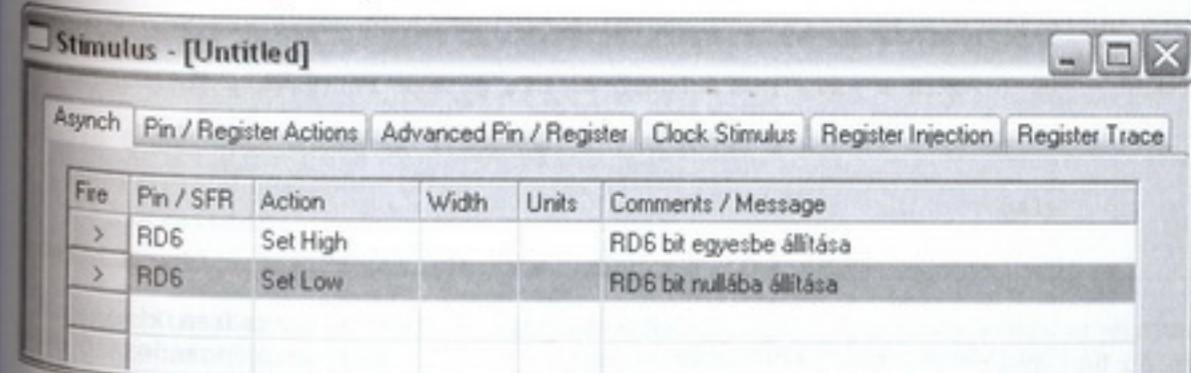
Módosítsuk az előző programunkat úgy, hogy a program minden egyes LED-léptetés előtt vizsgálja meg az RD6 láb állapotát. A láb állapotának függvényében, egy if-else szerkezet segítségével, a LED-sor állapotát ábrázoló ledék változó értékét vagy balra, vagy jobbra forgassa el. A késleltető rutint most újból szedjük ki a programból, mert először szimulátor segítségével fogjuk a programot tesztelni.

```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

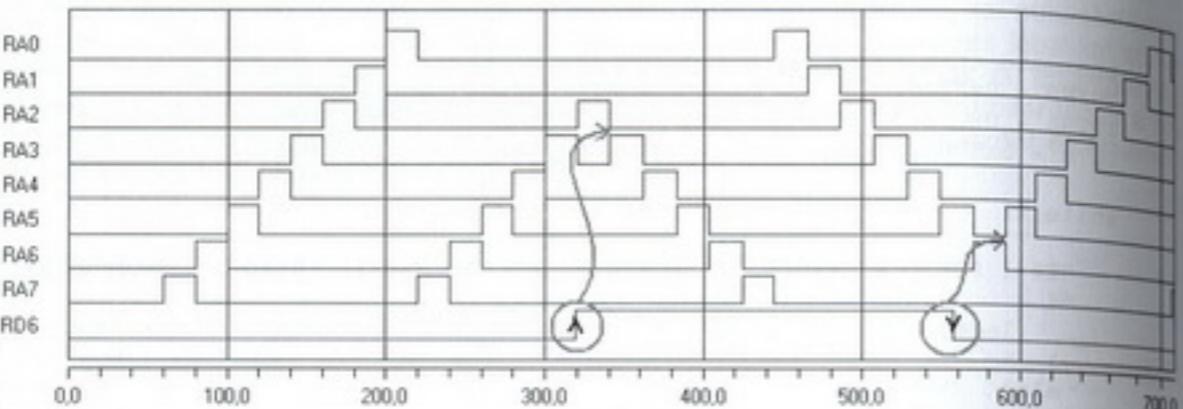
main ( )
{
    unsigned char ledék = 1; // futófény állapota
    TRISD = 0xFFFF;           // A PORTD összes lába bemenet
    TRISA = 0xFF00;           // PORTA alsó nyolc lába kimenet lesz,
    LATA = 0x0000;             // PORTA törlése

    while(1)                  // Végtelen ciklus
    {
        if ( PORTD & 0x40 ) // PORTD 6. bit értéke 1? (0x40 = 0b01000000)
        {
            ledék = (ledék & 0x80) ? 1 : ledék<<1; // Futófény 8 biten, Balra
        }
        else
        {
            ledék = (ledék & 1) ? 0x80 : ledék>>1; // Futófény 8 biten, Jobbra
        }
        LATA=ledék;           // ledék változó értékét kitesszük a kimenetre
    }
}
```

Teszteljük le a programunkat a szimulátor segítségével. Nyissuk ki újból a logikai analizátort, és az eddig figyelt bitekhez (RA0...7) adjuk hozzá RD6 bitet is. Következő lépésként válasszuk ki a **Debugger** → **Stimulus** → **New Workbook** menüpontot. A **Stimulus** programrész segítségével kontrollerre ható külső jelváltozásokat definiálhatunk a szimulátorba. Szimuláció alatt ezeket a jelváltozásokat manuálisan vagy időzítve is aktiválhatjuk. A menüpont hatására kinyíló ablakban válasszuk ki az **Asynch** lapot. Az **Asynch** lapon lévő táblázat első sorának **Pin/SFR** oszlopába válasszuk ki a **RD6** bitet, majd az **Action** oszlopba **Set High** parancsot. Ezzel a lépéssel létrehoztunk egy parancsot arra, hogy szimuláció során, felhasználói beavatkozás hatására az RD6 bitet magas szintre állítsa be. A következő sor **Pin/SFR** oszlopába újból keressük ki a legördülő menüből az **RD6** bitet, majd az **Action** oszlopba a **Set Low** parancsot válasszuk. Az új sorral az RD6 bit értékét nullára tudjuk majd állítani.

13.7. ábra
Beállított stimulus ablak

Most már ismét csak egy dolgunk maradt hátra, a szimuláció animálással történő elindítása. A szimuláció alatt kattintsunk rá a **Stimulus** ablak **Fire** oszlopában lévő $>$ jelkre, amelyekkel az előbb definiált parancsokat tudjuk aktiválni. A 13.8 ábra a szimuláció eredményét mutatja. Az ábrán megfigyelhető, hogy az RD6 láb állapotának függvényében a PORT A alsó nyolc bitjéből álló futófény különböző irányban halad.



13.8. ábra

A szimuláció eredménye az RD6 bitérték változásának függvényében

Utolsó lépésként helyezzük vissza a szoftveres késleltetést a programba. Először töltük le a programot az **Explorer 16** fejlesztőpanelen található mikrokontrollerbe, majd teszteljük le futás közben.

13.9. mintaprogram

```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

main ( )
{
    int ido;           // Késleltetéshez használt változó
    unsigned char ledek = 1; // futófény állapota
    TRISD = 0xFFFF;    // A PORTD összes lába bemenet
    TRISA = 0xFF00;    // PORTA alsó nyolc lába kimenet lesz.
    LATA = 0x0000;     // PORTA törlése

    while(1)          // Végtelen ciklus
    {
        if ( PORTD & 0x40 ) // PORTD 6. bit értéke 1? (0x40 = 0b01000000)
        {
            ledek = (ledek & 0x80) ? 1 : ledek<<1; // Balra futófény 8 biten
        }
        else
        {
            ledek = (ledek & 1) ? 0x80 : ledek>>1; // Jobbra futófény 8 biten
        }
        LATA=ledek;      // ledek változó értékét kitesszük a kimenetre
        for(ido=0; ido<30000; ido++)
        {
            // Szoftveres késleltetés:
            Nop();         // 30000-szer fut le a nop utasítás.
        }
    }
}
```

A programmal együtt elkészítettük az első játékprogramunkat is. A játékos feladata az, hogy az RD6 lábhoz csatlakozó gomb lenyomásával vagy felengedésével próbálja a világító LED-et a LED-sor közepén tartani! Jó szórakozást a játékhoz!

13.6. SWITCH-CASE SZERKEZET

Manapság talán már nincs is olyan ember, aki ügyeit intézve ne találkozott volna az automata telefonos ügyfélszolgálattal. Az ilyen típusú ügyfélszolgálatokra jellemző, hogy általában egy kellemes női hang üdvözöl minket, majd tájékoztat arról, hogy a telefonunk billentyűzete segítségével milyen menüpontokat tudunk elérni. Gondolunk bele, hogyan készítették el a menükiválasztó részt a programfejlesztők. Milyen programszerkezetben tudjuk megadni azt, hogy mit kell a programnak csinálnia, ha az egyes, kettes vagy hármás gombot nyomta meg az ügyfél. Erre a problémára megoldás lehet az **if** szerkezetek öröne, mint ahogy azt a következő példában is látjuk.

```
if( gomb == 1 )
{
    /* Az 1. menüpont lett kiválasztva */
}

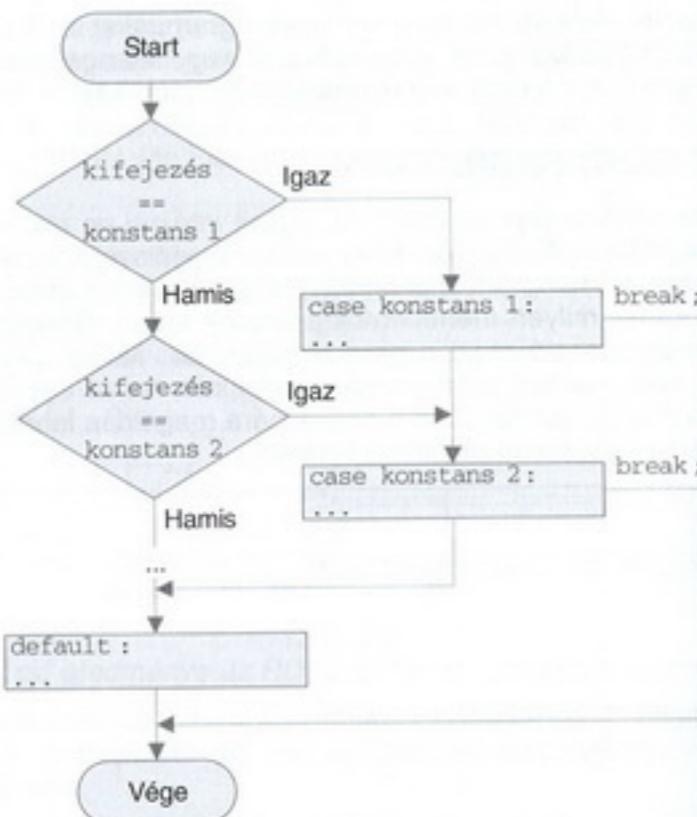
if( gomb == 2 )
{
    /* A 2. menüpont lett kiválasztva */
}

...
if( gomb == 9 )
{
    /* A 9. menüpont lett kiválasztva */
}
```

Az ilyen problémák kezelésére nyújt megoldást a **switch-case** szerkezet. A **switch-case** szerkezet általános alakja és folyamatábrája (13.9. ábra) a következő:

```
switch (kifejezés)
{
    case konstans1:
        utasítás1;
        -
    case konstans2:
        utasítás2;
        -
    default:
        utasítás;
}
```

A **switch** utasítás paramétereinek meg kell adni egy egész típusú kifejezést, amelynek értéke összehasonlítsára kerül a **switch** utasítást követő utasításblokkban használt **case** címkek konstansaival. Ha a kifejezés értéke valamelyik **case** címke utáni konstanssal megfelel, akkor a címke utáni utasításra kerül a program végrehajtása.



13.9. ábra

A switch-case szerkezet folyamatábrája

Abban az esetben, ha a switch utasítás paraméterében szereplő kifejezés értéke egyik case címke utáni konstanssal sem egyezik meg, akkor default címkére ugrik a vezérlés. Ha az utasításblokk nem tartalmaz default címkét, és nincsen case egyezés sem, akkor a vezérlés az utasításblokk után következő első utasításra kerül.

Abban az esetben, ha az egyik case címkénél belép a végrehajtás az utasításblokkba, akkor a belépési pont után következő case címek által jelzett utasítások is végrehajtássra kerülnek. Ha nem ez a szándékunk, akkor minden case címke által jelölt utasítássorozatot break; utasítással kell lezárnunk, amelynek hatására a vezérlés a switch blokk után következő első utasításra kerül.

A telefonos példa megvalósítása switch-case szerkezzel a következőképpen alakult:

```

switch( gomb )
{
    case 1:
        /* Az 1. menüpont lett kiválasztva */
        // Ha nincs break; akkor a 2. menüponttal folytatódik a végrehajtás
    case 2:
        /* A 2. menüpont lett kiválasztva */
        break;
    case 3:
    case 4:
        /* A 3. és 4. menüpont együtt */
        break;
}
  
```

```

    ...
    default:
        /* Nincs ilyen menüpont */
    }
  
```

A fejezet végén vessük be az eddigi tudásunkat. Készítünk egy olyan demóprogramot, amelyben egyesítjük a jobbra és balra futófényt, valamint a teljes LED-sorvillogást. A programunk a következőket tudja majd:

- Ha az RD6 lábon lévő gombot tartjuk lenyomva, akkor a LED-sorban balra fog futni a futófény.
- Ha az RD7 lábon lévő gombot tartjuk lenyomva, akkor a LED-sorban jobbra fog futni a futófény.
- Ha az RD6 és az RD7 lábon lévő gombokat egyszerre tartjuk lenyomva, akkor a teljes ledsor villogni fog.
- Ha egyik gombot sem tartjuk lenyomva, akkor a teljes LED-sor kialszik.

A programunk gerincét egy switch-case szerkezet adja. minden ciklus elején a program beolvassa az RD6 és az RD7 gombok állapotát, majd a gombkombinációból keletkező bináris szám (0-3) alapján az előbb felsorolt feladatok közül választ. Azért, hogy a futófény és a villogás ne zavarja egymást, az állapotukat más-más változóban tároljuk el. Vizsgáljuk meg az előbb említett feltételek alapján elkészült programunk forráskódját:

13.10. mintaprogram

```

#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

main ( )
{
    int ido;                                // Késleltetéshez használt változó
    unsigned int gomb;                        // Lenyomott gomb állapot
    unsigned char ledek = 1;                  // Futófény állapota
    unsigned char villog = 0;                 // Villogás állapota

    TRISD = 0xFFFF;                           // A PORTD összes lába bemenet
    TRISA = 0xFF00;                           // PORTA alsó nyolc lába kimenet lesz.
    LATA = 0x0000;                            // PORTA törlese

    while(1)                                  // Végtelen ciklus
    {
        gomb = ( ~PORTD & 0b11000000 ) >> 6;
        switch(gomb)
        {
            case 1:                         // Balra futófény 8 biten
                ledek = (ledek & 0x80) ? 1 : ledek<<1;
                LATA = ledek;               // A ledek változó értéke kimenetre kerül
                break;
            case 2:                         // Jobbra futófény 8 biten
                ledek = (ledek & 1) ? 0x80 : ledek>>1;
                LATA = ledek;               // A ledek változó értéke kimenetre kerül
                break;
            case 3:                         // LEDsor villogás
                villog = ~villog;           // villog változó bitenként negálása
                LATA = villog;              // A villog változó értéke kimenetre kerül
                break;
        }
    }
}
  
```

```

default:
    LATA = 0;           // Összes LED kikapcsolva
)
for(ido=0; ido<30000; ido++)
{
    Nop();             // Szoftveres késleltetés:
    // 30000-szer fut le a nop utasítás,
}
)

```

A program az eddigi kifejezésekhez képest egy kicsit komplexebb kifejezést is tartalmaz, méghozzá a gomb változó értékének kiszámolását. Nézzük meg az előbb említett utasítás részletes kifejtését:

```

gomb = (~PORTD & 0b11000000) >> 6;
/*
               +--RD7
               |+-RD6
PORTD = xxxx xxxx xx00 xxxx
Aktív nullás gombértékek megfordítása: ~PORTD = XXXX XXXX XXXX XXXX
Csak az RD6 és RD7 bitek megtartása:
0b11000000 = 0000 0000 1100 0000
~PORTD & 0b11000000 = 0000 0000 XX00 0000
Az RD6 és RD7 bitek alsó két helyi értékre mozgatása eltolással:
(~PORTD & 0b11000000) >> 6 = 0000 0000 00000000xx */

```

13.6.1. Csak haladóknak...

A fejezet lezárásaként beszéljünk az **A port** kimeneteinek frissítéséről. A 16 bites kontrollereknél a portok is 16 bites szóhosszúságúak. A mostani példában az **A port** alsó 8 bitjét használtuk csak ki. Mi történik a felső 8 bittel? Jelenleg a programunk nullákat írt a felső 8 bitbe. Most ezzel nem követünk el hibát, mert a **TRISA** regiszter segítségével csak az alsó 8 bitet állítottuk kimenetnek, a felső 8 bitet bemenetnek (magas impedanciájúnak) definiáltuk. Abban az esetben, ha a felső 8 bit közül is használunk kimeneti lábakat, akkor ilyen ponyolásigot nem engedhetünk meg. Ilyen esetekben a **LATx** regiszter írása előtt először be kell olvasni az eredeti **LATx** regiszter állapotát, majd módosítani kell a kívánt biteket, és legvégül vissza kell írni a teljes **LATx** regiszter értékét. Nézzük, hogy lehet a mostani programunkat úgy módosítani, hogy az **A porton csak az alsó 8 bit frissüljön**. Ha maszkolásos portfrissítés módszert választunk, akkor a típuskonverziós hibák elkerülése érdekében érdemes a ledék változót is 16 bitesnek (**unsigned int** típusúnak) választani.

```

LATA = ( LATA & 0xFF00 ) | ( ledék & 0x00FF );
/*
               LATA = xxxx xxxx xxxx xxxx
               0xFF00 = 1111 1111 0000 0000
               LATA & 0xFF00 = xxxx xxxx 0000 0000
               ledék & 0x00FF = 0000 0000 XXXX XXXX
               ( LATA & 0xFF00 ) | ( ledék & 0x00FF ) = xxxx xxxx XXXX XXXX */

```

Ha az A PORT felső nyolc bitjére szeretnénk a LED-ek változó alsó nyolc bitjének értékét kitenni, és a port alsó nyolc bitjét érintetlenül hagyni:

```

LATA = ( LATA & 0x00FF ) | ( ledék << 8 );
/*
               LATA = xxxx xxxx xxxx xxxx
               0x00FF = 0000 0000 1111 1111
               LATA & 0x00FF = 0000 0000 xxxx xxxx
               ledék << 8 = XXXX XXXX 0000 0000
               ( LATA & 0x00FF ) | ( ledék << 8 ) = XXXX XXXX xxxx xxxx */

```

14. VISSZA A VÁLTOZÓKHOZ

14.1. TÍPUSKONVERZIÓ

A 11. fejezetben megismert változók külön-külön szépen használhatók. De mi van abban az esetben, ha például egy **int** típusú változót össze kell szorozni egy **float** típusúval? Milyen típusú lesz az eredmény?

A C nyelv kétfajta típuskonverziót ismer. Ha nem használunk típuskonverziós operátort, akkor **implicit** vagy más néven **automatikus konverzióval** dolgozik a fordító. Ha két különböző típusú változó között kell műveletet végezni, akkor **implicit** konverzió esetén a fordító a „nagyobb” típusú változó körében végzi el a műveletet, azaz a „kisebb” típust „nagyobba” konvertálja. Természetesen az eredeti változó értéke nem változik meg, a konvertálás eredménye egy átmeneti változóba kerül. A változók „nagyságbeli” sorrendjét a bitosszúságuk határozza meg, figyelembe véve, hogy a lebegőpontos számok nagyobbak, mint az egész típusú változók. Az előbbi szabály alapján a változók növekvő sorrendje a következőképpen alakul:

```
char -> short -> int -> long -> long long -> float -> double -> long double
```

Az értékkadó operátorok esetén az operátor bal oldalán álló változó típusává fogja átkonvertálni a jobb oldalán álló kifejezés eredményének típusát.

Típuskonverzió esetén figyeljünk arra, hogy konvertálás esetén értékvesztés is előfordulhat. Például ha lebegőpontos számokat egész számmá konvertálunk, akkor a tizedes pont utáni értékek elvesznek. Nagyobb bitszámú változó kisebb bitszámúvá konvertálása esetén pedig a számábrázolásból adódó túlcordulási problémák léphetnek fel.

Nézzük egy példát az implicit típuskonverzió használatára:

14.1. mintaprogram

```

main ( )
{
    float a = 3.3;
    int b = 3;
    double c;
    long d;

    if ( a > b )
    {
        c = a;
    }
    else
    {
        c = b;
    }
    d = c;
}

```

Az if feltétele float típuskörben kerül kiértékelésre, mert a reláció jobb és bal oldalán álló két típus közül a float a nagyobb. Az elágazás mindenkor ágában, az értékadásnál, a jobb oldalon álló változó értékét double-ra konvertálja a fordító, mert az értékadás operátor jobb oldalán double típusú változó található. A d = c; utasítás végrehajtásakor adatvesztés történik, a c változó tizedes pont utáni értéke nem kerül át a d változóba, mert a d változó egész típusú számokat tud csak ábrázolni. A 14.1. ábra a program futásának végén lévő változóértékeket mutatja.

Nézzünk egy következő példát is. A kérdés az, hogy mennyi lesz a c változó értéke az osztás művelete után?

```
main ( )
{
    int a=5, b=2;
    float c;

    c = a / b;
}
```

A program tesztelése nem várt eredményt hoz (14.2. ábra). Hiába lebegőponos típusú a c változó, az osztás után a változó értéke 2.5 helyett 2.0 lesz. Hová tűnik el a c változó tizedes része?

Az implicit konverzió az operátorok közvetlen közelében történik. Az osztás operátor mindenkor oldalán int típusú változó áll, ezért az osztás az egész számok közötti történik. Az osztás eredményét, ami 2, az értékadó operátor float típusúvá konvertálja, így 2.0 lesz, majd ez az érték kerül bele a c változóba.

Hogyan lehet megoldani a változók típusának megváltoztatása nélkül azt, hogy az osztás eredménye helyes érték legyen? Erre találták ki a nyelv fejlesztői az explicit, más néven kényszerített konverziót és a hozzá tartozó típuskonverziós operátorot.

A típuskonverziós operátorral történő típuskényszerítés általános alakja a következő:

(típus) kifejezés

A típuskonverziós operátor hatására a típuskonverziós operátor jobb oldalán álló kifejezést értékét a zárójelek közötti típusára konvertálja, és az így kapott értéket adja át többi feldolgozásra.

Nézzük újból az előző mintapéldánkat, de most már típuskonverziós operátorok használatával:

| Add SFR | AD1CHS | Add Symbol | SP |
|---------|---------|-------------|----------|
| Update | Address | Symbol Name | Value |
| | 0810 | a | 3.300000 |
| | 080E | b | 2 |
| | 080A | c | 3 |
| | 0806 | d | 3 |

14.1. ábra

Implicit típuskonverzió mintapélda-szimulációjának eredménye

14.2. mintaprogram

| Add SFR | AD1CHS | Add Symbol | SP |
|---------|---------|-------------|----------|
| Update | Address | Symbol Name | Value |
| | 080C | a | 5 |
| | 080A | b | 2 |
| | 0806 | c | 2.000000 |

14.2. ábra
Tizedes értékek elvesztése

```
main ( )
{
    int a=5, b=2;
    float c;

    c = (float)a / (float)b;
```

A módosított programunkban az osztás operátor már lebegőpontos számok között dolgozik, mert az osztás előtt a két típuskonverziós operátor az osztás mindenkor oldalán álló változót float típusára konvertálja. Ebben az esetben már nem vesz el az osztás tizedes része sem, így a c változó értéke 2.5 lesz.

| Add SFR | AD1CHS | Add Symbol | SP |
|---------|---------|-------------|----------|
| Update | Address | Symbol Name | Value |
| | 080C | a | 5 |
| | 080A | b | 2 |
| | 0806 | c | 2.500000 |

14.3. ábra

Kikenyszerített típuskonverzió eredménye

Az előző példában elég lett volna csak az egyik változó előre kitenni a típuskonverziós operárt: c = (float)a / b; Ilyen esetben a másik változó már automatikusan átkonvertálódik az automatikus típuskonverzió segítségével, mert az osztás operátor jobb és bal oldalán nem ugyanolyan típusú változó áll.

Mi történik akkor, ha az előző példában a c változónak a következőképpen adunk kezdő értéket?

float c = 3/2;

A program tesztelése közben újból meglepetés ér minket. A c változó 1.5 helyet 1.0 értéket vesz fel. Ennek a jelenségnek az a magyarázata, hogy a nyelv a konstansok közötti műveleteket futási időben számolja ki. A 3/2 kiszámításához két egész típusú átmenneti változót hoz létre, majd az osztás eredményét (ami szintén egész típusú lesz) átkonvertálja float típusára, és a c változó értékét feltölti vele. Ilyen esetekben a változó kezdőértékét a következőképpen adhatjuk meg:

float c = 3.0/2.0;

Most már a fordító a két konstans számot lebegőpontos számkként fogja kezelní, így az osztás eredménye helyes lesz.

Ahogy láttuk, a C nyelvben a konstansok közötti műveletek kiszámítása futási időben történik, ezért az előbbi értékadás felesleges processzoridő- és programkód-növekedést okoz, ezért kerüljük az ilyen értékadásokat! Helyette a kiszámított értékkel tegyük egyenlővé a változót:

float c = 1.5;

Az egész konstans számokat int típusúnak tekinti a fordító. Ha azt szeretnénk, hogy a konstans értékünket a fordító long típusú egész számnak tekintse, akkor a szám mögé egy „L” betűt kell írni.

```
long nagy_szam = 1000000L;
```

14.1.1. Magyar jelölés

Hogy tudhatjuk meg programírás közben egy változóról, hogy milyen típusú? Meg kell keresnünk a változó deklarálásának helyét, és ott meg kell nézni. Nem lehetne becsempészni valahogyan a változó nevében a típusát is? Valami hasonló kérdést tehetett fel Simonyi Károly (Charles Simonyi) akkor, amikor megalkotta a Xerox PARC-ban a változók jelölésrendszeri ajánlását. Ez mára világszerte elterjedt és használt jelölésrendszer a változók jelölésére. A jelölésrendszer később nevet is kapott, „**Hungarian notation**” névvel terjedt el a számítástechnikai szakirodalomban.

A jelölésrendszernek több változata is elterjedt, amelyeket a programozók általában a saját szájuk íze szerint használnak. Az idő során nagyon sok változat (nyelvjárás) született. A jelölésrendszer alapszabályainak alapjait így lehet összefoglalni:

- 1) A változó a típusára jellemző kisbetű rövidítéssel kezdődjön. A rövidítés általában a típus kezdőbetűjéből alakul ki.
- 2) A változó a feladatát jellemző névvel folytatódjon úgy, hogy a szavak nagybetükkel kezdődjenek, a többi betű kicsi legyen.

Például egy kimeneti értékre használt, előjel nélküli integer típusú változó neve ilyen formában alakul:

```
unsigend int wKimenetiErtek;
```

A következő felsorolás, a teljesség igénye nélkül, pár lehetséges rövidítést tartalmaz:

- f: bit (flag);
- ch: karakter (char);
- b: bajt (szokás bitnek is használni, ebben az esetben az f-et, float-nak lehet használni);
- w: word (előjel nélküli 2 bajtos típus);
- dw: double word (előjel nélküli 4 bajtos típus);
- i: integer;
- l: long;
- db: double;
- p: mutató;
- sz: karakterlánc.

A könyv hátralévő mintapéldáiban mi is ezt a jelölésrendszert fogjuk alkalmazni. Az előző példákban azért nem használtuk a jelölésrendszert, mert nem szerettük volna, hogy túl nagyok legyenek a változóneveink. A túl hosszú nevek egy kezdő programozó számára akár zavaróak is lehetnek. Aki részletesebben meg szeretné ismerni a jelölésrendszert, az a Microsoft MSDN honlapján elolvashatja az eredeti ajánlást. ([http://msdn.microsoft.com/en-us/library/aa260976\(VS.60\).aspx](http://msdn.microsoft.com/en-us/library/aa260976(VS.60).aspx))

14.2. MUTATÓK

Aki már programozott assembly nyelven, az találkozott a processzorok által nyújtott indirekt címzési lehetőséggel. A PIC 16 bites kontrollerei is ismerik az indirekt címzést. Az assemblyben a szögletes zárójelek [] segítségével érhetjük el a mikrokontroller indirekt címzési szolgáltatását.

Mit jelent az indirekció? Indirekcióról akkor beszélünk, amikor egy változóban nem egy értéket tárolunk, hanem egy memóriacímet, ami valamilyen másik változóra mutat.

A C nyelv típusos nyelv. A nyelv e tulajdonsága a mutatótípusokra is igaz. A mutatók esetén is meg kell adnunk, hogy milyen típusú változóra fog mutatni a frissen deklarált mutatóink. Deklaráláskor a fordítónak a változó neve előtt megjelenő **csillaggal** * jelezhetjük azt, hogy most mutatót szeretnénk létrehozni. Egy mutató deklarálásának általános alakja a következő:

```
mutató_típusa *mutató_neve;
```

Például ha egy int típusú változóra szeretnénk egy mutat nevű mutatót létrehozni, akkor azt így tehetjük meg:

```
int *mutat;
```

Mutatók esetén is figyeljünk arra, hogy a deklarálásukkor ugyanúgy, mint a hagyományos változódeklaráció esetén, nincs meghatározva a kezdőértékük.

A mutató talán a „legélesebb fegyvere” a C nyelvnek. A rutinos programozó nagyon hatékony programokat tud készíteni a mutatók segítségével, de egy kezdő programozó számára öngyilkos eszköz is lehet. A mutatók segítségével korlátozások nélkül elérhető a teljes memória, így könnyen elkövethetünk olyan hibákat, hogy más változó által használt memóriaterületek értékét megváltoztatjuk, ami a programfutásunkat kiszámlíthatatlanná teszi.

A mutatókhoz két operátor tartozik szorosan. Az egyik operátor az **egyoperandusú & (és) operátor**, más néven **címképző operátor**. A címképző operátor az operátor jobb oldalán álló változó memóriacímével tér vissza. Például ha azt szeretnénk, hogy a mutat nevű mutatót a valtozo nevű változóra mutasson, azt a következőképpen tehetjük meg:

```
mutat = &valtozo;
```

Az & (és) jelnek, annak függvényében, hogy milyen környezetben szerepel, két különböző jelentése van. Ha változó előtt használjuk mint egyoperandusú operátort, akkor a változó címével tér vissza, ha kétoperandusú operátorként használjuk, akkor pedig bitenkénti „és” műveletet végez. Nem beszélve arról, ha két „és” jelet (&&) teszünk ki, mert ebben az esetben logikai „és” műveletet hajt végre.

A másik, a mutatókkal szorosan összefüggő operátor az **egyoperandusú csillag** *, más néven **indirekciós operátor**. Az indirekciós operátor a mutató által címzett memóriaterületre mutat. Például ha a mutat nevű mutató által mutatott memóriaterületre szeretnénk egy értéket beírni, azt a következőképpen tehetjük meg:

```
*mutat = 5;
```

A csillag jelnek, annak függvényében, hogy milyen környezetben szerepel, három különböző jelentése van. Ha változó deklarációjakor használjuk a csillag jelet, akkor mutató típusú változó hozunk létre. Egyoperandusú csillag operátor segítségével memóriaterületet címzünk meg, míg a kétoperandusú csillag operátor a matematikai szorzás műveletét jelenti.

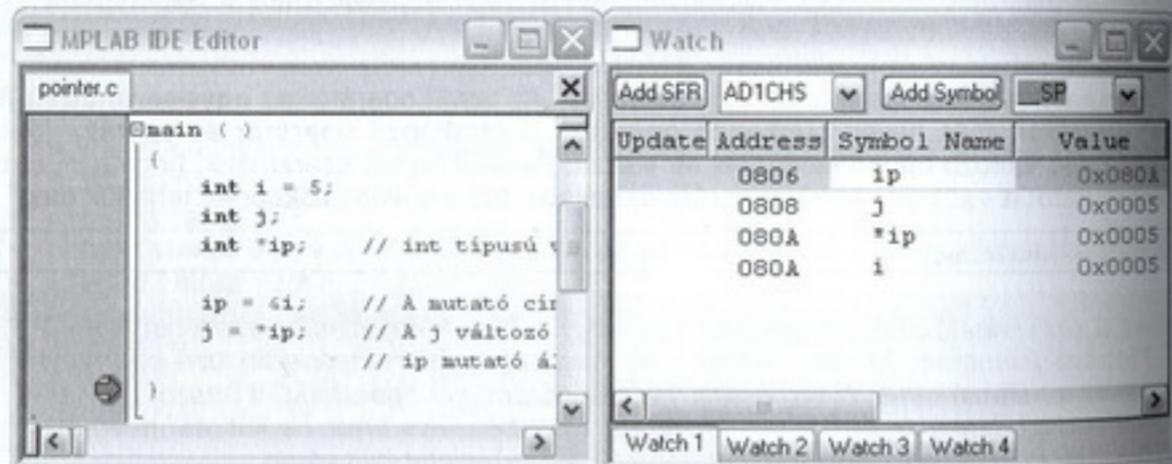
Érdemes megfigyelni, hogy ha egy mutató típusú változót **indirekciós operátor** (*) nélkül használunk, akkor a mutató által tartalmazott memóriacímet módosíthatjuk vagy olvashatjuk ki, míg az indirekciós operátorral használva a mutató által megcímezett memóriaterület értékét írhatjuk vagy olvashatjuk. Nézzünk egy példát a mutatók használatára:

14.4. mintaprogram

```
main ( )
{
    int i = 5;
    int j;
    int *pi; // int típusú változóra mutató

    pi = &i; // A mutató címe legyen egyenlő az i változó címével.
    j = *pi; // A j változó értéke legyen egyenlő az
              // ip mutató által mutatott memóriacímen lévő értékkel.
}
```

A programban három változót deklaráltam. Az i és j változók egyszerű int típusú változók, az *ip változó egy int típusú mutató. A ip = &i; utasítás az ip mutató értékét az i változó címével tölti fel. A j = *ip; utasítás a j változó értékét egyenlővé teszi az ip mutató által mutatott memóriaterületen lévő értékkel, azaz i változó értékével.



14.4. ábra

Mutatók mintapélda-szimulációjának eredménye

A szimulációban látható, hogy az ip mutató értéke 0x080A lett, vagyis az i változó memóriacíme, így a *ip tényleg az i változóra mutat.

Létezik egy különleges mutatótípus a nyelvben, a void* típus, azaz a határozatlan vagy általános típusú mutató. A void* típusú mutató nem tipizált memóriaterületre mutat. A void* mutatóval nem lehet műveleteket végezni, de szabadon átkonvertálható más típusú mutatóvá. Ezt a mutatótípust átmeneti mutatóként érdemes használni, mint ahogyan például a dinamikus memóriakezelés (lásd 14.9. fejezet) függvényei használják.

Az int, long vagy akár a double típusú változóra mutató változó memóriaigénye megegyezik, mert a memóriaigényét az adott architektúra által használt címzés memóriaigénye határozza meg. A PIC 16 bites mikrokontrollerek adatmemóriájának címbusa 16 bites, ezért a mutató típusú változók memóriaigénye 2 bájt.

14.3. TÖMBÖK

Sok esetben nemcsak egy változóra van szükségünk, hanem ugyanabból a változóból több elemet is el kell tárolnunk a memóriába, mint például mérési adatgyűjtés esetén. Ilyen problémák megoldására alkalmazzuk a tömböket. A C nyelv lehetőséget biztosít a tömbök létrehozására. Tömböt bármilyen változótípusból készíthetünk. A tömbök deklarációjának általános alakja a következő:

típus név[elemszám];

Például, ha egy ötelelmű, int típusú tömböt szeretnénk deklarálni, azt a következőképpen tehetjük meg:

int tomb[5];

Mivel az int típusú változó memóriaigénye 2 bájt, ezért az 5 elemből álló int típusú tömb memóriaigénye $5 \times 2 = 10$ bájt.

A tömbök létrehozása statikus módszerrel történik, ezért fordítási időben már tudnia kell a fordítónak, hogy hány elemük lesznek a tömbjeink. Ebből következik, hogy az elemszám csak **konstans** érték lehet. Mivel a későbbiekben sem lehet a tömb elemszámát megváltoztatni, ezért a tömb deklarálásakor úgy kell meghatározni a tömb nagyságát, hogy a várható maximális felhasználás esetén is elég legyen.

A tömb deklarálása után a tömb egyes elemeit a **tömbelem-kijelölő operátor** [] segítségével érhetjük el. A tömb elemeinek indexelése 0-tól kezdődik, és a (tömb elemszáma-1)-ig tart, például az 5 elemű tömb egyes elemeit 0–4 indexszámokkal érhetjük el. Tömbök használatakor figyeljünk arra, hogy elemkiválasztáskor **ne lépjünk ki a tömbből**, (elemszám-1)-nél nagyobb indexszámmal ne címezük meg a tömbelemeinket, mert ilyen esetekben más változók értékét írhatjuk felül, ami hibás programfutáshoz vezet!

```
tomb[2] = 3; // A tömb második indexű elemének értéke
              legyen egyenlő hárommal.
a = tomb[3]; // Az a változó értéke legyen egyenlő
              a tömb hármás indexű elemének értékével.
tomb[5] = 0; // FUTÁSI HIBA !!! Nincs ötös indexű elem!
```

Tömb deklarálásakor lehetőségünk van kezdőérték adására. A tömb deklarálásakor a tömb egyes elemeinek kezdőértékét halmazzárójelek között, vesszővel elválasztva adhatjuk meg.

int tomb[5] = { 1, 2, 3, 4, 5 };

Ha több elemet adunk meg deklaráláskor, mint a tömb elemszáma, akkor fordítási hiba következik be. Kevesebb elem esetén csak a tömb eleje töltődik fel. Lehetőségünk van arra is, hogy kezdőérték adása esetén nem adjuk meg a tömb elemszámát, ilyen esetben a tömb elemszámának értéke a kezdőértékként megadott lista elemszámával fog megfelelni.

int tomb[] = { 1, 2, 3, 4, 5 };

14.3.1. Jöjjenek újból a LED-ek...

Nézzünk egy mintapéldát a tömbökkel kapcsolatban. A példához használjuk újból az Explorer 16 fejlesztőpanelen található LED-sort. Az új programunk a LED-eket úgy fogja be- és kikapcsolni, mintha egy zenedobozban lévő henger mintázatát követné. A „henger domborzatának mintáját” egy tömbben fogjuk eltárolni, amit a főprogramunk körbe-körbe fog majd „lejátszani”.

14.5. mintaprogram

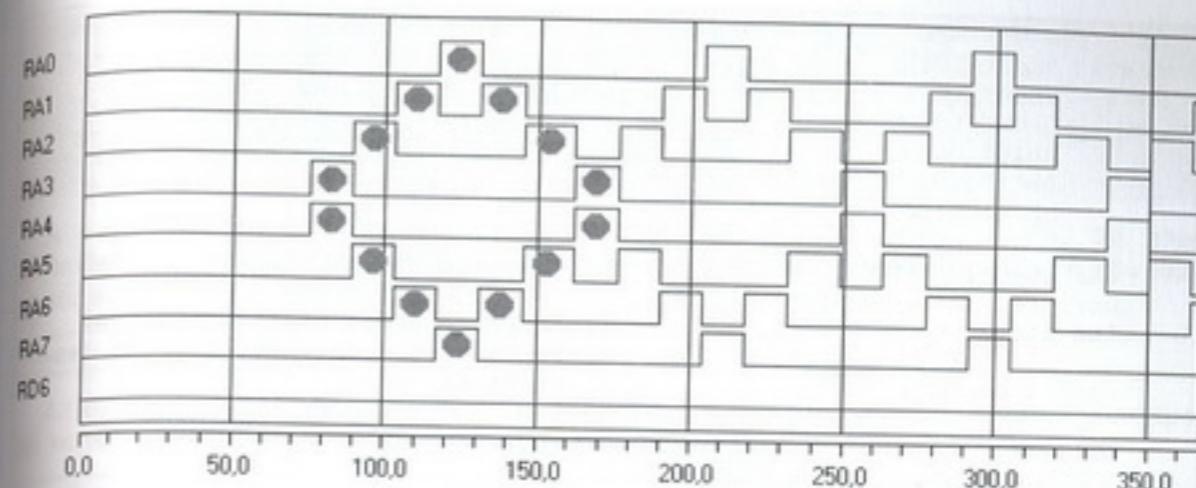
```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

main ( )
{
    unsigned char chLedKep[] = // A ledmintát tartalmazó táblázat
    {
        0b00011000,
        0b00100100,
        0b01000010,
        0b10000001,
        0b01000010,
        0b00100100
    };
    unsigned int wIndex=0; // Táblázat elemkijelölő indexe
    unsigned int wTombElemszam = 6; // Táblázat elemszama

    TRISA = 0xFF00; // PORTA alsó nyolc lába kimenet lesz.
    LATA = 0x0000; // PORTA törlése

    while(1) // Végtelen ciklus
    {
        LATA = chLedKep[wIndex++]; // ledkép kihelyezése A ledsortra
        if (wIndex >= wTombElemszam)
        {
            // A tömb indexmutatójának törlése
            wIndex = 0; // túlcsordulás esetén
        }
    }
}
```

Szimulátor segítségével futtassuk le a programot. Nyissuk ki a logikai analizátort, és adjuk hozzá a megjelenítendő portlábakhoz az A port alsó nyolc bitjét. Megjelent a sarkára állított négyzet?



14.5. ábra

Tárolt minta megjelenése a LED-soron

Érdemes a **watch** ablakban a tömbünket is hozzáadni a listához, vagy kinyitni a **View** → **Locals** menüpont alatt található **locals** ablakot (az ablakban az összes lokális változó megjelenik). Mind a két ablakban kinyitható a tömbváltozó, és az egyes elemek értéke egyenként megtekinthető.

| Locals | | |
|---------|---------------|----------|
| Address | Symbol Name | Value |
| 080A | chLedKep | |
| 080A | [0] | 00011000 |
| 080B | [1] | 00100100 |
| 080C | [2] | 01000010 |
| 080D | [3] | 10000001 |
| 080E | [4] | 01000010 |
| 080F | [5] | 00100100 |
| 0808 | wIndex | 0x0003 |
| 0806 | wTombElemszam | 0x0006 |

14.6. ábra

A chLedKep tömb elemei a Locals ablakban

Most helyezzük vissza a már megszokott késleltető rutinunkat a programba, de most az időkorlátot 2000-re csökkentsük. Töltsük le a programunkat a mikrokontrollerbe és indítsuk el. Programunk tesztelését az eddigiek töl eltérő módon érdemes elvégezni. A teszteléshez fogjuk a kezünkbe a fejlesztőpanelt, és kezdjük a szemünk előtt jobbra-balra mozgatni. A mozgatást olyan gyorsan végezzük, hogy a szemünk előtt is megjelenjen a szimulátor képernyőjén már látott alakzat. A programot érdemes sötétben kipróbálni, mert a LED-fényereje nem olyan erős, hogy nappali fény mellett is megjelenjen a kivetített kép.

14.6. mintaprogram

```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

main ( )
{
    unsigned char chLedKep[] = // A ledmintát tartalmazó táblázat
```

```

{
    0b00011000,
    0b00100100,
    0b01000010,
    0b10000001,
    0b01000010,
    0b00100100
};

unsigned int wIndex=0;           // Táblázat elemkijelölő indexe
unsigned int wTombElemszam = 6;  // Táblázat elemszama
int iido;                      // Késleltetéshez használt változó
TRISA = 0xFF00;                // PORTA alsó nyolc lába kimenet lesz.
LATA = 0x0000;                  // PORTA törlése

while(1)                         // Végtelen ciklus
{
    LATA = chLedKep[wIndex++];   // ledkép kihelyezése A ledsorra
    if (wIndex >= wTombElemszam)
    {
        . . .                   // A tömb indexmutatójának törlése
        wIndex = 0;              // túlcsordulás esetén
    }
    for(iido=0; iido<2000; iido++)
    {
        Nop();                 // Késleltetés
    }
}
}

```

Mielőtt a program tesztelése során az ICD2 programozónk a földön végezné, érdemes az ICD2-t debug üzemmódból programozó üzemmódba átváltani. Válaszuk ki a **Programmer → Select Programmer → MPLAB ICD2** menüpont segítségével az ICD2-t programozóként. Ha így töltjük le a programunkat, akkor végleges változatként tudjuk futtatni, azaz ICD2 nélkül is működni fog a programunk, így már ki lehet húzni a fejlesztőpanelből az ICD2 kábelét. Most már csak arra kell vigyáznunk, hogy a tápkábelt ne rántsuk ki a falból.

14.3.2. Const előtag

A chLedKep változónk jelenleg két helyen foglalja a mikrokontroller memóriáját. A tömb kezdőértékei a programmemoriába (flashbe) kerülnek, hogy a tömb inicializálásakor az általunk megadott kezdőértékeket a program be tudja másolni az adatmemoriába (RAM-ba). A C nyelvben a változók alapesetben az adatmemoriában jönnek létre. Mivel a tömbök is változók, ezért a RAM-területen jönnek létre, hogy a program futása közben a tömb elemeinek értékei módosíthatók legyenek. Sok esetben, mint az utolsó mintapéldában is, nincs szükségünk arra, hogy a tömbünk módosítható legyen, illyenkor érdemes kihasználni a konstans változók létrehozásának lehetőségét.

A Microchip C30 fordítójában, a const előtag használatával, a változó a programmemoriában jön létre, így nem foglalja az amúgys szükséges adatmemoriánkat. Az eredeti C nyelv nem különbözteti meg az adat- és programmemória fogalmát, így általános esetben a const előtag csak azt jelenti, hogy a fordító módosítási kísérletek esetén fordítási hibát ad, nem engedi a programunkat lefordítani. A const előtaggal rendelkező változóknak szigorúan kezdőértéket kell adni a deklarálásukkor, mert utána nincs lehetőségünk a konstans változók értékeinek módosítására.

Az eredeti ANSI C nyelv nem definiálja a programmemória fogalmát, mert a fordító Neumann architektúrára készült, ezért nem különbözteti meg a program- és adat-

memória fogalmát. Ebből következik, hogy a harvardi architektúrára készült C fordítóknak a nyelvet ki kellett bővíteniük ahhoz, hogy különböző adatterületre tudjanak változót elhelyezni. Annak a megadása, hogy egy változó ne az adat-, hanem a programmemoriába kerüljön, implementációfüggő, különböző C fordítóknál más a szintaktikája.

A const előtag segítségével deklarált változókat a C30 fordító, a PIC 16 bites kontrollereiben megtalálható PSV (Program Space Visibility) területre helyezi. A következő sor egy konstans változót deklarál:

```
const folat pi = 3.141593;
```

Az előző mintapéldában használt tömbünket a következő deklaráció segítségével tudjuk konstans változóként a programmemoriába elhelyezni:

```
const unsigned char chLedKep[] =      // A ledmintát tartalmazó táblázat
{
    0b00011000,
    0b00100100,
    0b01000010,
    0b10000001,
    0b01000010,
    0b00100100
};
```

14.3.3. Többdimenziós tömbök

A C nyelvben lehetőségünk van többdimenziós tömbököt is létrehozni. A deklarálás általános alakja a következő:

```
típus név[elemszám_N.Dimenzió]_[elemszám_2.Dimenzió][elemszám_1.Dimenzió];
```

Például a következőképpen lehet létrehozni egy négyzet hármás tömböt:

```
int tomb[4][3];
```

A többdimenziós tömb egyes elemeit hasonlóképpen érhetjük el, mint az egydimenziós tárak esetében tettük, csak most az összes dimenziót meg kell adnunk. A tömb elemeinek indexelése többdimenziós tömbök esetén is 0-tól kezdődik, és a (tömb adott dimenziójának elemszáma-1)-ig tart.

```
tomb[2][1] = 3; // A tömb (2;1) indexű elemének értéke
                  legyen egyenlő hárommal.
a = tomb[3][2]; // Az a változó értéke legyen egyenlő
                  a tömb (3;2) indexű elemének értékével.
```

Többdimenziós tömbök deklarálásakor is lehetőségünk van kezdőérték adására. Az elemeket egymás után is meg lehet adni, nem muszáj belső zárójeleket használni, de ha kizároljuk a dimenziókat, akkor sokkal áttekinthetőbb lesz a programkódunk.

```
int tomb[4][3] = { { 1 , 2 , 3 },
                  { 4 , 5 , 6 },
                  { 7 , 8 , 9 },
                  { 10 , 11 , 12 } };
```

Az előbb deklarált tömb elemei úgy helyezkednek el egymás után a memóriában, ahogy azt a 14.7. ábra mutatja.

| | | |
|-----------------|-----------------|-----------------|
| tomb[0][0] = 1 | tomb[0][1] = 2 | tomb[0][2] = 3 |
| tomb[1][0] = 4 | tomb[1][1] = 5 | tomb[1][2] = 6 |
| tomb[2][0] = 7 | tomb[2][1] = 8 | tomb[2][2] = 9 |
| tomb[3][0] = 10 | tomb[3][1] = 11 | tomb[3][2] = 12 |

14.7. ábra
Tömb elemeinek elhelyezkedése a memóriában

Kezdőérték adásával összekötött többszintű tömbdeklaráció esetén is megtehetjük, hogy nem adjuk meg az összes dimenzió elemszámát. Ilyen esetben elhagyhatjuk a legmagasabb dimenzió elemszámát, mint ahogy azt a következő példa is mutatja.

```
int tomb[] [3] = { { 1 , 2 , 3 },
                   { 4 , 5 , 6 },
                   { 7 , 8 , 9 },
                   {10 , 11 , 12 } };
```

14.3.4. Karakterláncok

A C nyelvben, ellentétben sok más nyelvvel, nincs külön string típusú változó. A C nyelv a karakterláncokat karaktertömbök formájában tárolja. A karaktertömbök megegyeznek a hagyományos egydimenziós tömbökkel (matematikusok kedvéért: vektorokkal), de egy plusz kikötéssel rendelkeznek: A tömb utolsó elemének **EOS** (End of String) ASCII karakterek kell lennie (**NULL** karakter), amelynek az értéke bináris nulla.

Egy karaktertömbben az egyes karakterek ASCII kódja kerül eltárolásra. Ha egy karakter ASCII kódját szeretnénk megadni, akkor azt két egyes aposztróf ('') között, az adott karakter beírásával tudjuk megtenni. Például, ha a z karakter ASCII kódjára van szükségünk, akkor azt a 'z' kifejezés segítségével tudjuk megadni. Az alap ASCII kódtábla csak hétbites, 128 karaktert tartalmaz. A felső 128 karaktert is használó ASCII táblákat kiterjesztett ASCII tábláknak hívjuk. A kiterjesztett ASCII táblákat már ritkán használjuk, mert a helyébe lépett a UNICODE karakterábrázolás.

A 14.8. ábra az alap ASCII táblázat egyes karaktereinek hexadecimális értékeit mutatja.

Az oszlopok tetején lévő számok a nagyobb, a sorok elején lévő számok a kisebb helyi értékű számjegyek. A táblázatból kiolvasva a 'z' karakter értéke 0x7A.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|-----|-----|-------|---|---|---|---|-----|
| 0 | NUL | DLE | space | 0 | @ | P | ' | p |
| 1 | SOH | DC1 | XON | I | 1 | A | Q | a |
| 2 | STX | DC2 | * | 2 | B | R | b | r |
| 3 | ETX | DC3 | XOFF | # | 3 | C | S | c |
| 4 | EOT | DC4 | \$ | 4 | D | T | d | t |
| 5 | ENQ | NAK | % | 5 | E | U | e | u |
| 6 | ACK | SYN | & | 6 | F | V | f | v |
| 7 | BEL | ETB | ' | 7 | G | W | g | w |
| 8 | BS | CAN | (| 8 | H | X | h | x |
| 9 | HT | EM |) | 9 | I | Y | i | y |
| A | LF | SUB | * | : | J | Z | j | z |
| B | VT | ESC | + | ; | K | [| k | { |
| C | FF | FS | . | < | L | \ | l | |
| D | CR | GS | - | = | M |] | m | } |
| E | SO | RS | . | > | N | ^ | n | ~ |
| F | SI | US | / | ? | O | _ | o | del |

14.8. ábra
ASCII karaktertáblázat

14. fejezet: Vissza a változókhöz

Üres karaktertömböt a már megismert tömbdeklarációval lehet létrehozni:

```
char szSzoveg[8]; // 7 karakter tárolására, az utolsó karakternek
// bináris nullának (NULL) kell lennie!
```

A karaktertömb deklarálásakor lehetőségünk van kezdőérték adására is.

```
char szSzoveg[] = {'C', ' ', 'n', 'y', 'e', 'l', 'v', 0};
```

A nyelv egy egyszerűbb formát is biztosít a karakterláncok megadására. A nullával záró karaktertömböt idézőjelek ("") között is megadhatjuk.

```
char szSzoveg[] = "C nyelv"; // Az így létrehozott tömb is
// nyolc bájt hosszúságú
```

A karakterláncok megadásakor bizonyos vezérlő karaktereket adhatunk meg, fordított törvonal után. A 14.9. ábra által felsorolt karakterek föleg a terminálprogramokkal való kommunikációkor kapnak szerepet.

| Jelölés | Beillesztett vezérlőkarakter |
|---------|------------------------------|
| \a | csengő karakter |
| \b | visszaléptetés karakter |
| \f | lapdobás |
| \n | új sor (soromelés) |
| \r | kocsivissza |
| \t | tabulátor karakter |
| \v | függőleges tabulátor |

| Jelölés | Beillesztett vezérlőkarakter |
|---------|---------------------------------|
| \e | escape karakter (csak gcc) |
| \\\ | backslash karakter |
| \' | aposztróf |
| \" | idézőjel |
| \?. | kérődjel |
| \ooo | ooo oktális kódú karakter |
| \hhh | hhh hexadecimális kódú karakter |

14.9. ábra
Vezérlőkarakterek kódjai

Hivatalosan nem jelenik meg a C30 fordító dokumentációjában, de a fordító képes kezelni a UNICODE karaktereket is. A UNICODE karakterek 16 bit hosszúságúak, ezért ezeket unsigned short típusként kell definiálni. Ha az stdlib.h fejlcálmányt is betöljük, akkor pedig a szokásos wchar_t származtatott típust is használhatjuk. (A származtatott változótípusról a 15.1. fejezetben lesz szó.) A UNICODE karaktereket és karakterláncokat „L” előtaggal kell ellátni. Sajnos konstansként csak ékezet nélküli karaktereket lehet megadni, mert az ékezes karaktereket a fordító nem tudja lefordítani.

```
unsigned short wCh = L'z';
unsigned short wszSzoveg[] = L"C nyelv";
vagy az #include<stdlib.h> után használva:
wchar_t wCh = L'z';
wchar_t wszSzoveg[] = L"C nyelv";
```

14.3.5. Készítsünk fényújságot!

A következő mintapéldában az utoljára megismert többszintű tömböket és a karakterláncokat is ki fogjuk használni. Az új program az előző programunk folytatatása, habár a teljes belső magot lecseréljük, csak az alapötletet fogjuk megtartani. Kirajzoltuk a LED-

sorra az előző programunk élére állított négyzeteket. A fejlesztőpanel jobbra-balra mozgatásával a négyzetek meg is jelentek a szemünk előtt.

A mostani program nemcsak négyzeteket fog majd kirajzolni a mozgó LED-sorra, hanem egy általunk megadott szöveget is. Nézzük, mire van szükségünk ahhoz, hogy elkezszítsük a fényújságprogramunkat.

- Először szükségünk van egy karaktermintákat tartalmazó táblázatra. A chLedKep változó egy kétdimenziós tömb, amely minden egyes dimenzióban egy betű bitmintáját tartalmazza. Egy karakterminta öt bájtból áll. Az első négy bájt az adott karakter biteszkját tartalmazza, az ötödik bájt nulla, ami az egyes betük közötti elválasztáshelyet biztosítja. Az egyszerűség kedvéért most csak az ábécé első három nagybetűjének bitképét készítsük el. A táblázatot mindenki a saját kreativitása szerint folytassa.
- A wKarakterHossz változó egy konstans ötös értéket tartalmaz, egy betű karakter mintájának hosszát adja meg.
- A szSzoveg változó egy nullával záródó karakterláncot tartalmaz. Ez a szöveg fog a programunk által a LED-sorra kirajzolódni.
- A wStrIndex változó a szSzoveg tömb aktuális karakterére mutat.
- A wIndex változó az aktuális karakter aktuális fénymintájára mutat.
- Az iido változót a késleltető rutin használja.

A program a LED-sor inicializálásával kezdődik. A programunk magját képező ciklus a szSzoveg karakterlánc egyes karakterein megy végig. A belső ciklus az előző ciklus által kiválasztott betű képét teszi ki a kimenetre.

A belső ciklus magját egy utasítás képezi (a késleltetési rutinon kívül). Érdemes ezt az utasítást egy kicsit tüzetesebben átnézni.

```
LATA = chLedKep[szSzoveg[wStrIndex]-'A'][wIndex];
```

A chLedKep tömb alsó dimenziójának indexét a wIndex változó határozza meg. A felso dimenziót a szSzoveg[wStrIndex]-'A' kifejezés határozza meg. A szSzoveg[wStrIndex] az éppen kiírás alatt álló karakter ASCII kódját adja vissza. Ebből az értékből azért kell az 'A' karakter értékét kivonni, mert a chLedKep tömbben az ábécé első három betűje van csak megvalósítva.

14.7. mintaprogram

```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

main ( )
{
    const unsigned char chLedKep[3][5] = // Karaktermintákat
    {                                     // tartalmazó táblázat
        {                                // A betű
            0b11111100,
            0b00010010,
            0b00010010,
            0b11111100,
            0b00000000
        },
        {                                // B betű
            0b11111110,
            0b10010010,
            0b10010010,
            0b01101100,
            0b00000000
        }
    };
}
```

```
),
{
    0b01111100, // C betű
    0b10000010,
    0b10000010,
    0b01000100,
    0b00000000
}

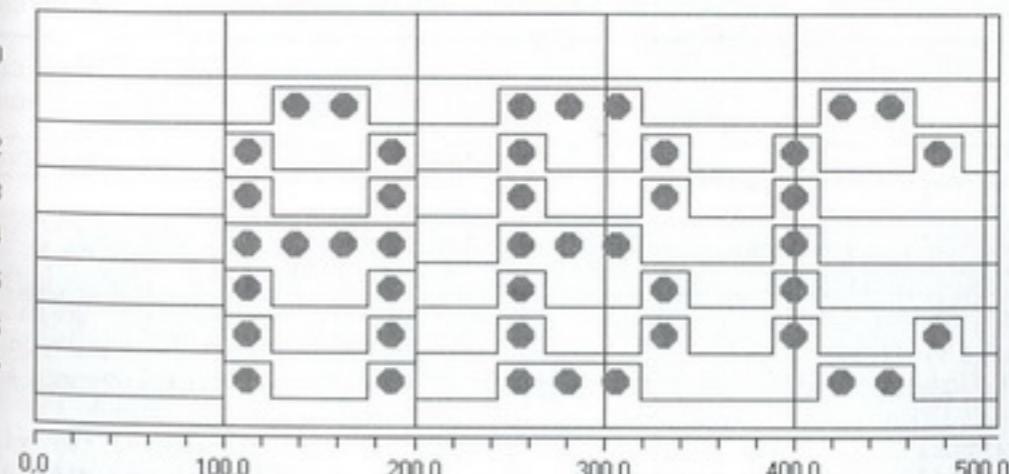
};

const unsigned int wKarakterHossz=5; // Egy karakter hosszúsága
const char szSzoveg[]="ABC"; // Kiírásra kerülő szöveg
unsigned int wStrIndex=0; // Karakterkijelölő index
unsigned int wIndex=0; // Fénymintát kijelölő index
int iido; // Késleltetéshez használt változó

TRISA = 0xFF00; // PORTA alsó nyolc lába kimenet.
LATA = 0x0000; // PORTA törlése

while(1) // Végtelen ciklus
{
    // Addig lépkedjünk karakterenként véig a karakterláncon,
    // amíg véget nem ér, azaz nullás karakter nem jön.
    for( wStrIndex=0; szSzoveg[wStrIndex] > 0; wStrIndex++ )
    {
        // Jelezzük ki a ledsoron a kiválasztott karaktert
        for( wIndex=0; wIndex < wKarakterHossz; wIndex++ )
        {
            // Fényminta kihelyezése a ledsorra
            LATA = chLedKep[szSzoveg[wStrIndex]-'A'][wIndex];
            for(iido=0; iido<2000; iido++)
            {
                Nop(); // Késleltetés
            }
        }
    }
}
```

A késleltető rutin nélkül program szimulációjának eredményét a 14.10. ábra mutatja.



14.10. ábra
Fényújság szimulációjának eredménye

14.4. MUTATÓARITMETIKA

A mutatókkal foglalkozó fejezetrészben csak értékeket adtunk mutatóknak vagy hivatkoztunk mutatókkal. Ezeken a műveleten kívül lehetőségünk van a mutatókkal bármelyik matematikai műveleteket is elvégezni. A mutatóaritmetikai műveletek három csoportra oszthatók:

- Mutatóhoz egész szám hozzáadása vagy kivonása:
 - pointer + int
 - pointer - int
 - pointer++, pointer--
 - ++pointer, --pointer
 - pointer - pointer
- Mutató inkrementálása vagy dekrementálása:
- Két mutató kivonása egymásból:

A mutatók alapegysége a mutató típusának bájtból értelmezett nagysága. Ha egy mutató értékét növeljük vagy csökkentjük, akkor a mutató alapegységével nő vagy csökken. Például egy int típusú mutatóhoz hozzáadunk egyet, akkor az értéke kettővel nő, mert az int típus nagysága (memóriaigénye) kettő bájt.

```
int *ptr; // int típusú változóra mutató
          // Az int típus memóriaigénye 2 bájt
ptr++; // a ptr értéke 1x2=2 bájttal nő
ptr = ptr + 3; // a ptr értéke 3x2=6 bájttal nő
```

Az általános típusú mutató (`void*`) alapegysége 1 bájt. A `void*` mutató segítségével bájtonként elérhetjük a teljes memoriaterületet.

Két mutató különbsége csak ugyanolyan típusú mutatók között van értelmezve, eredménye a két mutató címértékének különbsége lesz, de nem bájtból, hanem az adott változó bájtból mért alapegységében értelmezve. Márshog megfogalmazva: a két mutató különbsége azt mondja meg, hogy hány változónyira van egymástól a két mutató által mutatott két változó.

14.5. KAPCSOLAT A TÖMBÖK ÉS A MUTATÓK KÖZÖTT

A C nyelv a tömböket és a mutatókat hasonlóképpen kezeli. Ha nagyon sarkítva fogalmazunk, akkor azt is mondhatjuk, hogy igazából nincs különbség a mutatók és a tömbök között. A tömb típusú változók nevét (szöveges zárójelek nélkül) a tömb első elemére mutató pointerként kezeli a fordító. Ahhoz, hogy jobban megértsük az összefüggést, nézzük egy példát:

14.8. mintaprogram

```
main ( )
{
    int tomb[3]={10, 20, 30}; // 3 elemű, int típusú tömb
    int *ptr; // int típusú mutató
    int a, b, c, x, y, z;

    a=*tomb; // tomb 0. indexű elemének értékével tér vissza
    b=tomb[1]; // tomb 1. indexű elemének értékével tér vissza
    c=(tomb+2); // tomb+2 által mutatott memoriaterület értékével tér
    vissza

    ptr = tomb; // ptr mutasson a tomb-re

    x=*ptr; // ptr által mutatott memoriaterület értékével tér vissza
    y=ptr[1]; // ptr (tömb) 1. indexű elemének értékével tér vissza
    z=(ptr+2); // ptr+2 által mutatott memoriaterület értékével tér vissza
}
```

A programot lefuttatva a változók értékei a következőképpen alakulnak:

| Address | Symbol Name | Value |
|---------|-------------|--------|
| 0814 | tomb | |
| 0814 | [0] | 10 |
| 0816 | [1] | 20 |
| 0818 | [2] | 30 |
| 0812 | ptr | 0x0814 |
| 0810 | a | 10 |
| 080E | b | 20 |
| 080C | c | 30 |
| 080A | x | 10 |
| 0808 | y | 20 |
| 0806 | z | 30 |

14.11. ábra

Az a, b, c, x, y, z változók értékei tömbbel és mutatókkal történő értékkedés után

Az a, b, c változók értékét a `tomb` változó, amíg az x, y, z változók értékeit a `ptr` mutató segítségével adtuk meg. Az a, b, c változók értékei rendre megegyeznek az x, y, z változók értékeivel, holott különböző módon állítottuk be az értékeiket. A `ptr = tomb;` utasításból jól látható, hogy a tömb típusú változó egyben mutató is. Az utasítás véghajtása után a `tomb` és a `ptr` változó is a `0x0814` memóriacímrre mutat. Az a, b, c és az x, y, z változók értékkedésénél megfigyelhető, hogy a tömb típusú változók és a mutatók is egyformán kezelhetők. A mutatókat lehet tömbindexelő operátorokkal címezni, még a tömb típusú változókon is lehet címaritmetikát végezni.

A tömbindexelő operátor és a mutatók címaritmetikája között egy az egyes összefüggés írható fel:

```
pointer[egész] = *(pointer + egész)
```

Karakterláncok estén is megvan a tömbök és a mutatók kettősége. A karakterláncokról szóló 14.3.4. fejezetben kétféleképpen deklaráltunk karakterláncot. A második deklarációval, amikor idézőjelek között adtuk meg a karakterlánc értékét (`char szSzoveg[] = "C nyelv";`), akkor valójában létrejött egy különálló karaktertömb, amelynek címét a `szSzoveg` (mint mutató) kapta meg. Ebből következik, hogy lehetőségünk van egy harmadik fajta karaktertömb deklarálására is. Ilyen esetben nem tömböt deklarálunk, hanem egy mutatót, ami a fordító által létrehozott karaktertömbre mutat.

```
char *szSzoveg = "C nyelv";
```

Az idézőjelek közé elhelyezett szöveg ideiglenes karaktertömbként is működni. Ezt a tulajdonságot akkor tudjuk kihasználni, ha egy függvény bemeneti paramétere karaktertömböt vár. Ilyen esetekben nem kell egy karaktertömböt külön létrehozni, hanem a függvény hívásakor, a paraméterek között egyenesen csak a karakterláncot kell megadni. (A függvények létrehozásáról és használatáról a 16.1. fejezetben lesz szó.)

14.6. MUTATÓTÖMBÖK

Az eddigi fejezet részletekben megismertük a mutatókkal és a tömbökkel is. Most keressük össze a kettőt, és hozunk létre mutatótömböket. A következő példában egy kételemű tömb kerül deklarálásra, amelynek elemei int típusú változóra mutatók.

14.9. mintaprogram

```
main ( )
{
    int *ptr[2]; // Kételemű mutatótömb
    int a, b;

    ptr[0] = &a; // A 0. indexű elem mutasson az a változóra
    ptr[1] = &b; // Az 1. indexű elem mutasson az b változóra

    *ptr[0] = 10; // ptr[0] mutatott memóriaterület értéke
                   // legyen egyenlő 10-zel, azaz a = 10;
    *ptr[1] = 20; // ptr[1] mutatott memóriaterület értéke
                   // legyen egyenlő 20-szal, azaz b = 20;
}
```

A ptr tömb egyes elemei int típusú változóra mutatók. Ha a mutatótömb valamelyik elemevel mutatni szeretnénk egy memóriaterülethez, akkor azt az indirekciós * operátorral lehetjük meg, ugyanúgy, mint a hagyományos mutatók esetén. A 14.12. ábra az előző példa változóinak értékét mutatja, a program utolsó sorának végrehajtását követően.

| Address | Symbol Name | Value |
|---------|-------------|--------|
| 0806 | - ptr | |
| 0806 | [0] | 0x080A |
| 0808 | [1] | 0x080C |
| 080A | a | 0x000A |
| 080C | b | 0x0014 |

14.12. ábra

A mutatótömb elemeinek és az elemek által mutatott változók értéke

Mit kell tenni, ha a mutatótömb egyes elemeit nem tömboperátor segítségével szeretném elérni, hanem mutatóaritmetika használatával? Ebben az esetben a következőképpen alakul a programunk, amely működés szempontjából megegyezik az előző mintapéldával. (Az ilyen megoldásokat ritkán alkalmazzuk, csak azért került a könyvbe, hogy mások által írt programkód értelmezése esetén világos legyen a két darab indirekciós * operátor megjelenése.)

14.10. mintaprogram

```
main ( )
{
    int *ptr[2]; // Kételemű mutatótömb
    int a, b;

    *ptr = &a; // A 0. indexű elem mutasson az a változóra
    *(ptr+1) = &b; // A 1. indexű elem mutasson a b változóra
```

14. fejezet: Vissza a változókhöz

```
    **ptr = 10; // ptr mutatott memóriaterület értéke által mutatott
                 // memóriaterület értéke legyen egyenlő 10-zel.
    **(ptr+1) = 20; // ptr[1] mutatott memóriaterület értéke
                     // memóriaterület értéke legyen egyenlő 20-szal.
}
```

És a végére a kegyelemdőfés! Hogyan tudunk a mutató tömbre mutatni? Íme, a megoldás:

```
int *ptr[2]; // Kételemű mutatótömb
int **p; // Mutató tömbre mutató
p = ptr; // A mutató mutasson a mutatótömbükre. Ezután előző példa
          // utolsó négy utasításában szereplő ptr lecserélhető p-re.
```

14.6.1. Karakterláncokat tartalmazó tömbök

Az idézőjelek közé helyezett szöveg esetén, ahogy arról a 14.5. fejezet elején már szó esett, a fordító egy különálló karaktertömböt hoz létre. Abban az esetben, ha egy tömb több karakterláncot tartalmaz, akkor a tömbben az egyes karakterláncok kezdőcíméi (mutatói) kerülnek eltárolásra. Mutatótömböket már hoztunk létre, most vegyitsük a karakterláncokkal, és hozzunk létre karakterlánc-tömböket. A következő példa egy a hétnapjait felsoroló karakterlánc-tömböt hoz létre.

```
char *chHetiNapjai[] = { "Hétfő",
                         "Kedd",
                         "Szerda",
                         "Csütörtök",
                         "Péntek",
                         "Szombat",
                         "Vasárnap" };
```

A karakterlánc-tömböt később kétdimenziós tömbökként tudjuk használni. Például ha a kettős indexű karakterlánc hármás indexű betűjére van szükségünk, ami a „Szerda” karakterlánc „r” betűje, akkor azt a chHetiNapjai[2][3]-t kifejezéssel érhetjük el. (Ne feledjük el, hogy a tömbök indexelése nullától kezdődik.)

14.7. MUTATÓK NAGYSÁGA

Mutató deklarálásakor a near és far előtaggal tudjuk megmondani a fordítónak, hogy hányszélességű mutatót hozzon létre. A near előtagú mutatók rövidebbek, így használatuk kisebb és gyorsabb programkódot eredményez, de általában a teljes memóriaterület nem címzhető meg velük. A far előtagú mutatók nagyobbak, így nagyobb memóriaterületet képesek megcímzni, de lassabb és nagyobb programkódot eredményez a használatuk.

A Microchip C30 fordítója, a PIC 16 bites mikrokontrollerek architektúrájából kiindulva, nem különbözteti meg a far és near mutatótípusokat. Függetlenül a deklaráláskor használt előtagtól a program- és az adatmemoriára mutató nagysága is 16 bit, azaz 2 bajt lesz.

A Microchip C30 fordító nem csak mutatók deklarálásakor használja a far és near attribútumokat. Változó deklaráláskor megadhatjuk, hogy egy változó far vagy near adatterületre kerüljön. A 16 bites PIC mikrokontrollerek architektúrájából következik, hogy az adatmemória első 8 kibiszaván lévő változókat különböző módszerekkel élémi, mint a felülete lévő területen lévő változókat. A mikrokontroller assembly nyelve segítségével az első 8 kibisző adatmemorián lévő változókat direkt és indirekt is el lehet érni, ezt a C fordító near adatterületnek hívja. A felső adatmemória csak indirekt memóriacímzéssel (mutató használatával) érhető el, ezt a C fordító far adatterületnek nevezi.

14.8. **SIZEOF() OPERÁTOR**

Az implementációfüggetlen programozásnak elengedhetetlen része a `sizeof()` operátor használata. Különböző implementációk esetén előfordulhat az, hogy ugyanolyan típusú változók, különböző cégek C fordítói esetén, különböző nagyságú memóriaterületet igényelnek. Ennek a problémának a kiküszöbölésére használható a `sizeof()` operátor, ami az adott változótípus, változó, tömb, mutató, a származtatott típusok és azok változónak bájtból mért memóriaigényét adja vissza. Az operátor fordítási időben kerül kiértékelésre, azaz csak olyan változók memóriaigényét tudja kiszámolni, amelyek nagysága a forrás-kód fordításának idején rendelkezésre áll. A következő program a `sizeof()` operátor használatára mutat pár példát.

14.11. mintaprogram

```
main ( )
{
    int a, b, c, d, e, f, g;
    int iSzam;
    int iTomb[5];
    long lTomb[5];
    float *ptr;
    char szStr1[] = "Szia!";
    char *szStr2 = "Szia!";

    a = sizeof(int); // a = 2; Az int típus 2 bájt memóriaigényű.
    b = sizeof(iSzam); // b = 2; Az iSzam változó 2 bájt memóriaigényű.
    c = sizeof(iTomb); // c = 10; Az iTomb tömb 5x2=10 bájt memóriaigényű.
    d = sizeof(lTomb); // d = 20; Az lTomb tömb 5x4=20 bájt memóriaigényű.
    e = sizeof(ptr); // e = 2; Az ptr mutató 2 bájt memóriaigényű.
    f = sizeof(szStr1); // f = 6; Az szStr1 karaktertömb 6 bájt hosszú.
    g = sizeof(szStr2); // f = 2; Az szStr2 mutató!!!, ezért 2 bájt hosszú.
}
```

Ahogy a mintapéldában is látható, a Microchip C30 fordítójában a mutató típusú változók memóriaigénye 2 bájt, mert a címzéshez 16 bitre van szüksége. A mutatók memóriaigénye architektúrafüggő. A mutatók nagysága annak a függvénye, hogy az adott hardver-architektúra mekkora címbusz segítségével éri el az adatmemóriáját.

A `sizeof()` operátor alap-változótípusok körében értelmezett visszatérési típusa `unsigned int`. Mivel a `sizeof()` operátor értékét sok helyen kihasználják a C fordítóhoz tartozó függvénykönyvtárak, ezért saját típust vezettek be rá, amelynek neve `size_t`. A `sizeof()` operátor hivatalos származtatott típusát a `stdlib.h` fejlecállomány definiálja.

14.9. DINAMIKUS MEMÓRIAKEZELÉS

A fejezet végén nézzük meg egy kicsit a dinamikus memória foglalás használatát. Ahogyan arról a 14.3. fejezetben már szó esett, a C nyelvben a tömbök statikus memória foglalás segítségével jönnek létre, azaz a tömb nagyságát fordítási időben tudunk kell, és a tömb nagyságát futási időben nem tudjuk módosítani.

Vannak azonban olyan esetek, amikor futási időben kell memóriát foglalnunk. Vegyük csak szemügyre a saját mobiltelefonunk telefonkönyvét. Vannak olyan bejegyzések, amihez több telefonszám is tartozik, van, ahol e-mail címet is beírtunk stb. Mekkora nagy-

ságú legyen egy telefonkönyv-bejegyzéshez tartozó tárhely? Ha a maximumot foglaljuk le, akkor pazaroljuk a memóriát, kicsibe meg nem fér bele az összes információ. Ilyen esetekre szolgál a dinamikus memória foglalás, amelynek segítségével a program futása közben, felhasználói igények szerint foglalhatjuk le a szabadon lévő memóriánkat.

Aki már foglalkozott programozással, az már valószínűleg hallott a STACK memóriaterületről. Ez a memóriaterület egy dinamikusan nyújtózkodó terület, ide kerülnek mentésre a függvényeink visszatérési értékei, megszakítások esetén a processzorkörnyezet változói stb.

Most megismerkedünk egy új memóriaterület, a HEAP fogalmával. A heap memóriaterület a dinamikus memória foglalások helye. A C30 fordítói környezetben a heap memóriaterület nagysága alapesetben 0 bájt. Mielőtt továbbmennénk a dinamikus memória foglalás részleteihez, változtassuk meg ezt az értéket nagyobbra azért, hogy használni tudjuk a fordító dinamikus memória foglalási szolgáltatását. A heap értékét a **Project → Build Options... → Project** menüpont segítségével kinyíló ablakban válaszuk ki a **MPLAB LINK30** fájlt. Itt az általános (**General**) kategória alatt, a „**Heap size:**” mezőben adhatjuk meg a dinamikus memória foglalásokra a linker által fenntartott memóriaterület nagyságát bájtból. Állitsuk be a heap memóriaterület nagyságát 256 bájtba.

A statikus, fordítási időben történő memória foglalásnak van egy nagy előnye: fordítási időben derül ki az, ha nincs elég memóriánk az adott tömb létrehozásához. Dinamikus memória foglaláskor futási időben kell megvizsgálnunk a memória foglalásunk sikerességet. A dinamikus memória foglalással foglalkozó függvények deklarációi a `stdlib.h` fejlcállományban találhatók.

Memóriaterületet a `void *malloc(size_t size);` függvény segítségével tudunk lefoglalni. A függvény paramétereinek meg kell adni a lefoglalni kívánt memóriaterületet bájtból. A függvény visszatérési értéke a lefoglalt memóriaterület kezdőcímére mutató érték, vagy `NULL`, ha nem sikerült az adott memóriaterületet lefoglalni. Mielőtt a lefoglalt memóriaterületet használni kezdenénk, minden esetben meg kell győzödni arról, hogy sikeres volt-e a memória foglalásunk. Ebből következik, hogy dinamikus memória foglalással dolgozó programokhoz vészforgatókönyvet is ki kell dolgozni arra az esetre, ha valamilyen oknál fogva elfogy a szabadon felhasználható memóriaterületünk.

A dinamikus memória foglalással lefoglalt memóriaterületeket használata után fel kell szabadítani a `void free(void *ptr);` függvény segítségével, amelynek bemeneti paramétere a felszabadítandó memóriaterület mutatója. Nagyon fontos, hogy minden szabadságot fel a már nem használt területeinket, mert ellenkező esetben egy idő után „felesszük” a saját memóriánkat. Ezt a jelenséget a szakirodalom memóriaszivárgásnak hívja, azaz a programnak már attól a ténytől elfogy a szabadon felhasználható memóriája, hogy fut.

Nézzük egy példát a dinamikus memória foglalásra. A példa egy 50 elemű, `int` típusú tömböt foglal le, amit sikeres foglalás után fel is szabadít.

14.12. mintaprogram

```
#include <p24fj128ga010.h>

// A NULL definíciója végett kell meghívni
#include <stdio.h>
// A malloc() és a free() függvények fejlécét tartalmazza
#include <stdlib.h>

_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRIPLL )

main(void)
{
}
```

```

int *tomb;

tomb = (int*) malloc( 50*sizeof(int) );
if ( tomb != NULL )
{
    // Sikeres memóriafoglalás esetén itt folytatódik a programunk.
    // A dinamikusan lefoglalt tömböket is ugyanúgy használjuk,
    // mint a hagyományos tömböket.

    tomb[30] = 120;

    // Ha már nem használjuk a tömböt, akkor fel kell szabadítani!
    free(tomb);
}
else
{
    // Sikertelen volt a memóriafoglalás
    // Itt kell a hiba kezelését megoldani.
    Nop();
}

```

A dinamikus memóriakezeléshez az előbb megismert két függvényen kívül még két függvény tartozik. A `void *calloc(size_t nelem, size_t size);` függvény hasonlóképpen működik, mint a `malloc()`, csak itt a két paraméter szorzata adja a lefoglalt memóriaterület nagyságát, és a memóriaterület egyben fel is töltődik nullával. A `void *realloc(void *ptr, size_t size);` függvény segítségével a már egyszer lefoglalt memóriaterület nagyságát változtathatjuk meg, más néven újrafoglalhatjuk a memóriaterületet.

Ezen kívül érdemes megjegyeznünk, hogy a fordítóhoz mellékelt függvénykönyvtárban található `printf()` függvény használja a heap memóriát, ezért felhasználásánál a heap memóriát be kell állítani.

15. MÉG MINDIG A VÁLTOZÓKRÓL

15.1. VÁLTOZÓTÍPUS-DEFINÍCIÓ TYPEDEF SEGÍTSÉGÉVEL

A `typedef` parancs segítségével lehetőségünk nyílik már meglévő változótípusokból új változótípust deklarálni, más néven származtatott változótípust készíteni. A `typedef` kulcsszó használatával kevesebbet kell gépeinknél, mert hosszú kifejezések helyett rövidebb megfelelőket alkalmazhatunk. A `typedef` parancs általános alakja a következő:

```
typedef meglévő_típus új_típus;
```

A következő példában az `unsigned int` változótípus helyett egy új típust fogunk bevezetni. Az új, `WORD` változótípus bevezetésével, új változó deklarációjakor nem kell az `unsigned int`-et kiírni, elég csak a `WORD` kifejezést használni.

```

typedef unsigned int WORD;

main (void)
{
    WORD wSzam=5;
}

```

Új változótípusokat definiálni a függvényeken kívül érdemes, mert ilyenkor a teljes forrásállományunkban elérhetők. Abban az esetben, ha a `typedef` kifejezést függvényen belül használjuk, akkor az adott típus csak abban függvényben látható.

Elterjedt szokás, hogy a `typedef` segítségével definiált új változótípusokat végig nagybetűvel írjuk. A jelölés használatával a beépített változótípusok, a származtatott változótípusok és a segítségükkel létrehozott változók könnyen megkülönbözhetők egymástól.

15.2. FELSOROLÁSTÍPUS

Vannak olyan esetek, amikor a változó számértéke nem érdekel minket, csak az érték által hordozott információra, jelentésre van szükségünk. Ilyen esetekre szolgál az `enum` (*enumerated data*) felsorolástípus. Mielőtt egy mintát néznénk az `enum` használatára, először ismerkedjünk meg egy új fogalommal, a **típusdeklarálással**.

Az eddigi fejezetekben csak változódeklarációt végeztünk (eltekintve a `typedef` használatától), azaz már meglévő változótípusból hoztunk létre változót. Típusdeklaráláskor nem új változót, hanem új változótípust deklarálunk.

A típusdeklarációs kifejezéseket háromféleképpen is használhatjuk. Lehetőségünk van csak új változótípust deklarálni, vagy egyszerre a változótípus-deklarációval, a frissen definiált változótípusból változókat is létrehozni. Ezenkívül lehetőségünk van a típusdeklaráció segítségével csak változót deklarálni, és új változótípust nem definiálni.

Az `enum` felsorolástípus deklarációjának az általános alakja a következő:

```
enum típusnév { felsorolás lista } változók;
```

Példaként hozzunk létre egy felsorolást a hétfő napjaiból:

```
enum HetNapjai {
    Hetfo, Kedd, Szerda, Csutortok, Pentek, Szombat, Vasarnap
} ma, holnap;
```

Az előző példában létrehoztunk egy `HetNapjai` nevű felsorolástípust, majd az új felsorolástípusból létrehoztunk két új változót is, ma és `holnap` névvel.

Felsorolástípus használatakor lehetőségünk van a típusnév vagy a változónév (vagy változólista) elhagyására. Ilyen esetben vagy csak felsorolástípust, vagy csak változót deklarálunk.

```
// Csak változótípust deklarálunk:
enum HetNapjai {
    Hetfo, Kedd, Szerda, Csutortok, Pentek, Szombat, Vasarnap
};

// Csak változót deklarálunk:
enum {
    Hetfo, Kedd, Szerda, Csutortok, Pentek, Szombat, Vasarnap
} ma, holnap;
```

Ha már létrehoztuk az új felsorolástípusunkat, akkor később bármikor létrehozhatunk belőle változókat. A változó létrehozásának általános alakja a következő:

```
enum típusnév változólista;
```

Hozzunk létre pár változót az előbb deklarált `HetNapjai` felsorolástípusból:

```
enum HetNapjai ma, holnap; // ma és a holnap változó létrehozása
enum HetNapjai tegnap=Kedd; // tegnap változó létrehozása kezdőértékkel
```

Ahogy az a `tegnap` változó deklarációjánál megjelenik, a felsorolás típusú változók értékei a típusdeklarációt definált lista elemei lehetnek. Nézzük pár példát az enum típusú változók használatára:

```
ma = Szerda; // A ma változó értéke legyen szerda
holnap = ma + 1; // A holnap változó értéke a ma változó értékénél
                  // legyen eggyel nagyobb.
```

Az első sor (`ma = Szerda;`) nem tartalmaz újdonságot, de mi van a második sorral? A második sorban a `holnap` változó értékét a `ma` változó eggyel növelt értékével tessük egyenlővé. Hogyan értelmezhető egy lista elemének eggyel növelt értéke? A válasz az enum típusú változó kettősségeiben rejlik. Az enum típusú változó egyben felsorolás, a változó értéke a listában felsorolt elemek egyikével lehet egyenlő, és egyben int típusú egész szám is. Ha külön nem definiáljuk felül, akkor a felsorolás egyes eleméinek értéke, nullától kezdve, a felsorolás sorrendjében, balról jobbra eggyel nő. Ebből következik, hogy a `HetNapjai` felsorolástípusban a `Hetfo` elem értéke 0, a `Kedd` elem értéke 1, és a `Vasarnap` elem értéke 6.

A C nyelv lehetőséget biztosít számunkra az egyes elemek értékeinek megadására is, amit a felsorolásban, az egyes elemek után egyenlőséggel segítségével adhatunk meg.

```
enum HetNapjai {
    Hetfo=1, Kedd=2, Szerda=3, Csutortok=4, Pentek=5, Szombat=6,
    Vasarnap=7
};
```

15. fejezet: Még mindig a változókról

Nem muszáj a listában szereplő összes elemeknek értéket adni. Ha nem adunk egy elemnek értéket, akkor az alapértelmezett szabály teljesül rá: **ha a lista első eleme, akkor nulla értéket kap, különben az előző elem eggyel növelt értékét.** Például ha azt szeretnénk, hogy a felsorolásunk egyes értékkel kezdődjön, akkor elég csak a lista első eleménél jeleznünk.

```
enum HetNapjai {
    Hetfo=1, Kedd, Szerda, Csutortok, Pentek, Szombat, Vasarnap
};
```

Az MPLAB IDE felülete is támogatja az enum felsorolástípusok megjelenítését. Ha egy enum típusú változót felveszünk a fejlesztőfelület Watch vagy Locals ablakába, akkor a változó számértéke és az elemértéke is megjelenik, ahogy azt a következő mintapélda végrehajtása után, az egyes változók értékeinél is láthatjuk (15.1. ábra).

15.1. mintaprogram

```
main(void)
{
    enum HetNapjai {
        Hetfo, Kedd, Szerda, Csutortok, Pentek, Szombat, Vasarnap
    } ma, holnap;
    enum HetNapjai tegnap=Kedd;
    ma = Szerda;
    holnap = ma + 1;
}
```

| Address | Symbol Name | Value |
|---------|-------------|-------------------|
| 090A | ma | 0x02 == Szerda |
| 0908 | holnap | 0x03 == Csutortok |
| 0906 | tegnap | 0x01 == Kedd |

15.1. ábra
A `HetNapjai` felsorolástípus változóinak értékei

Lehetőségünk van az enum kulcsszót a `typedef` előtaggal együtt használni. Ilyen esetben egy új változótípust deklarálunk, amely elő a változó deklarálásakor már nem kell tüenni az enum kifejezést. Az enum felsorolástípus deklarálásának általános alakja, a `typedef` utasítás alkalmazásával, a következőképpen alakul:

```
typedef enum ( felsorolás lista ) típusnév;
```

Nézzük az előző példánkat `typedef` kulcsszóval egybekötve:

15.2. mintaprogram

```
typedef enum (
    Hetfo, Kedd, Szerda, Csutortok, Pentek, Szombat, Vasarnap
) HET_NAPJAI;

main(void)
{
    HET_NAPJAI tegnap=Kedd, ma, holnap;
    ma = Szerda;
    holnap = ma + 1;
}
```

És végül egy hasznos típusdefiníció logikai változók használatára:

```
typedef enum { false = 0, true = 1 } BOOL;

BOOL logikai_valtozo;
logikai_valtozo = true;
```

15.3. STRUKTÚRÁK

Képzeljük el, hogy megkeresnek minket a következő feladattal: Készítsünk egy olyan programot, amely teherautók súlyának mérésére és a mérési eredmények tárolására alkalmas. A feladat egyszerűsítése érdekében tételezzük fel, hogy a mérés pillanatában rendelkezésünkre áll a mérésre váró teherautó rendszáma, a mérés ideje és a mérleg által mért érték. Milyen változókban mentsük el az egyes értékeket? Szükségünk van egy karakterláncra, amiben a kamion rendszámát tároljuk el. A változó hosszúságú karakterláncra azért van szükség, hogy a külföldi teherautók rendszámát is el tudjuk tárolni. Tételezzük fel, hogy napi mentést kell készítenünk, ezért a dátumot nem kell minden alakkal elmentenünk, és a megrendelőn megelégszik a perces pontossággal. A mérleg által mért értéket lebegőpontos számként tároljuk el. Az így kapott változók deklarációja a következőképpen néz ki:

```
char szRendSzam[10]; // Kamion rendszáma
unsigned char cOra; // Mérés idejének órája
unsigned char cPerc; // Mérés idejének perce
float fKg; // Kamion súlya
```

Egy teherautó mérlegelésének eredményét négy változó segítségével sikerült feldolgozunk. A négy változó egy mérési esemény különböző értékeit tárolja. Nem lehetne a négy változó helyett csak egyet használni? A problémára a struktúrák bevezetése ad megoldást.

15.3.1. Struktúrák deklarálása

A struktúrák összetett adatszerkezetek, amelyek segítségével különböző típusú, de logikailag összefüggő adatokat tárolhatunk. A struktúrák egyes elemeit külön-külön is el tudjuk érni, de együttesen kezelhetjük a struktúrát. A struktúrák egyes elemei már az eddig megismert változótípusok vagy a már korábban definiált struktúrák lehetnek.

A struktúra deklarálásának általános alakja hasonlít a nemrég megismert felsorolástípus deklarálásához.

```
struct tipuscimke
{
    típus változó_név1;
    típus változó_név2;
    ...
} hivatkozási_nevek;
```

- Hasonlóan a felsorolástípushoz (enum), a struktúrák készítését is két részre lehet bontani.
- Lehetőségünk van csak új típusú struktúrát definiálni, ebben az esetben nem történik memóriafoglalás. Struktúradefiníció esetén csak a tipuscímkkét kell megadni.
- Lehetőségünk van egyszer használatos struktúrából készült változót deklarálni. Ilyen esetben létrejön az új változó a memoriában. Az egyszer használatos struktúra deklarációja esetén csak a hivatkozási neveket kell megadni.
- Végül lehetőségünk van új típusú struktúrát definiálni, és egy lépésekben változókat is létrehozni. Ilyen esetben a tipuscímkkét és a hivatkozási neveket is meg kell adni.

15. fejezet: Még mindig a változókról

Ahhoz, hogy egy teherautó mérlegelésének adatait egyszerre tudjuk kezelní, hozunk létre egy struktúrát.

```
struct stMerleg
{
    char szRendSzam[10]; // Teherautó rendszáma
    unsigned char cOra; // Mérés idejének órája
    unsigned char cPerc; // Mérés idejének perce
    float fKg; // Teherautó súlya
} stMerlegAdat;
```

Nézzük azokat az eseteket, amikor csak új struktúratípust definiálunk, vagy egyszer használatos struktúrából változót deklarálunk:

```
// Csak típust deklarálunk:
struct stMerleg
{
    char szRendSzam[10]; // Teherautó rendszáma
    unsigned char cOra; // Mérés idejének órája
    unsigned char cPerc; // Mérés idejének perce
    float fKg; // Teherautó súlya
};
```

```
// Csak változókat deklarálunk:
struct
{
    char szRendSzam[10]; // Teherautó rendszáma
    unsigned char cOra; // Mérés idejének órája
    unsigned char cPerc; // Mérés idejének perce
    float fKg; // Teherautó súlya
} stMerlegAdat, stMerlegAdat2, *pstMerlegAdat;
```

Már deklarált struktúrából később létrehozhatunk változókat. A változó létrehozásának általános alakja a következő:

```
struct tipuscimke hivatkozási_nevek;
```

Hozzunk létre pár változót az előbb deklarált stMerleg struktúrából:

```
struct stMerleg stMerlegAdat, stMerlegAdat2, *pstMerlegAdat;
```

Most már más dolgunk nincs, mint az stMerleg típusú struktúrából egy tömböt létrehozni, így már több mérési eredményt is el tudunk tárolni:

```
struct stMerleg stMerlegTomb[30];
```

Struktúrák és struktúratömbök deklarálásakor is lehetőségünk van kezdőérték adására. Ilyenkor az egyes elemek értékeit halmazzárójelek között, vesszővel elválasztva kell megadni.

```
struct stMerleg stMerlegTomb[] =
{
    { "ABC123", 10, 20, 4050.2 },
    { "DEF456", 10, 22, 8052.3 }
};
```

A sizeof() operátor struktúrák esetén is alkalmazható. Ilyen esetben az operátor az adott struktúra memóriaigényével tér vissza. Például a sizeof(stMerlegAdat) $10+1+1+4=16$ értékkel fog visszatérni.

15.3.2. Hivatkozás a struktúrák elemeire

Most, hogy sikerült a mérlegelés adataira alkalmas struktúrát létrehoznunk, nézzük, hogyan tudjuk a struktúra egyes elemeit elérni. Ha egy struktúra valamelyik elemére szeretnénk hivatkozni, akkor azt a mezőkiválasztó operátor (.) segítségével tehetjük meg. A struktúra adott elemére történő hivatkozás általános alakja a következő:

```
hivatkozási_név.elem_név
```

Ha az stMerleg struktúrából létrehozott stMerlegAdat változó cOra elemének szeretnénk értékét adni, azt a következőképpen tehetjük meg:

```
stMerlegAdat.cOra = 10;
```

Az stMerleg struktúra szRendSzam elem egy tömb. A tömb egyes elemeit az elemneve után megjelenő tömboperátor segítségével érhetjük el. Ha a rendszám egyes indexű elemét szeretnénk megváltoztatni, azt a következőképpen tehetjük meg:

```
stMerlegAdat.szRendSzam[1] = 'B';
```

Struktúratömb esetén a hivatkozási név után, a mezőkiválasztó operátor előtt kell az elemszámot jelezni. Ha az stMerlegTomb tömb kettes indexű elemének fKg elemét szeretnénk elérni, azt a következőképpen tehetjük meg:

```
stMerlegTomb[2].fKg = 3822.5;
```

Abban az esetben, ha az stMerlegTomb tömb kettes indexű elemének szRendSzam nevű tömb egyes indexű elemét szeretnénk elérni, azt a következőképpen tehetjük meg:

```
stMerlegTomb[2].szRendSzam[1] = 'B';
```

Struktúrák esetén is lehetőségünk van mutatók használatára. Nézzük egy példát struktúramutató használatára:

```
struct stMerleg stMerlegAdat;
struct stMerleg *pstMerlegAdat;

pstMerlegAdat = &stMerlegAdat; // A mutató az stMerlegAdat-ra fog mutatni
(*pstMerlegAdat).fKg = 3500; // pstMerlegAdat által mutatott struktúra
// fKg elemének módosítása
```

A struktúramutatók használatának egyszerűsítése érdekében a nyelvben a (*struktúra_mutató).elem_név kifejezés kiváltására bevezetésre került a „->” operátor. A „->” operátor használatával az előbbi kifejezés a következőképpen módosul:

```
struktúra_mutató->element_név
```

Az előző példa a „->” operátor használatával a következőképpen módosul:

```
pstMerlegAdat = &stMerlegAdat; // A mutató az stMerlegAdat-ra fog mutatni
pstMerlegAdat->fKg = 3500; // pstMerlegAdat által mutatott struktúra
// fKg elemének módosítása
```

15.3.3. Struktúrák egybeágazása

Fejlesszük tovább az előző példáinkat. Most készítünk egy olyan struktúrát, amely nemcsak a mérlegelés idejét, hanem a mérlegelés napjának dátumát is tartalmazza. Azért, hogy ne legyen nagyon hosszú a struktúra, a dátumot és az időt külön-külön készítünk el egy struktúrába.

15.3. mintaprogram

```
struct stDATUM
{
    unsigned char cEv; // 0-99 évszám utolsó két számjegye
    unsigned char cHonap; // 1-12 hónap száma
    unsigned char cNap; // 1-32 nap száma
};

struct stIDO
{
    unsigned char cOra; // 0-23 óra
    unsigned char cPerc; // 0-59 perc
    unsigned char cMasodperc; // 0-59 másodperc
};

struct stMerleg
{
    char szRendSzam[10]; // Kamion rendszáma
    struct stDATUM stDatum; // Mérés napjának dátuma
    struct stIDO stIdo; // Mérés ideje
    float fKg; // Kamion súlya
};
```

Az stMerleg struktúra definíciójánál megfigyelhető, hogy a mérés dátumának és idejének egyes értékeit nem külön-külön változókban tároljuk, hanem egy struktúrában. A főprogramunkban hozunk létre az stMerleg struktúrából egy stMerlegAdat változót. A változónak adjuk kezdőértékét, úgy, ahogy a következő sorok mutatják.

```
struct stMerleg stMerlegAdat = { "ABC123",
{ 8, 10, 26 },
{ 10, 12, 34 },
12556.7 };
```

Az egybeágazott struktúrák egyes elemeit természetesen most is külön-külön el lehet érni. A struktúrák faszerkezetet alkotnak, úgy, ahogy az adattárolásnál használt könyvtárszerkezetek is. A könyvtárszerkezetekben vannak könyvtárak, amik itt a struktúrák, és vannak állományok, amik a struktúrák változói. Abban az esetben, ha az stDatum struktúra cHonap változó értékét szeretnénk elérni, azt a következőképpen tehetjük meg:

```
stMerlegAdat.stDatum.cHonap = 11;
```

Az stMerlegAdat struktúrváltozó egyes elemeinek értékeit az MPLAB IDE Locals ablakában is nyomon követhetjük, ahogy azt a 15.2. ábra is mutatja.

| Address | Symbol Name | Value |
|---------|--------------|----------|
| 0906 | stMerlegAdat | |
| 0906 | szRendSzam | "ABC123" |
| 0906 | [0] | 'A' |
| 0907 | [1] | 'B' |
| 0908 | [2] | 'C' |
| 0909 | [3] | '1' |
| 090A | [4] | '2' |
| 090B | [5] | '3' |
| 090C | [6] | '.' |
| 090D | [7] | '.' |
| 090E | [8] | '.' |
| 090F | [9] | '.' |
| 0910 | stDatum | |
| 0910 | cEv | 8 |
| 0911 | cHonap | 11 |
| 0912 | cNap | 26 |
| 0913 | stIdo | |
| 0913 | cOra | 10 |
| 0914 | cPerc | 12 |
| 0915 | cMasodperc | 34 |
| 0916 | fKg | 12556.70 |

15.2. ábra

stMerlegAdat struktúra elemeinek értékei

A struktúrák használatával változóink elnevezéseinek hossza megnő. A túl hosszú változónevek egy idő után kezelhetetlennek válnak, mert minél hosszabb a változó neve, annál nagyobb az esélye, hogy elgépeljük azt. Ennek a problémának a kiküszöbölése érdekében az MPLAB IDE fejlesztőkörnyezet ismeri a változónevek kiengészítésének funkcióját. A szolgáltatást az **Edit → Properties...** menüpont segítségével megnyíló ablak **Tooltips** füle alatt, az **Autocomplete** részben lehet aktiválni. A szolgáltatás bekapcsolása után a szerkesztőablakunkban, a **Ctrl+Space** billentyűkombináció lenyomásakor egy lista jelenik meg, ami az utolsó fordításnál használt változóink nevét tartalmazza.

Struktúrába bármilyen, már előtte definiált struktúrát be lehet ágyazni. Mi van abban az esetben, ha az éppen deklarálás alatt álló struktúrákat szeretnénk elemként is megadni? Természetesen ez nem oldható meg, mert az végtelen mélységű struktúrát okozna. De arra van lehetőségünk, hogy a struktúrába a deklarálás alatt lévő struktúránakra mutató helyezzünk el.

```
struct stElem
{
    int iErtek;
    struct stElem *ptrElem;
};
```

A saját típusára mutató struktúrákkal például láncolt listák készíthetők. A láncolt listákat érdemes dinamikus memóriakezeléssel együtt használni.

15.3.4. Bitstruktúrák

A mikrokontrolleres problémák egyik sajátossága, hogy nagyon sok esetben kell bitekkel műveleteket végezni. Vegyük csak egy olyan egyszerű esetet, amikor egy kimeneti lábra kötött LED-et kell be- vagy kikapcsolunk. Az ilyen problémákat eddig csak bitoperátorok segítségével tudtuk megoldani. A C nyelv lehetőséget biztosít arra, hogy struktúrák elemeiként a **char**, 8 bites változó típusánál kisebb bitszámú, akár egy bit hosszúságú elemeket is deklaráljunk.

A programozó fantáziájának egyik korlátja az adatmemória nagysága. Általában minél több adatot szeretnénk minél kisebb helyen tárolni. Nézzük például az előző fejezetben használt **stDATUM** struktúrát. A mérés napját három bajtban tároltuk el, egy bajt hosszúságú a **cEv**, a **cHonap** és a **cNap** mező is. Biztosan szükségünk van három bajtra egy dátum tárolásához? Ha az évnek csak az utolsó két számjegyét tároljuk el, akkor elég hozzá 7 bit ($2^7 = 128$). Hónap 4 bitben fér el ($2^4 = 16$), míg a nap tárolása 5 bitet igényel ($2^5 = 32$).

A struktúrák és a unionok belsejében definíálhatók az alap-változótípusoktól eltérő bit-hosszúságú változók. A változó előjeles vagy előjel nélküli egész szám lehet. Ha a változó neve előtt **signed** előtag szerepel, akkor előjeles, ha **unsigned** előtag szerepel, akkor előjel nélküli szám lesz. Az adott változótag bithosszúságát a változó nevét követően, kettősponttal elválasztva adhatjuk meg. Módosítsuk az **stDATUM** struktúrát úgy, hogy csak az adatok tárolásához szükséges minimális bithosszúságú változókat tartalmazza.

```
struct stDATUM
{
    unsigned ev : 7;           // 0-99 évszám utolsó két számjegye
    unsigned honap : 4;        // 1-12 hónap száma
    unsigned nap : 5;          // 1-32 nap száma
};
```

Az így deklarált struktúra 16 bites hosszúságú változó lesz, ami az eredeti három bajt memóriaigényű struktúrához képest egy bajt spórolást jelent.

A változó bithosszúságú tagokat tartalmazó struktúrák alapegysége az adott architektúra szóhosszúsága. Ha az előző struktúrában csak három egy bites változót deklaráltunk volna, akkor is kétbajtos memóriaigényű struktúra jött volna létre, mert jelenleg 16 bites kontrollercsaládra fordítottuk le a programunkat. Ha 8 bites architektúráakra fordítanánk le a három bitet deklaráló struktúrákat, akkor csak egy bajt hosszúságú változó jönne létre.

Lehetőségünk van név nélküli tagot is definálni, ilyen esetben az adott mennyiségi bitszámot a fordító üresen hagyja. Abban az esetben, ha név nélküli, nulla bithosszúságú tagot definiálunk, akkor a fordító az adott szó töltését befejezi, és a következő tagokat új szóba kezdi lehelyezni. A következő struktúradeklaráció a különböző tagdefiníciókat mutatja be.

```
struct minta{
    unsigned tag_1 : 4; // 4 bit hosszúságú előjel nélküli változó
    unsigned      : 3; // 3 bit üresen marad
    signed       tag_2 : 1; // 1 bit hosszúságú előjeles változó
                    // A változó értéke 0 vagy -1 lehet
    unsigned      : 0; // A tárolás a következő szóban folytatódik
    signed       tag_3 : 6; // 6 bit hosszúságú előjeles változó
};
```

A változó bithosszúságú tagokat tartalmazó struktúrák használatánál figyelnünk kell arra, hogy a nyelv nem definiálja a feltöltés sorrendjét, ezért ez implementációfüggő. Az MPLAB C30 fordítója az egyes tagváltozókat a legkisebb helyi értékű bittől kezdi lehelyezni.

15.3.5. Előre deklarált bitek használata

A következő struktúra a P24FJ128GA010.H fejlécállományból származik. A struktúra az A PORT egyes bitjeit deklarálja.

```
typedef struct tagPORTABITS {
    unsigned RA0:1;
    unsigned RA1:1;
    unsigned RA2:1;
    unsigned RA3:1;
    unsigned RA4:1;
    unsigned RA5:1;
    unsigned RA6:1;
    unsigned RA7:1;
    unsigned :1;
    unsigned RA9:1;
    unsigned RA10:1;
    unsigned :3;
    unsigned RA14:1;
    unsigned RA15:1;
} PORTABITS;
```

A frissen deklarált struktúrából a fejlécállomány készítői a következő sorban egy PORTAbits nevű változót deklaráltak. Ökölszabályként kijelenthető, hogy a SFR regiszterek nagybetűvel írt neve után, kisbetűs bits utótaggal ellátott struktúrák segítségével lehet az adott SFR regiszter bitjeit elérni. A bitek elnevezései megegyeznek az adott mikrokontroller adatlapjában használt elnevezésekkel, és csak nagybetűkből állnak. A következő mintapélda az A PORT legkisebb helyi értékű bitjét először kimenetnek konfigurálja, majd az értékét egyesre állítja.

15.4. mintaprogram

```
#include <p24fj128ga010.h>

_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRIPLL )

main (void)
{
    TRISAbits.TRISA0 = 0; // Az A PORT legkisebb helyi értékű bitje
                           // kimenet lesz
    PORTAbits.RA0 = 1; // Az A PORT legkisebb helyi értékű bitjének
                       // értékét egyesre állítjuk
}
```

15.4. UNIONOK

A unionok deklarálása és használata teljes mértékben megegyezik a struktúrákkal, azzal az egy különbséggel, hogy struct előtag helyett union előtagot kell írni. A unionok abban különböznek a struktúráktól, hogy az egyes tagok nem egymás után következnek a memóriában, hanem ugyanarról a memóriacímtől kezdődnek. Mivel a unionban szereplő tagok ugyanazt a memóriaterületet használják, ezért nem függetlenek egymástól, hanem egymás szinonimái.

A következő union két tagot tartalmaz, egy int típusú és egy char típusú változót. A char típusú változó és az int típusú változó alsó helyi értékű bájtja ugyanazt a memóriaterületet használja.

```
union {
    unsigned char c;
    unsigned int i;
} valtozo;
```

A következő mintapélda második sorának végrehajtása után a valtozo.i értéke 0xAABB-ről 0xACC-re módosul:

```
valtozo.i = 0xAABB;
valtozo.c = 0xCC;
```

A unionokban lehetőségünk van névtelen struktúrák használatára is. Névtelen struktúrát akkor lehet használni, ha a struktúra elemeinek neve nem egyezik meg a union többi elemeinek nevével. A belső struktúratag használatával az előző példában használt int típusú változó minden bájtja elérhetővé válik:

```
union integer
{
    int word;
    struct
    {
        char low;
        char high;
    };
};

union integer a;
a.word = 0xAABB; // a.word értéke:0xAABB
a.high = 0xCC; // a.word új értéke:0xCCBB
```

15.5. OPERÁTOROK KIÉRTÉKELÉSI SORRENDJE

Most, hogy megismerkedtünk a nyelv összes operátorával, egy táblázat segítségével foglaljuk össze az egyes operátorok kiértékelési irányát és precedencia-sorrendjüket. A táblázatban szereplő operátorok a kiértékelési sorrendjük alapján csökkenő sorrendben szerepelnek. Az egy cellában szereplő operátorok kiértékelési sorrendje megegyezik.

| Operátor | Operátor leírása | Kiértékelés iranya |
|----------|---|--------------------|
| ++ -- | Postfix inkrementáló és dekrementáló operátor | Balról jobbra |
| () | Zárójel operátor | |
| [] | Tömbelem-kijelölő operátor | |
| . | Struktúraelem-kijelölő operátor | |
| -> | Struktúramutató elemet kijelölő operátor | |
| ++ -- | Prefix inkrementáló és dekrementáló operátor | |
| + - | Előjelként használt plusz és mínusz | Jobbról balra |
| ! ~ | Logikai és bináris tagadás | |
| (type) | Típuskonverzió | |
| * | Indirekciós operátor | |
| & | Címképző operátor | |
| sizeof | sizeof operátor | |

| Operátor | Operátor leírása | Kiértékelés irányába |
|----------|---|----------------------|
| * / % | Szorzás, osztás, maradékképzés | |
| + - | Összeadás és kivonás | |
| << >> | Bitenkénti eltolás balra és jobbra | Balról jobbra |
| < <= | Kisebb és kisebb-egyenlő operátor | |
| > >= | Nagyobb és nagyobb-egyenlő operátor | |
| == != | Egyenlő és nem egyenlő operátor | |
| & | Bitenkénti ÉS operátor | |
| ^ | Bitenkénti KIZÁRÓ-VAGY operátor | |
| | Bitenkénti VAGY operátor | |
| && | Logikai ÉS operátor | |
| | Logikai VAGY operátor | |
| ? : | Feltételes operátor | |
| = | Értékadó operátor | |
| += -= | Adott értékkel növelő és csökkentő operátor | |
| *= /= %= | Adott értékkel megszorzó, osztó, maradékképző operátor | |
| <<= >>= | Adott értékkel binárisan balra és jobbra eltoló operátor | Jobbról balra |
| &= ^= = | Adott értékkel binárisan és, kizáró-vagy, vagy műveletet végző operátor | Balról jobbra |
| , | Vessző operátor | |

15.3. ábra

Operátorok összefoglalása kiértékelési sorrendjük alapján

16. ÉS A KÉP ÖSSZEÁLL...

16.1. FÜGGVÉNYEK

Az eddigi programjainkban nem használtunk függvényeket, mert a felmerülő problémákat még egységesen tudtuk kezelni. Egy idő után a megoldandó feladatok „kezdenek a fejünkre nőni”, és már nem tudjuk együtt kezelni őket. Ilyen esetben érdemes a feladatainkat több apró részre bontani, majd ezeket külön-külön elkészíteni. Az apró részből álló programok nagy előnye az átláthatóságuk és a kód újrafelhasználhatósága. A kis feladatkörök bontott feladatokat érdemes univerzálisra elkészíteni, mert így később más programjainkban is fel tudjuk használni őket. A részfeladatokra bontás egyben növeli a kódunk biztonságát is, mert minden feladatot csak egyszer kell megvalósítanunk a programban, így a később módosításokat és teszteléseket is csak egy helyen kell elvégezni. A C nyelvben a részfeladatokra bontás eszköze a függvények használata. A függvények segítségével programunkat egymástól jól elkülöníthető részekre tudjuk bontani. Az így elkülönített programrészek függvényhívások segítségével tudnak kommunikálni. A függvények rendelkezhetnek saját változókkal, de lehetőség nyílik arra is a függvényhívás segítségével, hogy a függvények bizonyos változóik értékét kicséréljék egymással.

Vegyük elő a 12.5. fejezetben elkészített LED-villogató programunkat. A programba a for ciklus segítségével késleltetést készítettünk, amelyet a LED-sor be- és kikapcsolását követően is lefuttattunk. Ennek a programnak a forrásállománya a következő:

(Megegyezik a 12.5. mintaprogrammal)

```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

main ( )                                // Főprogram kezdete
{
    unsigned int uiIdo;                  // Késleltetéshez használt változó
    TRISA = 0xFF00;                     // PORTA alsó nyolc lába kimenet lesz.

    while(1)
    {
        LATA = 0x0000;                  // PORTA összes bitje nulla lesz.

        for(uiIdo=0; uiIdo<30000; uiIdo++)
        {
            // Szoftveres késleltetés:
            Nop();                      // 30000-szer fut le a nop utasítás.
        }

        LATA = 0x00FF;                  // PORTA alsó nyolc bitje egyes lesz.

        for(uiIdo=0; uiIdo<30000; uiIdo++)
        {
            // Szoftveres késleltetés:
            Nop();                      // 30000-szer fut le a nop utasítás.
        }
    }
}
```

Nem lehetne a két késleltető ciklus helyett csak egyet használni? A kérdés megoldása egy késleltető függvény elkészítése lesz. Mielőtt a konkrét, késleltetést végző függvényt elkészítenénk, először nézzük meg a függvénydefiniálás szabályait.

16.1.1. Függvények definiálása

A C nyelvben használt függvények hasonlítanak a matematikában használatos függvényekre. Egy függvénynek lehetnek bemenő paraméterei, és lehet visszatérési értéke is: $z = f(x,y)$. Precízebben fogalmazva: a C nyelvben használt függvénynek egy darab visszatérési értéke lehet és bármennyi bemenő paramétere. Ahogy azt már megszoktuk, a C nyelv típusos nyelv, azaz minden a bemenő, minden a visszatérési értékeknek típusos változónak kell lennie. A bemeneti paramétereket a függvénydefinícióban, a függvény neve után lévő zárójelek között, egymástól vesszővel elválasztva kell megadni. A függvényeket a fordító a nevük alapján különbözteti meg, ezért a függvényneveknek egyedinek kell lenniük. A függvénynevekre a változónevekkel azonos szabályok vonatkoznak. A függvény fejléce után, halmazzárójelek között a függvény belsejét tudjuk megadni. A függvénydefiníció általános alakja a következőképpen néz ki:

```
visszatérési_érték_típusa azonosító (típus1 praméter1, típus2 praméter2,
...
{
    /* Belső változók deklarációja; */

    /* Függvény törzse */

    return visszatérési_érték;
}
```

A függvénydefiníció általános alakja utolsó sorában egy `return` utasítás szerepel. A `return` utasítás hatására a függvény végrehajtása befejeződik, és a vezérlés visszakerül a hívó függvényhez. A `return` utasítás után szereplő kifejezés értéke lesz a függvény visszatérési értéke, ezért visszatérési értékkel rendelkező függvény esetén a `return` utasítást kötelezően használni kell. A visszatérési érték típusának meg kell egyeznie a függvénydefinícióban a függvény neve előtt álló típussal. Abban az esetben, ha nem egyezik a két típus, akkor a függvény típusa felé automatikus konverziót hajt végre a fordító.

Abban az esetben, ha nem szeretnénk, hogy visszatérési értékkel rendelkezzen a függvényünk, akkor a függvényt `void` típusúként kell definiálnunk. Ha egy függvénynek nincs visszatérési értéke, akkor nem kötelező a `return` utasítás használata, mert ilyen esetben, ha a vezérlés a függvény végéhez ér, akkor automatikusan visszatér a hívójához. Abban az esetben, ha nem adunk meg visszatérési típust a függvénydefinícióban a függvény neve előtt, akkor `int` típusú függvény keletkezik, de ez implementációfüggő lehet.

Lehetőségünk nyílik a bemeneti paraméterek elhagyására is. Ilyen esetben vagy üresen hagyjuk függvény neve után található zárójelek közötti részt, vagy csak egy `void` kulcsszót írjuk be.

A már definiált függvényt a függvény nevével tudjuk meghívni, ahol zárójelek között, vesszővel elválasztva felsoroljuk a bemenő paraméterek értékeit, amelyek kifejezések eredményei is lehetnek. Fontos, hogy a bemenő paraméterek sorrendjének meg kell egyeznie a függvény definíálásakor használt paramétersorrenddel. A függvényhívást pontosvesszővel kell lezární.

Módosítuk az előző programunkat úgy, hogy a várakozást egy külön függvénybe tesszük, és a várakozás helyén csak meghívjuk. A várakozó függvényüknek nem definiálunk visszatérési értéket és bemeneti paramétert sem. Ezenkívül a main függvényünk definícióját is „szépítük meg”, írjuk elő az `int` jelzöt, így a fordításkor nem kapunk egyetlen figyelmeztetést sem. Természetesen a main függvényünk soha nem fog visszatérni a hívójához, ezért a `return` utasítást nem kell használnunk.

A GNU C fordítója `int` típusú visszatérési értékkel rendelkező `main()` függvényt vár, mert az eredeti, linux operációs rendszerhez kifejlesztett C fordító is `int` típusú `main()` függvényekkel dolgozik. Más, kimondottan mikrokontrollerekre készített C fordítók általában `void` visszatérési típusú `main()` függvényt várnak. Mindig érdemes az adott fordító felhasználói kézikönyvében megnézni, hogy az adott fordító milyen típusú visszatérési értékkel rendelkező `main()` függvénytel szeret együtt dolgozni.

16.1. mintaprogram

```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

/* Szoftveres késleltető függvény */
void varakozas ( void )
{
    unsigned int uiIdo; // Késleltetéshez használt változó

    for(uiIdo=0; uiIdo<30000; uiIdo++)
        {
            Nop(); // Szoftveres késleltetés:
                    // 30000-szer fut le a nop utasítás.
        }
}

int main ( ) // Főprogram kezdete
{
    TRISA = 0xFF00; // PORTA alsó nyolc lába kimenet lesz.

    while(1)
    {
        LATA = 0x0000; // PORTA összes bitje nulla lesz.
        varakozas(); // Várakozó függvény meghívása.
        LATA = 0x00FF; // PORTA alsó nyolc bitje egyes lesz.
        varakozas(); // Várakozó függvény meghívása.
    }
}
```

A módosított programunkban a várakozó rutint csak egyszer kellett elkészítenünk. A főprogramban a várakozó rutint a LED-ek bekacsolása és kikapcsolása után is meghívjuk, így működésében nem változott meg a programunk.

Most módosítsuk úgy a programunkat, hogy különböző késleltetési időkkel is meg tudjuk hívni a késleltető rutinunkat. A várakozást végző függvényben a számláló (`uiIdo`) és a bemenő paraméter (`uiMaxIdo`) is legyen `unsigned int` típus.

16.2. mintaprogram

```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

/* Szoftveres késleltető függvény */
void varakozas ( unsigned int uiMaxIdo )
{
    unsigned int uiIdo; // Késleltetéshez használt változó

    for(uiIdo=0; uiIdo<uiMaxIdo; uiIdo++)
        
```

```

    {
        Nop();           // Szoftveres késleltetés:
        // uiMaxIdo-szor fut le a nop utasítás.
    }

int main ()           // Főprogram kezdete
{
    TRISA = 0xFF00;    // PORTA alsó nyolc lába kimenet lesz.

    while(1)
    {
        LATA = 0x0000;    // PORTA összes bitje nulla lesz.
        varakozas(30000); // Várakozó függvény meghívása.
        LATA = 0x00FF;    // PORTA alsó nyolc bitje egyes lesz.
        varakozas(60000); // Várakozó függvény meghívása.
    }
}

```

Az új késleltető függvényünk már rendelkezik egy bemeneti paraméterrel is, az uiMaxIdo változóval. A főprogramunk a várakozó rutint mind a kétszer más várakozási idővel hívja meg, így eltérő ideig világít és nem világít a LED-sor.

16.1.2. Függvények deklarálása

A függvényeket a használatuk pillanatában a fordítónak ismernie kell. Ez azt jelenti, hogy a függvény használata nem következhet hamarabb a forráskódban, mint a függvény ismertetése. Bizonyos fordítók nem kezelik ezt a problémát szigorúan vett fordítási hibának, de figyelmeztetést biztosan generálnak. Ebből következik, hogy a függvényeinket minden a main() függvény előtt kell elhelyezni? A válasz: nem, mert lehetőségünk van a függvény definiálása helyett függvénydeklarációra is.

A függvénydeklaráció egy előzetes információ a fordítónak a függvény nevéről, visszatérési érték típusáról, a bemeni paraméterek számáról és típusairól. A függvény-deklaráció általános alakja hasonlít a függvénydefiníciót használt fejléc alakjához, csak ebben az esetben nem kell a függvény belsejét is megadni, hanem pontosvesszővel kell lezárnai a deklarációt. Ilyenkor a fordító tudomásul veszi, hogy a függvény belseje később, esetleg más forrásállományban került kifejtésre. A függvénydeklaráció általános alakja a következőképpen néz ki:

```
visszatérési_érték_típusa azonosító (típus1 paraméter1, típus2 paraméter2,...);
```

Függvénydekláráskor nem kötelező a bemeneti paraméterek nevét is megadni, elég csak a bemeneti paraméterek típusát meghatározni. Így az előző függvénydekláráls általános alakjában a paraméter1, paraméter2 változónevek elhagyhatók.

Az előző példánkat módosítuk úgy, hogy a késleltetést végző függvényünk a főprogramunk után kerüljön, és a main() függvény előtt csak a varakozas() függvény deklarációja szerepeljen.

```

#include <p24fj128ga010.h>
_CONFIG1(JTAGEN_OFF & FWDTEN_OFF)
_CONFIG2(POSCMOD_HS & FNOSC_PRI)

/* Késleltető függvény deklarációja */
void varakozas ( unsigned int uiMaxIdo );

int main ()           // Főprogram kezdete
{
    TRISA = 0xFF00;    // PORTA alsó nyolc lába kimenet lesz.

    while(1)
    {
        LATA = 0x0000;    // PORTA összes bitje nulla lesz.
        varakozas(30000); // Várakozó függvény meghívása.
        LATA = 0x00FF;    // PORTA alsó nyolc bitje egyes lesz.
        varakozas(60000); // Várakozó függvény meghívása.
    }
}

/* Szoftveres késleltető függvény */
void varakozas ( unsigned int uiMaxIdo )
{
    unsigned int uiIdo;    // Késleltetéshez használt változó

    for(uiIdo=0; uiIdo<uiMaxIdo; uiIdo++)
    {
        Nop();           // Szoftveres késleltetés:
        // uiMaxIdo-szor fut le a nop utasítás.
    }
}

```

Ahogy az előbb már említettük, a fordítónak csak a függvény nevére és a használt változótípusokra van szüksége, ezért deklarációkor nem szükséges a változónevek megadása, elég a paraméterek típusainak felsorolása. Az előző mintapéldában használt függvénydeklaráció helyett a következő deklaráció is használható:

```
void varakozas ( unsigned int );
```

16.1.3. Függvények paraméterátadása

A következő mintapéldában egy összegfüggvényt találunk, ami a paramétereként kapott két szám összegét képzi. A példa egy kicsit „szószátyár”, hiszen a függvényben csak egy egyszerű összeadás szerepel. Valószínűleg a gyakorlati életben senki sem fog készíteni ilyen egyszerű függvényt. Ennek a függvénynek nem is az a célja, hogy gyakorlatban alkalmazzuk, hanem hogy a függvényen keresztül megértsük a függvények közötti paraméterátadás menetét.

```

/* összegfüggvény deklarációja */
int osszeg ( int a, int b );

int main ()           // Főprogram kezdete
{
    int x, y, z;
    x = 5, y = 4;
    z = osszeg( x, y );
}

```

```
/*
 * összeg függvény definíciója */
int osszeg ( int a, int b )
{
    int c; // Függvényre lokális változó.
    c = a + b; // A c értéke a két paraméter összege lesz.
    return c; // A függvény a c változó értékével tér vissza.
}
```

Nézzük meg, hogy mi történik az egyes változókkal a program futtatása közben! A 16.1. ábra a program végrehajtásának menetét, az egyes változók memóriacímét és utoljára felvett értékét mutatja.

```
int main ( )           int osszeg ( int a, int b )
{
    int x, y, z;         {
        int c;
        c = a + b;
        return c;
    }
}
```

| Address | Symbol Name | Value |
|---------|-------------|--------|
| 090A | x | 0x0005 |
| 0908 | y | 0x0004 |
| 0906 | z | 0x0009 |
| 0912 | c | 0x0009 |
| 0914 | a | 0x0005 |
| 0916 | b | 0x0004 |

16.1. ábra

Függvény végrehajtásának menete és a lokális változók címei és értékei

A main() függvény három saját, más néven lokális változót tartalmaz, az x, y, z változót. A lokális változók láthatóságával részletesebben a 16.2. fejezet foglalkozik. Az osszeg függvény paramétereinek az x és az y változó értéke szerepel. A függvény meghívása után a vezérlés a meghívott függvényre kerül. A függvény meghívását követően először az x változó értéke átmásolódik az a változóba, és az y változó értéke átmásolódik a b változóba, így a függvény paraméterlistájában megadott a és b változó értéke az x és y változó értékét veszi fel. Figyeljük meg, hogy az a és b változó más memóriacímen található, mint az x és y változó, így függetlenek egymástól. A c változó az osszeg függvény lokális változója. A return utasítás hatására a függvény visszatéréskor a c változó értéke lesz az osszeg függvény végrehajtására. Az osszeg függvény végrehajtása után a függvény visszatérési értékét a z változóba másolja át a program. Mivel a két függvény változói teljesen függetlenek egymástól, ha az osszeg függvényben az a és b változó értékét megváltoztatjuk, az a hívó x és y változó értékére nem lesz kihatással.

A most bemutatott paraméterátadást **érték szerinti paraméterátadásnak** nevezünk.

A meghívott függvény is hívhat meg függvényt, amely újabb függvényt hívhat meg, és így tovább. A függvényhívások mélységenek csak a rendelkezésre álló memória szab határt, mert függvényhíváskor a visszatérési címet, a paraméterként átadott változókat és bizonyos esetekben a függvény saját változóit is el kell menteni a stack memóriaterületre.

16. fejezet: És a kép összeáll...

A függvények paraméterei alapesetben egyirányúak, a meghívott függvény oldaláról bemenetnek számítanak, a bemeneti paraméterek értékeit a meghívott függvények nem tudják módosítani. Vannak olyan esetek, amikor a függvény paramétereit is módosítani szeretnénk, mert nem elég a függvények egy darab visszatérési értéke. Példaként egyes kommunikációs függvényeket lehet említeni. Az ilyen függvények visszatérési értéke a kommunikáció sikereségéről ad információt, míg az egyik paraméterén a kommunikáció adatforgalmát adja vissza a hívójának.

Abban az esetben, ha azt szeretnénk, hogy a meghívott függvény módosítani tudja a paraméterként megkapott változó értékét, akkor nem a változó értékét, hanem a változó memóriacímét kell átadnunk.

A változó címét átadó paraméterátadást **cím szerinti paraméterátadásnak** hívjuk.

Módosítsuk az előző programunkat úgy, hogy az osszeg függvény minden paramétere esetén ne az x és y változók értékét adjuk át, hanem az x és y változók memóriacímét.

16.5. mintaprogram

```
/* összeg függvény deklarációja */
int osszeg ( int *a, int *b );
```

```
int main ( ) // Főprogram kezdete
{
    int x, y, z;
    x = 5, y = 4;
    z = osszeg( &x, &y ); // A x és y változó címét küldjük át.
}
/* összeg függvény definíciója */
int osszeg ( int *a, int *b ) // Bemeneti paraméterek:
{
    int c; // Kettő darab int típusú változóra mutató
    c = *a + *b;
    return c;
}
```

A 16.2. ábra az előbbi mintapélda egyes változói címét és értékét mutatja. Érdemes megfigyelni, hogy az a és b változó értéke az x és y címe lesz. Így például ha az osszeg függvényben valahol beírjuk az *a=0; kifejezést, akkor az x változó értéke is meg fog változni.

| Address | Symbol Name | Value |
|---------|-------------|--------|
| 0908 | x | 0x0005 |
| 090A | y | 0x0004 |
| 0906 | z | 0x0009 |
| 0912 | c | 0x0009 |
| 0914 | a | 0x0908 |
| 0916 | b | 0x090A |
| 0908 | *a | 0x0005 |
| 090A | *b | 0x0004 |

16.2. ábra

Cím szerinti paraméterátadás változóinak értékei és a változók memóriacímei

Abban az esetben, ha tömböt szeretnénk paraméterként átadni egy függvénynek, akkor az átadni kívánt tömb mutatóját kell átküldenünk. A függvény paraméterét mutatóként kell definiálnunk, hogy fogadni tudja a tömb kezdőcímét. A következő mintapélda a tömböt mutatóként kezeli a fordító, ezért a tömb elemszámát is át kell adni a függvénynek.

16.6. mintaprogram

```
/* összeg függvény deklarációja */
int osszeg ( int *tomb, int elemszam );

int main ( ) // Főprogram kezdete
{
    int t[5] = { 1, 2, 3, 4, 5 };
    int eredmény;
    eredmény = osszeg( t , 5 ); // A tömb neve a tömb
                                // első elemének kezdőcíme.

/* összeg függvény definíciója */
int osszeg ( int *tomb, int elemszam ) // Bemeneti paramétere:
{ // A tömb és elemszáma
    int i, szum;
    for (i=0, szum=0; i<elemszam ; i++) // A ciklus a tömb összes elemét
    { // kiválasztja, és az egyes elemek
        szum+=tomb[i];
    }
    return szum; // A tömb egyes elemeinek értékének
                  // összegével tér vissza a függvény
}
```

Tömbök használatakor minden figyeljünk arra, hogy a tömbök esetén a paraméterben csak a tömb mutatója megy át, ezért a meghívott függvény módosítani tudja a paraméterként kapott tömb elemeinek értékét. A könnyebb kódolvasás érdekében lehetséges a függvény deklarációjának, illetve definíciójának paraméterlistájában üres tömbparamétert deklarálni. Az üres tömb csak szintaktikailag különbözik a mutatótól, használata teljes mértékben megegyezik. Az előző mintapéldában az int *tomb paramétert cseréljük le üres tömb (int tomb[]) paraméterre:

16.7. mintaprogram

```
/* összeg függvény deklarációja */
int osszeg ( int tomb[], int elemszam );

int main ( ) // Főprogram kezdete
{
    int t[5] = { 1, 2, 3, 4, 5 };
    int eredmény;
    eredmény = osszeg( t , 5 ); // A tömb neve a tömb
                                // első elemének kezdőcíme.

/* összeg függvény definíciója */
int osszeg ( int tomb[], int elemszam ) // Bemeneti paramétere:
{ // A tömb és elemszáma
    int i, szum;
    for (i=0, szum=0; i<elemszam ; i++) // A ciklus a tömb összes elemét
    { // kiválasztja, és az egyes elemek
        szum+=tomb[i];
    }
    return szum; // A tömb egyes elemei értékének
                  // összegével tér vissza a függvény
}
```

Ahogy látjuk, a tömbök csak mutatóként utaznak függvényhívásokor, ami nincs más képpen a függvény visszatérési értékével sem. Visszatérési értékként is csak mutatót lehet definiálni, tömböt nem (int* mutato(...)). Ebből következik, hogy a meghívott függvényben létrehozott tömbbel nem lehet visszatérni, mert a függvényben létrehozott válto-

zók csak a függvény végrehajtása idején élnek. Ilyen esetben a hívó függvény már csak egy érvénytelen területre kap vissza mutatót, mely memóriaterület értéke nem definiált, így instabil programműködéshez vezet.

Függvényhívás esetén a struktúrák érték szerint kerülnek átadásra, ezért nagy struktúrák használatakor érdemes csak a struktúramutatót átadni, mert az érték szerinti átadásnak nagy a memóriaigénye, mivel az egész struktúráról másolatot kell készítenie a fordítónak.

```
/* Struktúra használata érték szerinti paraméterátadással*/
int nap ( struct stDATUM d )
{
    return d.nap;
}

/* Struktúra használata cím szerinti paraméterátadással*/
int nap ( struct stDATUM *d )
{
    return d->nap;
}
```

16.1.4. Rekurzív függvényhívás

A nyelv lehetőséget ad arra, hogy a függvény a végrehajtása során újból meghívja saját magát. Az ilyen megoldásokat nagyon ritkán alkalmazzuk mikrokontrolleres problémák megoldása során, ezért nem az összes mikrokontrolleres C fordító támogatja a rekurzív függvényhívást. A rekurzív függvényhívások használatával nagyon óvatosan kell bánni, mert végtelen mélységű függvényhíváshoz vezethet, ami a program összeomlását okozza. Abban az esetben, ha a függvény meghívja önmagát, akkor az előző függvény összes változóját el kell menteni, hogy az új futás ne zavarja meg az előző futást. A rekurzív függvényhívások memóriahasználata magas, ezenkívül különböző paraméterek esetén változó lehet, ezért ha lehetséges, kerüljük az alkalmazásukat.

Most nézzünk egy kis példát a rekurzív függvényhívás használatára! A fakt függvény a paramétereként kapott szám faktoriálisát számolja ki ($n! = 1 \cdot 2 \cdot 3 \cdots \cdot n$). A függvény a következő rekurzív matematikai definíciót valósítja meg ($n \in \mathbb{N}$):

$$n! = \begin{cases} 1 & \text{ha } n \leq 1 \\ n \cdot (n-1)! & \text{ha } n > 1 \end{cases}$$

16.8. mintaprogram

```
unsigned int fakt ( unsigned int n )
{
    if ( n > 1 )
        return n*fakt(n-1); // Rekurzív függvényhívás
    else
        return 1;
}

int main ( )
{
    int eredmény;
    eredmény = fakt(5); // Faktoriális függvény meghívása
```

16.1.5. Függvénymutatók

Komolyabb programok igényelhetik, hogy a függvények hívási címét ne fordítási időben, hanem indirekt módon, függvénymutató segítségével, futási időben számoljuk ki. A mikrokontrolleres programok által megoldott problémák ritkán igénylik a függvénymutatók használatát. A következő mintapélda a teljesség igénye nélkül mutatja be a függvénymutatók használatát. A mintapélda a 16.1.3. fejezetben szereplő, két szám összegét képző program módosítása, ahol az összeg függvény nem direkt módon kerül meghívásra, hanem indirekt módon, függvénymutató segítségével.

16.9. mintaprogram

```
/* összeg függvény deklarációja */
int osszeg ( int a, int b );

int main ( )           // Főprogram kezdete
{
    int x, y, z;
    /* Függvénymutató deklarálása. A mutató olyan típusú függvényre fog
     * mutatni, amely int típusú visszatérési értékkel rendelkezik,
     * és két darab int típusú paramétert vár meghívásakor.
    */
    int (*ptr_osszeg)(int, int);
    x = 5, y = 4;

    /* A ptr_osszeg függvénymutató mutasson az osszeg függvényre. */
    ptr_osszeg = osszeg;

    /* A ptr_osszeg függvénymutató által mutatott függvény meghívása. */
    z = (*ptr_osszeg)( x , y );

    /* összeg függvény definíciója */
    int osszeg ( int a, int b )
    {
        int c;           // Függvényre lokális változó.
        c = a + b;       // A c értéke a két paraméter összege lesz.
        return c;         // A függvény a c változó értékével tér vissza.
    }
}
```

16.2. VÁLTOZÓK LÁTHATOSÁGA ÉS ÉLETTARTAMA

Egy program általában több függvényből áll össze. Az egyes függvények kommunikálhatnak egymással, például a függvények paraméterein és visszatérési értékein keresztül. Érdemes megnéznünk, hogy az egyes függvények hogyan látják a saját, illetve más függvények változóit, és az egyes változóknak mekkora az élettartamuk.

16.2.1. Lokális, automatikus változó

A függvények használatánál láttuk, a függvény belséjében deklarált változók csak a függvény belséjében látszanak, kívülről, más függvények számára elérhetetlenek. A függvény alapesetben az előző végrehajtására (állapotára) nem emlékszik, mert a függvény belső változói minden egyes híváskor újrainicializálódnak. Az ilyen változókat automatikus változóknak nevezzük. Az automatikus változók elő ki lehet írni az auto kifejezést, de ez nem szükséges, mert ha a változó deklarációban nem jelezzük, akkor a fordító auto típusúnak tekinti a változót.

```
auto int c;           // Függvényre lokális változó.
```

16. fejezet: És a kép összeáll...

16.2.2. Lokális, statikus változók

Bizonyos esetekben szükségünk van arra, hogy a függvényünk emlékezzen az előző állapotára, előző futásakor képződött eredményére. Azt, hogy a függvény emlékezzen a saját, előző állapotára, például az állapotgépek programozói modell alkalmazásával készült programknál vagy a megszakítások által meghívott függvények esetében lehet kihasználni.

Statikus változót a static kulcsszó segítségével tudunk létrehozni. A statikus változók a main() függvény meghívása előtt létrejönnek, és a program végrehajtásának teljes ideje alatt léteznek. A függvényekben létrehozott statikus változókat a függvényen kívül nem lehet elérni, csak a függvény belséjében láthatók. A statikus változók kezdőértékének adása a főprogram meghívása előtt megtörténik, a későbbiekben, a statikus változót tartalmazó függvény meghívásakor már nem történik kezdőérték adása. Hogyha nem adunk kezdőértéket egy statikus változónak, akkor nulla értéket fog felvenni a program inicializálását követően.

A következő mintapélda a statikus változók használatát mutatja be. A program egy szamlalo függvényt tartalmaz, ami minden meghívását követően visszatérési értékben közli a hívójával, hogy hányszor került eddig meghívásra.

16.10. mintaprogram

```
/* szamlalo függvény deklarációja */
int szamlalo ( void );

int main ( )           // Főprogram kezdete
{
    int a, b;
    a = szamlalo();      // szamlalo függvény első meghívása
    b = szamlalo();      // szamlalo függvény második meghívása
}

/* szamlalo függvény definíciója */
int szamlalo ( void )
{
    static int iSzamlalo=0; // Statikus, a függvényre lokális változó.
    // A változó kezdőértékadása a főprogram
    // indulása előtt megtörténik.
    iSzamlalo++;
    return iSzamlalo;      // A függvény az iSzamlalo változó
                           // értékével tér vissza a hívójához.
}
```

A főprogram futásának befejeztével az a változó egyet, míg a b változó kettes értéket fog felvenni, mert az a változó esetén először, míg b változó esetén másodszor került meghívásra a szamlalo függvény.

16.2.3. Globális változók

Ha egy változót függvényen kívül deklarálunk, akkor az adott forrásállomány összes függvényében láthatóvá válik. A globális változók, hasonlóképpen a statikus változókhöz, a main() függvény meghívása előtt létrejönnek, és a program végrehajtásának teljes ideje alatt léteznek. A globális változók kezdőértékének adása a főprogram meghívása előtt megtörténik. Abban az esetben, ha nem adunk kezdőértéket egy statikus változónak, akkor nulla értéket fog felvenni a program inicializálását követően.

A globális változókban olyan értékeket, beállításokat érdemes eltárolni, amelyre futás alatt minden függvénynek szüksége lehet. A globális változók függvények közötti kommu-

nikációra is használhatók, ilyen lehet például a megszakítás rutin és a főprogram közti adatcsere. Globális változók használatát lehetőleg kerüljük, mert a sok globális változó használata a programunkat átláthatatlanná teszi.

Ha egy függvényben egy globális változóval megegyező nevű lokális változót deklarálunk, a globális változó nem lesz elérhető, csak a függvényben deklarált lokális változó. Ezt a jelenséget a szakirodalom változó túlterhelésnek hívja.

A 16.11. mintapéldában egy `iGlobalisValtozo` nevű, a forrásállományra globális változót deklarálunk. A mintapélda a főprogramon kívül még egy függvényt is tartalmaz. A globális változó értékét minden a két függvény módosítja. Érdemes a programot lépésenként lefuttatni, és közben az `iGlobalisValtozo` változó értékének változását megfigyelni.

16.11. mintaprogram

```
/* Globális változó deklarációja */
int iGlobalisValtozo = 1;

void fuggveny ( void )
{
    iGlobalisValtozo = 3;
}

int main ( )           // Főprogram kezdete
{
    iGlobalisValtozo = 2;
    fuggveny();
    iGlobalisValtozo = 4;
}
```

16.2.4. volatile előtag

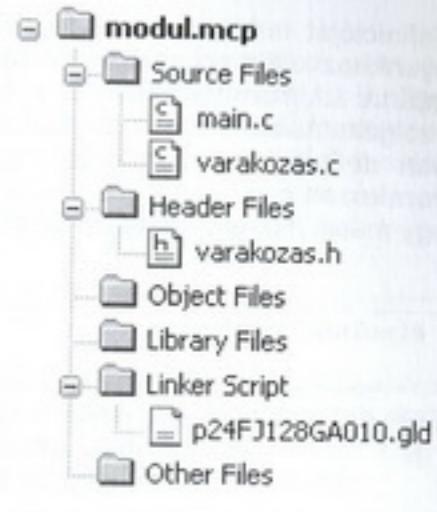
A változók deklarációjánál a `volatile` előtag arra szólítja fel a fordítót, hogy az adott változón ne végezzen optimalizációt. A `volatile` előtagot olyankor érdemes használni globális változók deklarációjánál, amikor a változó értékét egy megszakítás rutin módosítja. Ilyen esetben a fordító a főprogramban az adott változó tesztelését nem fogja optimalizálni, habár úgy tünhet, hogy optimalizálható, hisz a változó értéke nem tud megváltozni a főprogramban.

16.3. TÖBB FORRÁSÁLLOMÁNYBÓL ÁLLÓ PROGRAMOK KÉSZÍTÉSE

Programjaink jobb áttekinthetősége érdekében érdemes a programunkat egy nagy állomány helyett több apró forrásállományban eltárolni, mert így növelhető a programunk átláthatósága. Az egy adott problémával foglalkozó függvényeket érdemes egy forrásállományba elhelyezni, és szükség esetén a projektünkhez hozzáadni.

A több forrásállományból álló programok készítését a következő példán keresztül szereinem bemutatni. A példa a 16.1.2. fejezetben szereplő mintapéldán alapul. Készítünk egy projektet, és a már megszokott módon konfiguráljuk be. Hozzunk létre három üres állományt a következő nevekkel: `main.c`, `varakozas.c`, `varakozas.h`. A frissen létrehozott állományokat adjuk hozzá a projektünkhez. Ha sikeresen hozzáadtuk mindenáron állományukat a projekthez, akkor a projektablakunk a 16.3. ábrához hasonlóan fog kinézni.

16. fejezet: És a kép összeáll...



16.3. ábra

Több forrásállományból álló projekt

16.12. mintaprogram

```
#include <p24fj128ga010.h>
#include "varakozas.h"

_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

int main ( )           // Főprogram kezdete
{
    TRISA = 0xFF00;      // PORTA alsó nyolc lába kimenet lesz.

    while(1)
    {
        LATA = 0x0000;    // PORTA összes bitje nulla lesz.
        varakozas(30000); // Várakozó függvény meghívása.
        LATA = 0x00FF;    // PORTA alsó nyolc bitje egyes lesz.
        varakozas(60000); // Várakozó függvény meghívása.
    }
}
```

Következő lépésben a `varakozas.h` állományba másoljuk be a `varakozas` nevű függvény deklarációját.

16.13. mintaprogram

```
/* Késleltető függvény deklarációja */
void varakozas ( unsigned int uiMaxIdo );
```

A fejlécállományra azért van szükségünk, hogy a `main.c` állomány fordításakor, a `varakozas` függvény hívásánál tisztában legyen a fordító a `varakozas.c` állományban található függvény bemeneti paramétereivel és visszatérési értékével.

Utoljára pedig töltük meg tartalommal a `varakozas.c` állományt, másoljuk bele a `varakozas` függvény definícióját. Ehhez az állományhoz is hozzá kell adni a mikrokontroller-specifikus fejlécállomány behívását, mivel a függvényben szereplő `Nop()` makró

definícióját tartalmazza. A fejlécállományokat azért kell minden egyes C forrásállományunkhoz külön hozzárendelni, mert a C fájlokat egymástól függetlenül fordítja a fordító. A **main.c** állománnyal ellentétben, kihasználva az MPLAB C30 fordítókörnyezet által nyújtott szolgáltatásokat, itt egy általános fejlécállományt hívunk meg. A fejlécállomány a projektben definiált mikrokontroller fejlécállományát fogja meghívni. Ezzel a módszerrel a **varakozas.c** állományunk mikrokontroller-függetlenné vált, mert egy másik projektben, egy másik mikrokontrollerre fordítva ugyanúgy működni fog.

16.14. mintaprogram

```
#include <p24fxxxx.h>

/* Szoftveres késleltető függvény */
void varakozas ( unsigned int uiMaxIdo )
{
    unsigned int uiIdo; // Késleltetéshez használt változó

    for(uiIdo=0; uiIdo<uiMaxIdo; uiIdo++)
    {
        Nop(); // Szoftveres késleltetés:
        // uiMaxIdo-szor fut le a nop utasítás.
    }
}
```

Most már más dolog nem maradt hátra, mind a program lefordítása. A fordító mind a két állományt külön-külön lefordítja, tárgykódot készít belőük, majd a linker a tárgykódokat egy futtatható állománnyá egyesíti.

16.3.1. *extern* előtag

A 16.2.3. fejezetben a globális változók deklarálásáról volt szó. Ott láthattuk, hogyan kell deklarálni egy olyan globális változót, amely az egész forrásállományban látható. A kérdés az, hogyan lehet az ilyen, állományra lokális változók láthatóságát kibővíteni.

Az *extern* előtag segítségével közölni tudjuk a fordítóval, hogy olyan globális változóra szeretnénk hivatkozni, amely egy másik állományban van deklarálva. Abban az esetben, ha az egyik állományban deklarálunk egy globális változót – úgy, ahogy azt a 16.2.3. fejezetben is tettük –, akkor a másik állományban már csak az *extern* kulcsszóval kell rá hivatkoznunk, így ott is az egész állományban láthatóvá válik.

Az egyik állományban, ahol az *iGlobalisValtozo* változót deklaráljuk, a következőt kell beírni:

```
/* Globális változó deklarációja */
int iGlobalisValtozo = 1;
```

A többi állományban, ahol el szeretnénk érni az *iGlobalisValtozo* változót, a következőt kell beírni:

```
/* Külső globális változó deklarációja */
extern int iGlobalisValtozo;
```

Abban az esetben, ha egy másik állományban definiálunk egy függvényt, akkor a függvény deklarációja előre is kiírható az *extern* előtag, de ez nem kötelező, mert alapesetben a függvények deklarációját külsőnek tekinti a fordító. A nemrég elkészített **varakozas.h** állományban szereplő függvénydeklarációt a következőképpen is megadhatjuk:

```
/* Külső késleltető függvény deklarációja */
extern void varakozas ( unsigned int uiMaxIdo );
```

16.3.2. Statikus globális változók

Ha egy globális változó deklarációja előre kiírjuk a *static* előtagot, akkor az a változót csak az öt deklaráló állományba lesz elérhető, más állományok elől rejtve marad. Az így deklarált változót az *extern* kulcsszó segítségével másik forrásállományból nem lehet elérni. A *static* előtag segítségével deklarált változó láthatóság szempontjából a saját állományára lokális változóvá válik.

```
/* Az állományra lokális, azon belül globális változó deklarációja */
static int iGlobalisValtozo = 1;
```

16.3.3. Statikus függvények

Ha egy függvény definíciója (vagy ugyanabban az állományban szereplő deklarációja) előre kiírjuk a *static* előtagot, akkor az a függvény csak az öt definiáló állományban lesz elérhető, más állományok elől rejtve marad. A *static* előtag segítségével definiált függvény láthatóság szempontjából a saját állományára lokális függvényévé válik.

```
static void sajatFuggveny ( void )
{
    /* Ez a függvény másik állományból nem érhető el! */
}
```

16.4. AZ ELŐFORDÍTÓNAK SZÓLÓ UTASÍTÁSOK

Ebben a fejezetrészben az előfordítónak szóló föbb utasításokat foglaljuk össze. Az előfordítónak szóló utasítások a # jellel kezdődnek, és az utasítás végét nem kell pontosvesszővel lezárni.

Fontos megemlíteni, hogy terjedelmi okokból nem foglalkozunk az összes előfordítói utasítással. A könyv által nem részletezett előfordítói utasításokat az adott implementáció fordítói kézikönyvében érdemes megnézni, mert az előfordítói utasítások implementáció-függők lehetnek.

16.4.1. Szimbolikus konstansok és makrók definiálása

Az előfordító lehetőséget biztosít szimbolikus konstansok definiálására. A *#define* utasítással létrehozott szimbolikus konstansokat az előfordító a fordítás megkezdésekor lecseréli az utasítás paramétereiben megadott kifejezésre.

A konstans definiálásának általános alakja a következő:

```
#define szimbolum helyettesítendő szöveg
```

Példa a szimbolikus konstansok használatára:

```
#define TRUE    1
#define FALSE   0
-
int bool;
-
bool = TRUE; // Az fordító bool = 1; -ként fogja fordítani.
```

A *#define* utasítással nemcsak szimbolikus konstansok használatára van lehetőségünk, hanem makrók készítésére is. Makrók esetén a szimbólumban, zárójelek között meg kell adni a makró paramétereit, utána pedig a makró belsejét. A következő példa egy minimumkiválasztó makró definiálását és használatát mutatja be:

```
#define min(a, b) ((a) < (b) ? (a) : (b))
...
z = min(x, y);
// Az fordító valójában a következő utasítás fogja lefordítani:
z = ((x) < (y) ? (x) : (y));
```

A makrók használatával rövid függvényeket lehet kiváltani. A makrók hívása szintaktikailag megegyezik a függvényhívással. Makrók használatánál soha ne tévesszük szem elől azt, hogy valójában nem függvényhívás történik, csak szimbolikus behelyettesítés. A makrók használata rejtegett hibákat hordozhat magában, mint ahogy azt a következő példa is mutatja:

```
z = min(x++, y);
// Az fordító valójában a következő utasítást fogja lefordítani:
z = ((x++) < (y) ? (x++) : (y));
// HIBA: Az x változó feltételtől függően egyvelővel vagy kettővel növekszik!
```

A makrók definiálásakor lehetőségünk van két új operátor használatára is. Az `#` operátor egyoperandusú operátor, és a töle jobbra álló paraméter értékét karakterláncként adja vissza.

Az `##` operátor kétoperandusú operátor, és az öt körülvevő két paramétert egy paraméterről füzi össze. A következő két makródefiníció a két új előfordítói operátor használatát mutatja be.

```
#define STR(x) #x
#define CONC(x, y) x ## y
```

Az előző példában definiált két makró használatakor az `str(100)` makróhívás `*100` karakterláncjal tír vissza, míg a `CONC(x, 6)` makróhívás `x6`-tal tér vissza.

Lehetőségünk van szimbolikus konstans megszüntetésére is, amit az `#undef` utasítással végezhetünk el. Az `#undef` utasítás után a fordító már nem fogja ismerni a visszavont szimbólumot. Az `#undef` utasítás általános alakja a következőképpen néz ki:

```
#undef szimbólum
```

Példa szimbolikus konstans megszüntetésére:

```
#define MAX_ERTEK 100
a = MAX_ERTEK;

#undef MAX_ERTEK;
b = MAX_ERTEK; // Ez a sor fordítási hibát fog okozni,
// mert a MAX_ERTEK nem ismert kifejezés
```

Az Microchip C30 fordító mikrokontroller-specifikus fejlécállományaiban, ahogy arról már a 15.3.5. fejezetben szó esett, a speciális funkcióregiszterek egyes bitjei bitstruktúrákban kerültek definiálásra. Azért, hogy a regiszterek egyes bitjeinek eléréséhez ne kelljen hosszú kifejezéseket használni, a fejlécállományok végén az egyes bitek rövidítéseit is definiálták. A következő sorok a `24FJ128GA010.H` fejlécállományból származnak.

```
#define _TRISA0 TRISAbits.TRISA0
#define _RA0 PORTAbits.RA0
#define _LATA0 LATAbits.LATA0
```

16.4.2. Előre definiált szimbólumok

Az ANSI C szabvány előre definiált fordítói szimbólumokat definiál. Az előre definiált szimbólumok két aláhúzásjellel kezdődnek és végződnek. A következő felsorolás néhány előre definiált szimbólum leírását tartalmazza:

- `_LINE` Egész szám, a forrásállomány aktuális sorszáma.
- `_FILE` Karakterlánc, a forrásállomány neve.
- `_DATE` Karakterlánc, a fordítás kezdetének dátuma.
- `_TIME` Karakterlánc, a fordítás kezdetének időpontja.
- `_STDC` Abban az esetben, ha a szimbólum értéke egyre definiált, akkor a fordító csak az eredeti ANSI C kulcsszavait ismeri fel, ellenkező esetben a szimbólum definíciótlan.

16.4.3. Fejlécállományok betöltése

Az `#include` utasítás már az előző fejezetekben is megjelent. Az `#include` utasítás a paraméterében megadott állomány tartalmát beilleszti az utasítás helyére. Az `#include` utasítás általános alakja a következőképpen néz ki:

```
#include <állománynév>
#include "állománynév"
```

Az első forma használata esetén a fordító az előre definiált, de a projektbeállításoknál módosítható „`include`” könyvtárakban keresi a fejlécállományt, míg a második forma használatakor a projektkönyvtárban teszi ugyanezt.

Minden fordítói környezethoz előre készített függvénykönyvtárak is tartoznak. Az előfordított függvénykönyvtárak függvényeinek deklarációját a könyvtárhoz tartozó fejlécállományok tartalmazzák. A könyvtárak különböző feladatak elvégzésére szolgálnak, mint például a `string.h` állományban deklarált függvények karakterláncok feldolgozását, a `math.h` állományban deklarált függvények bonyolultabb matematikai műveletek használatát, a `stdio.h` állományban deklarált függvények az alapértelmezett kimenet, a soros port használatát segítik elő.

Az egyes implementációkhoz különböző függvénykönyvtárak tartoznak, ezért ezek használatát az adott fordító dokumentációjában érdemes megnézni. A Microchip C30 fordítójához tartozó könyvtárak fejlécállományait a fordító főkönyvtárából nyíló `include` könyvtár tartalmazza. Abban az esetben, ha a könyvtárak forrásait is feltelepítjük, az egyes függvények implementációját is megnézhetjük.

16.4.4. Feltételes fordítás utasításai

Végül nézzük meg a feltételes fordításhoz használható előfordítói utasításokat.

- `#if` feltétel – Abban az esetben, ha a feltétel értéke hamis, azaz nulla, a feltételt követő sorokat a fordító nem veszi figyelembe. Abban az esetben, ha a feltétel értéke igaz, azaz nem nulla, a feltételt követő sorokat a fordító figyelembe veszi.
- `#elif` feltétel – „kölönben-ha” Abban az esetben, ha a kifejezés előtti `#if` vagy `#elif` utasítás hamis volt, a feltételt megvizsgálja, és ha a feltétel értéke hamis, azaz nulla, a feltételt követő sorokat a fordító nem veszi figyelembe. Abban az esetben, ha a feltétel értéke igaz, azaz nem nulla, a feltételt követő sorokat a fordító figyelembe veszi.
- `#else` – Az `#if` vagy `#elif` utasítás hamis ága. Abban az esetben, ha az öt megelőző feltétel igaz volt, az `#else` utasítás utáni sorokat a fordító nem veszi figyelembe. Abban az esetben, ha az öt megelőző feltétel hamis volt, az `#else` utasítás utáni sorokat a fordító figyelembe veszi.
- `#ifdef` szimbólum – Abban az esetben, ha az utasítás paraméterében szereplő szimbólum definiálva van, akkor az utasítás után álló részek fordításra kerülnek, ellenkező esetben kimaradnak a fordításból.

- #ifndef szimbólum – Abban az esetben, ha az utasítás paraméterében szereplő szimbólum nincs definiálva, az utasítás után álló részek fordításra kerülnek, ellenkező esetben kímaradnak a fordításból.
- #endif – Feltételes fordítás lezárása, a fordító az utasítás után álló sorokat feltétel nélkül figyelembe veszi.

Mivel a fejlécállományok is meghívhatnak fejlécállományokat, előfordulhat, hogy egy fejlécállományt a fordító kétszer is betölt. Ilyenkor fordítási hibát kapunk, mert az adott fejlécállományban szereplő deklarációk egy fordítás alatt kétszer is megjelennek. Ennek a hibának a kiküszöbölésére érdemes a saját fejlécállományainkat a következő feltételes fordítási keretben elhelyezni:

```
#ifndef _HEADERFILE_H_
#define _HEADERFILE_H_

/*
 A fejlécállomány belseje
 */

#endif
```

Például a 16.3. fejezetben létrehozott varakozas.h fejlécállomány, az előző feltételes fordítással kiegészítve, így fog kinézni:

16.15. mintaprogram

```
#ifndef _VARAKOZAS_H_
#define _VARAKOZAS_H_

/* Késleltető függvény deklarációja */
void varakozas ( unsigned int uiMaxIdo );

#endif
```

16.4.5. Implementációfüggő utasítások

A #pragma utasítás arra szolgál, hogy implementációfüggő vezérlősorokat helyezzünk el a kódunkba. Mivel az utasítás, ahogy a nevéről is következik, implementációfüggő, a használatát az adott fordító kézikönyvében találjuk meg. A Microchip C30 fordító nem használja ki a #pragma utasítást, de például a Microchip C18 fordító igen, ezért a fordító bemutatásánál majd találunk példát a használatára.

17. AMIKOR TÖBB SZÁLON FUTNAK AZ ESEMÉNYEK...

17.1. TIMER1 IDŐZÍTŐ/SZÁMLÁLÓ MODUL HASZNÁLATA

Vegyük elő a 13.4.1. fejezetben elkészített futófényprogramunkat. A programban egy for ciklust használtunk a futófény késleltetéséhez. Első lépésben cseréljük le a várakozó ciklusunkat. Módosítsuk úgy a programunkat, hogy 250 ms-onként lépjön egyet a futófény állapota. Az ilyen pontos időzítéseket már nehéz megoldani egyszerű várakozó ciklussal, helyette használjuk a Timer1 időzítő modult. A várakozó rész nem egy lefutó ciklus lesz, hanem az időzítő megszakítás bitjére fog várakozni. Amíg a megszakításbit nem érkezik meg, addig a program egy helyben áll. A megszakításbit megérkezése után törölni kell a bitet, majd egygel léptetni a futófény állapotát. Mielőtt a futófényt elindítanánk, a megszakítás modult fel kell konfigurálnunk úgy, hogy a megszakításbit 250 ms-onként érkezzen meg.

Mielőtt a Timer1 modul egyes bitjeit beállítanánk, vegyük szemügyre a modul kialakítását, amelyet a 17.1. ábra mutat!

Az időzítő modul órajelimpulzusát vagy külső, lábról érkező jel biztosítja, vagy a kontroller belső oszcillátorfrekvenciájának a fele. A mikrokontroller lábaira nem csak négy szögjel-generátor kapcsolható, hanem külső oszcillátor is. Igény szerint a számlalandó jel frekvenciáját egy előosztó segítségével csökkenteni tudjuk, hogy nagyobb időzítési idők esetén ne generáljon sok megszakítást az időzítő. Az időzítő az előosztó által leosztott jel hatására növeli a TMR1 16 bites regiszter értékét.

A T1IF megszakításbitet abban az esetben állítja be az időzítő modul egybe, ha a TMR1 regiszter értéke megegyezik a PR1 komparáló regiszter értékével. A megszakításbit generálásával egy időben a TMR1 regiszter értéke automatikusan töröldik. Az időzítő a T1IF bit értékét csak egyesbe tudja állítani, ezért törölni szoftverből kell.

Programunk jobb átláthatóságának érdekében hozzunk létre egy InitT1() függvényt, amelyben az időzítő modult konfiguráljuk fel. Az időzítő modult csak a modul beállítását követően érdemes bekapcsolni, mert így biztos, hogy az első lefutásunk is pontos lesz.

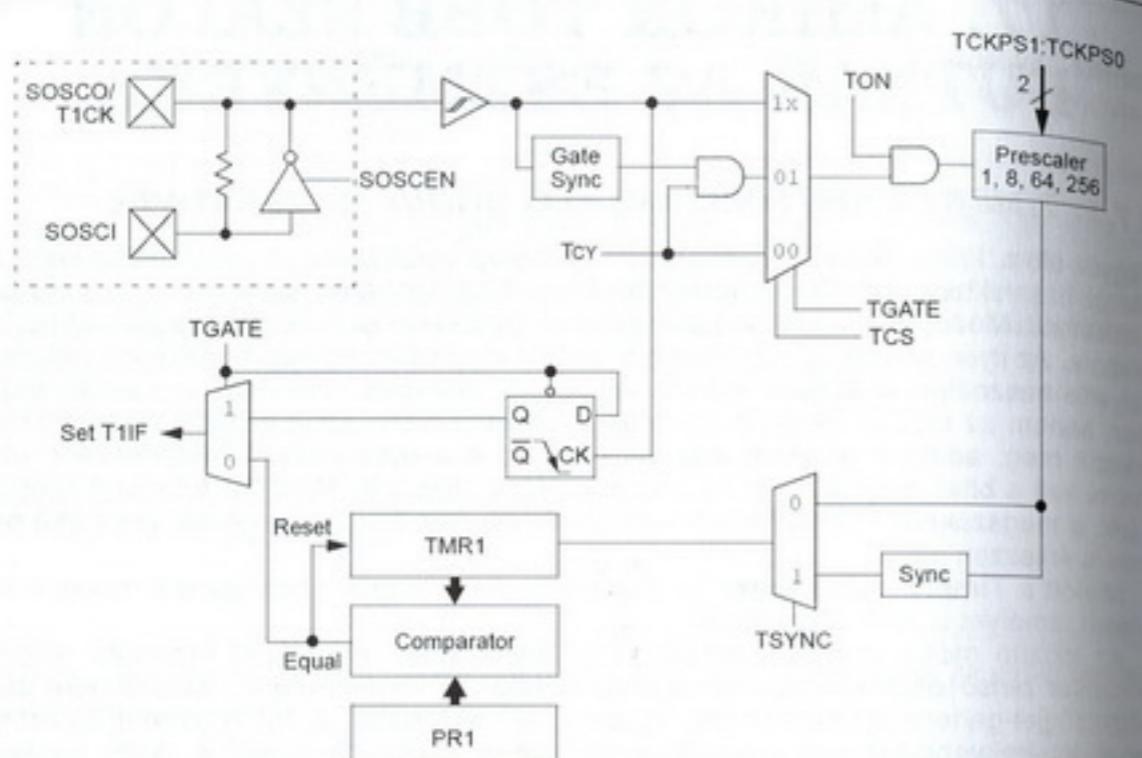
A Timer1 időzítő modult állítsuk be úgy, hogy a kontroller belső oszcillátorfrekvenciájának a felét számolja. 8 MHz oszcillátorfrekvencia esetén érdemes a számlalandó jelet 1:64 arányú előosztóval leosztani, hogy a 250 ms-os időzítő esetén is csak egyszer érje el a komparálási szintet.

A PR1 regiszter értékét a következő képlet segítségével tudjuk kiszámítani:

$$PR1 = \frac{f_{osc}}{m} \cdot T = \frac{2}{64} \cdot 250 \cdot 10^{-3} s = 15625 ,$$

ahol:

f_{osc} = a kontroller oszcillátorának frekvenciája,
 m = az előosztó értéke,
 T = a megszakítás ideje.



17.1. ábra
Timer1 modul felépítése

Az eredeti futófényprogramunkat módosítva a következő programot kapjuk:

17.1. mintaprogram

```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

void InitT1 ( void ); // Timer1 modult inicializáló függvény

/* Főprogram */
int main ( void )
{
    unsigned char chLedek = 1; // futófény állapota

    InitT1(); // Timer1 inicializálása
    TRISA = 0xFF00; // PORTA alsó nyolc lába kimenet lesz.
    LATA = 0x0000; // PORTA törlése

    while(1) // Végtelen ciklus
    {
        if(chLedek & 0x80 // chLedek változó legfelső bitje egyenlő eggyel?
        { // Igaz ág:
            chLedek = 1; // chLedek változó legalsó bitje legyen egy.
        } // a többi nulla
        else
        { // Hamis ág:
            chLedek<<=1; // chLedek változó eggyel balra léptetése
        }
    }
}
```

```
LATA=chLedek; // chLedek változó értékét kitesszük a kimenetre
/* Várakozás */
while(!IFS0bits.T1IF); // Várakozás T1IF megérkezésére
IFS0bits.T1IF = 0; // T1IF törlése
}

/* Timer1 modult inicializáló függvény */
void InitT1 ( void )
{
    T1CONbits.TCS = 0; // Belső oszcillátor használata (Fosc/2)
    T1CONbits.TCKPS = 0b10; // 1:64 frekvenciaosztó használata
    T1CONbits.TGATE = 0; // Kapuzó áramkör letiltása
    T1CONbits.TSIDL = 0; // IDLE állapotban is fusson az áramkör
    TMR1 = 0; // Timer1 számlálójának törlése
    PR1 = 15625; // Időzítési ciklus: 250ms
    IFS0bits.T1IF = 0; // Timer1 megszakításbitjének törlése
    T1CONbits.TON = 1; // Timer1 bekapcsolása
}
```

A frissen elkészített programunk most már pontosan 250 ms-onként lépteti a LED-sor állapotát. Következő lépésként alakítsuk át a programunkat úgy, hogy a LED-sor állapotát ne a főprogram számolja ki, hanem a **Timer1** modulhoz rendelt megszakításrutin.

Természetesen a **T1CON** regisztert nem kötelező bitenként állítani, egyszerre is lehet állítani az értékét. Például az **InitT1()** függvény első négy sora helyett a következő kifejezés használható: **T1CON = 0x0020;**

17.2. MEGSZAKÍTÁSKEZELÉS

Az eddigi programjaink egy szálból álltak, a főprogramból. Ennek a megoldásnak az a hátránya, hogy a programunk egyszerre csak egy dologra tud figyelni, így párhuzamos eseményeket nem tud feldolgozni.

A kontrollerek erőssége a megszakítások kezelése. A processzor környékére integrált periferiák külső vagy belső hardveresemények hatására megszakítást kezdeményezhetnek. A megszakítás-kezdeményezés elfogadása után a program fő szála, a főprogram megszakad, és az adott megszakításkezelő rutinra kerül a programvégrehajtás.

A megszakításrutinoknak rövideknek és gyorsaknak kell lenniük, hogy a főprogram futását ne zavarják meg. Azért, hogy a főprogram „ne vegye észre” a főprogram megszakítását, a megszakító rutinnak el kell mentenie a processzor állapotát leíró regisztereit, majd a megszakító rutin végén vissza kell állítania. A fordító átveszi ezt a feladatot a programozótól, mert a megszakítások által meghívott rutinokat automatikusan kiegészít a mentésekkel és a visszaállításokat végző programkóddal.

17.2.1. A PIC24F család megszakításvezérlője

Ahogy már a könyvben olvasni lehetett róla, a PIC 16 bites mikrokontrollerei megszakításvektor-táblát tartalmaznak. minden egyes megszakításhoz hozzárendelhető az általunk írt megszakításrutiin kezdőcíme. Jobban mondva a mikrokontroller két megszakításvektor-táblával rendelkezik, mert egy alternatív ugrótábla létrehozására is lehetőségünk van. Az alternatív ugrótáblára az **INTCON2** regiszterben található **ALTI** bit segítségével váthatunk át.

A processzorhoz és minden egyes megszakításforráshoz egyedi prioritásszint rendelhető. Az egyes megszakításforrások és a processzor prioritásszintjét három bit segítségével 0 és 7 közé lehet beállítani. A processzort vagy egy másik megszakítást az a megszakításforrás tudja megszakítani, melynek nagyobb a prioritásszintje. Ebből következik, hogy ha a processzor szintje hetesre van állítva, akkor egy megszakítás sem tudja megszakítani. Ha egy megszakításforrás szintjét nullára állítjuk, akkor gyakorlatilag kikapcsoljuk az adott megszakításforrást, hisz nem tudja a processzort megszakítani. A mikrokontroller reset feltétele után az összes megszakítás prioritásszintje négy, a processzor prioritásszintje nulla értéket vesz fel.

Minden egyes megszakításforráshoz tartozik egy jelzőbit (*Interrupt Flag*) és egy engedélyező bit is. Abban az esetben, ha az adott megszakításhoz tartozó engedélyező bit értéke nulla, az adott megszakítás ki van kapcsolva.

A 17.2. ábra táblázata a PIC24FJ128GA010 mikrokontrollerben implementált megszakításokat fogalja össze. A táblázat az előbb említett mikrokontroller felhasználói kézikönyvből származik. A táblázat tartalmazza az egyes megszakításforráshoz rendelt ugróímeket és az adott megszakításforráshoz tartozó konfigurációs biteket is.

| Megszakítás-forrás | Vektor sorszáma | Megszakításvektor címe | Alternatív megszakításvektor címe | Megszakításhoz tartozó bitek | | |
|-------------------------|-----------------|------------------------|-----------------------------------|------------------------------|-------------|--------------|
| | | | | Jelző | Engedélyező | Prioritás |
| ADC1ConversionDone | 13 | 00002Eh | 00012Eh | IFS0<13> | IEC0<13> | IPC3<6:4> |
| ComparatorEvent | 18 | 000038h | 000138h | IFS1<2> | IEC1<2> | IPC4<10:8> |
| CRCGenerator | 67 | 00009Ah | 00019Ah | IFS4<3> | IEC4<3> | IPC16<14:12> |
| ExternalInterrupt0 | 0 | 000014h | 000114h | IFS0<0> | IEC0<0> | IPC0<2:0> |
| ExternalInterrupt1 | 20 | 00003Ch | 00013Ch | IFS1<4> | IEC1<4> | IPC5<2:0> |
| ExternalInterrupt2 | 29 | 00004Eh | 00014Eh | IFS1<13> | IEC1<13> | IPC7<6:4> |
| ExternalInterrupt3 | 53 | 00007Eh | 00017Eh | IFS3<5> | IEC3<5> | IPC13<6:4> |
| ExternalInterrupt4 | 54 | 000080h | 000180h | IFS3<6> | IEC3<6> | IPC13<10:8> |
| I2C1MasterEvent | 17 | 000036h | 000136h | IFS1<1> | IEC1<1> | IPC4<6:4> |
| I2C1SlaveEvent | 16 | 000034h | 000034h | IFS1<0> | IEC1<0> | IPC4<2:0> |
| I2C2MasterEvent | 50 | 000078h | 000178h | IFS3<2> | IEC3<2> | IPC12<10:8> |
| I2C2SlaveEvent | 49 | 000076h | 000176h | IFS3<1> | IEC3<1> | IPC12<6:4> |
| InputCapture1 | 1 | 000016h | 000116h | IFS0<1> | IEC0<1> | IPC0<6:4> |
| InputCapture2 | 5 | 00001Eh | 00011Eh | IFS0<5> | IEC0<5> | IPC1<6:4> |
| InputCapture3 | 37 | 00005Eh | 00015Eh | IFS2<5> | IEC2<5> | IPC9<6:4> |
| InputCapture4 | 38 | 000060h | 000160h | IFS2<6> | IEC2<6> | IPC9<10:8> |
| InputCapture5 | 39 | 000062h | 000162h | IFS2<7> | IEC2<7> | IPC9<14:12> |
| InputChangeNotification | 19 | 00003Ah | 00013Ah | IFS1<3> | IEC1<3> | IPC4<14:12> |
| OutputCompare1 | 2 | 000018h | 000118h | IFS0<2> | IEC0<2> | IPC0<10:8> |

17. fejezet: Amikor több szálon futnak az események...

| | | | | | | |
|------------------------|----|---------|---------|----------|----------|-------------|
| OutputCompare2 | 6 | 000020h | 000120h | IFS0<6> | IEC0<6> | IPC1<10:8> |
| OutputCompare3 | 25 | 000046h | 000146h | IFS1<9> | IEC1<9> | IPC6<6:4> |
| OutputCompare4 | 26 | 000048h | 000148h | IFS1<10> | IEC1<10> | IPC6<10:8> |
| OutputCompare5 | 41 | 000066h | 000166h | IFS2<9> | IEC2<9> | IPC10<6:4> |
| ParallelMasterPort | 45 | 00006Eh | 00016Eh | IFS2<13> | IEC2<13> | IPC11<6:4> |
| RealTimeClock/Calendar | 62 | 000090h | 000190h | IFS3<14> | IEC3<13> | IPC15<10:8> |
| SPI1Error | 9 | 000026h | 000126h | IFS0<9> | IEC0<9> | IPC2<6:4> |
| SPI1Event | 10 | 000028h | 000128h | IFS0<10> | IEC0<10> | IPC2<10:8> |
| SPI2Error | 32 | 000054h | 000154h | IFS2<0> | IEC0<0> | IPC8<2:0> |
| SPI2Event | 33 | 000056h | 000156h | IFS2<1> | IEC2<1> | IPC8<6:4> |
| Timer1 | 3 | 00001Ah | 00011Ah | IFS0<3> | IEC0<3> | IPC0<14:12> |
| Timer2 | 7 | 000022h | 000122h | IFS0<7> | IEC0<7> | IPC1<14:12> |
| Timer3 | 8 | 000024h | 000124h | IFS0<8> | IEC0<8> | IPC2<2:0> |
| Timer4 | 27 | 00004Ah | 00014Ah | IFS1<11> | IEC1<11> | IPC6<14:12> |
| Timer5 | 28 | 00004Ch | 00014Ch | IFS1<12> | IEC1<12> | IPC7<2:0> |
| UART1Error | 65 | 000096h | 000196h | IFS4<1> | IEC4<1> | IPC16<6:4> |
| UART1Receiver | 11 | 00002Ah | 00012Ah | IFS0<11> | IEC0<11> | IPC2<14:12> |
| UART1Transmitter | 12 | 00002Ch | 00012Ch | IFS0<12> | IEC0<12> | IPC3<2:0> |
| UART2Error | 66 | 000098h | 000198h | IFS4<2> | IEC4<2> | IPC16<10:8> |
| UART2Receiver | 30 | 000050h | 000150h | IFS1<14> | IEC1<14> | IPC7<10:8> |
| UART2Transmitter | 31 | 000052h | 000152h | IFS1<15> | IEC1<15> | IPC7<14:12> |

17.2. ábra
A PIC24FJ128GA010 mikrokontrollerben megvalósított megszakítások összefoglalása

A mikrokontroller architektúrája nyolc darab nem maszkolható (nem kikapcsolható) megszakítás (angolul **trap**) kialakítására nyújt lehetőséget. Jelenleg a nyolc **trap**-ból négy került kialakításra. Ezeket a megszakításokat nem lehet kikapcsolni, a megszakításforrásuk aktivizálódása után mindenkorábban meghívásra kerülnek. Abban az esetben, ha nem használjuk ki a **trap** megszakításcímeket, akkor a fordító automatikusan egy `reset` utasítást ír be a megszakításrutin helyére. A **trap** megszakításokat olyan hardver- vagy szoftverhibák generálnak, melyek a kontrollerben futó program működését drasztikusan befolyásolják. Az implementált négy **trap** forrása a következő lehet:

- oszcillátorhiba
- címzési hiba
- stackhiba
- matematikai hiba (pl. nullával való osztás).

A 17.3. ábra a PIC24FJ128GA010 mikrokontrollerben implementált, nem maszkolható megszakításokat fogalja össze. A táblázat, az előbbi táblázathoz hasonlóan, a mikrokontroller felhasználói kézikönyvből származik.

| Vektor sor-száma | Megszakítási vektor címe | Alternatív megszakítási vektor címe | Trap forrása |
|------------------|--------------------------|-------------------------------------|-------------------|
| 0 | 000004h | 000104h | Reserved |
| 1 | 000006h | 000106h | OscillatorFailure |
| 2 | 000008h | 000108h | AddressError |
| 3 | 00000Ah | 00010Ah | StackError |
| 4 | 00000Ch | 00010Ch | MathError |
| 5 | 0000Eh | 00010Eh | Reserved |
| 6 | 000010h | 000110h | Reserved |
| 7 | 000012h | 0001172h | Reserved |

17.3. ábra

Nem maszkolható megszakítások összefoglalása

Érdemes még megemlíteni az INTCON1 regiszter NSTDIS bitjét. Ezzel a bittel kikapcsolható a megszakítások egybeágyazása. Abban az esetben, ha az NSTDIS bit értéke egyre van állítva, egy magasabb prioritással rendelkező megszakítás nem tudja megszakítani az alacsonyabb szinten lévő megszakításhoz tartozó megszakítás rutin végrehajtását. Ilyen esetben a megszakítások egymás után kerülnek végrehajtásra, az egyes prioritások csak az egyszerre megérkező megszakításkérelmek közötti végrehajtási sorrendet határozzák meg.

17.3. A MEGSZAKÍTÁSRUTIN ELKÉSZITÉSE

Most, hogy áttekintettük az általunk használt kontroller megszakításrendszerét, készítsük el a Timer1 modulhoz tartozó megszakítás rutint, és ágyazzuk bele az előző programunkba.

Egy adott megszakításhoz a megszakítás nevéből képzett függvény segítségével tudunk hozzárendelni egy megszakítás rutint. A megszakítás rutin neve az aláhúzásjellel (_) kezdődik, majd a megszakítás forrásnevével folytatódik, és az „Interrupt” szóval záródik. A megszakítás rutinok nevét az adott mikrokontrollerhez tartozó gld kiterjesztésű állomány tartalmazza. A megszakítás rutinok nem rendelkeznek visszatérési értékkel és bemenő paraméterrel sem. A megszakítás rutinokat nem a főprogram hívja meg, hanem a mikrokontroller megszakításvezérlője. Azt, hogy az adott függvény megszakítás rutin lesz, a fordítónak az __attribute__ ((interrupt)) előfordítói utasítás segítségével tudjuk megadni. A következő kód a Timer1 időzítőhöz hozzárendelt megszakítás rutin definíciója:

```
void __attribute__ (( interrupt)) _T1Interrupt ( void )
{
    // Megszakítás rutin belseje
}
```

Hogy ne kelljen az __attribute__ ((interrupt)) kifejezést minden megszakítás rutin előre kiírni, a mikrokontrollerhez tartozó fejlécállományban definíálásra került egy __ISR makró, ami az előző kifejezés használatát váltja ki. Az előbb említett makró használatával a megszakítás rutin definíciója már átláthatóbbá válik:

```
void __ISR _T1Interrupt ( void )
{
    // Megszakítás rutin belseje
}
```

17. fejezet: Amikor több szálon futnak az események...

A mikrokontroller assembly nyelvében lehetőségünk van arra, hogy megszakítást kiszolgáló rutinban a w0-w5 és az SR regiszter tartalmát egy utasítással (push.s) mentük az árnyék- (shadow) regiszterekbe. Mivel az árnyékregiszterek maximum egy mentés tárolására alkalmasak, ezért ezeket csak egy megszakításforrás esetén célszerű használni. Ha azt szeretnénk, hogy a megszakítás rutinunk az árnyékregiszterek felhasználásával végezze el a mentést és a visszaállítást, akkor az __ISR makró helyett az __ISRFast makró kell használnunk.

Abban az esetben, ha az alternatív megszakítás táblába szeretnénk egy megszakítást kiszolgáló rutint beregisztrálni, akkor a megszakítás rutin nevét az __Alt előtaggal kell kezdeni. Például a Timer1 időzítőhöz hozzárendelt alternatív megszakítás rutin neve: __AltT1Interrupt

Ne felejtük el, hogy a megszakítás jelző biteket a hardver csak egybe tudja állítani, törleszürkről a programozónak kell gondoskodnia. Figyelembe véve az előző mondatot is, az üres megszakítás rutinunk a következőképpen alakul:

```
void __ISR _T1Interrupt ( void )
{
    // Megszakítás rutin belseje
    IFS0bits.T1IF = 0;           // Timer1 megszakítás bitjének törlése
}
```

A következő mintapélda a fejezet elején elkészített futófény program módositott változata. Most a LED-sor állapotát a megszakítás rutin fogja kiszámolni, a főprogramban már csak egy végtelen ciklus lesz, ami a chLedek változó értékét kihelyezi az A portra. Mivel a megszakítás rutin, amely beállítja a LED-sor állapotát, időközönként automatikusan meghívódik, ezért a főprogramban már nincs szükség várakozó ciklusra. Azért, hogy a megszakítás rutin és a főprogram kommunikálni tudjon egymással, a chLedek változót globálisra kell állítani. A volatile előtag használatára azért van szükség, mert a változó értéke a főprogram futása alatt is meg tud változni, mert a megszakítás rutin változtatja meg a változó értékét. Az InitInterrupt() függvény a Timer1 időzítő megszakítását engedélyezi, és a processzor és a Timer1 időzítő prioritásszintjét is beállítja.

17.2. mintaprogram

```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

/* Globális változó, futófény állapotát tárolja */
volatile unsigned char chLedek = 1;

void InitT1 ( void );           // Timer1 modult inicializáló függvény
void InitInterrupt ( void );   // Megszakítások konfigurálása

/* Főprogram */
int main ( void )
{
    InitT1();                   // Timer1 inicializálása
    InitInterrupt();            // Megszakítások konfigurálása
}
```

```

TRISA = 0xFF00;           // PORTA alsó nyolc lába kimenet lesz.
LATA = 0x0000;           // PORTA törlése

while(1)                 // Végtelen ciklus
{
    LATA=chLedek;        // chLedek változó értékét kitesszük a
kimenetre
}

/* Timer1 modult inicializáló függvény */
void InitT1 ( void )
{
    T1CONbits.TCS = 0;      // Belső oszcillátor használata (Fosc/2)
    T1CONbits.TCKPS = 0b10;  // 1:64 frekvenciaosztó használata
    T1CONbits.TGATE = 0;     // Kapuzó áramkör letiltása
    T1CONbits.TSIDL = 0;     // IDLE állapotban is fussen az áramkör
    TMR1 = 0;                // Timer1 számlálójának törlése
    PR1 = 15625;             // Időzítési ciklus: 250ms
    IFS0bits.T1IF = 0;       // Timer1 megszakításbitjének törlése
    T1CONbits.TON = 1;       // Timer1 bekapsolása
}

/* Megszakítások konfigurálását végző függvény */
void InitInterrupt ( void )
{
    SRbits.IPL = 0;          // Processzor prioritásszintjének beállítása
    IPC0bits.T1IP = 4;        // Timer1 megszakítás prioritásszintjének
                            // beállítása
    IEC0bits.T1IE = 1;        // Timer1 megszakításának engedélyezése
}

/* Timer1 megszakítás rutinja */
void _ISR _T1Interrupt ( void )
{
    if(chLedek & 0x80)      // chLedek változó legfelső bitje egyenlő eggyel?
    {
        // Igaz ág:
        chLedek = 1;          // chLedek változó legalsó bitje legyen egy,
        // a többi nulla
    }
    else
    {
        // Hamis ág:
        chLedek<<=1;         // chLedek változó eggyel balra léptetése
    }

    IFS0bits.T1IF = 0;        // Timer1 megszakításbitjének törlése
}

```

17.4. ENERGIATAKARÉKOS ÜZEMMÓDOK HASZNÁLATA

A 17.3. fejezetben elkészített programunk már megszakítást használ, de a főprogramban lévő ciklus túl sokszor fut le a LED-sor állapotának változásához képest, ezzel növelte a mikrokontroller energiafogyasztását. Meg lehet valahogyan oldani, hogy csak akkor dolgozzon a mikrokontroller, amikor feladata van, más esetben pihenjen, így csökkentve a saját energiafogyasztását?

Ahogy arról a könyv architekturális részében már szó esett, a Microchip PIC 16 bites mikrokontrollerek négyfajta energia-üzemmódban képesek dolgozni.

- **Power On** állapot – Ilyenkor a mikrokontroller maximális sebességen dolgozik, a bekapcsolt perifériák is működnek. Ebben az állapotban a legnagyobb a mikrokontroller energiafogyasztása.
- **Idle** üzemmód – A mikrokontroller minden egyes perifériájához tartozik egy **Idle** üzemmód bit, amellyel megadhatjuk, hogy egy periféria **Idle** üzemmódban dolgozzon, vagy ki legyen kapcsolva. **Idle** üzemmód alatt a processzor leáll, csak az **Idle** üzemmódban engedélyezett perifériák működnek. A mikrokontrollert a mikrokontrollerhez tartozó fejlécállományában definiált **Idle()**; makró segítségével tudjuk **Idle** üzemmódba helyezni.
- **Sleep** üzemmód – A mikrokontrollernek **Sleep** üzemmódban van a legkisebb energiafogyasztása. **Sleep** üzemmódban a processzor és a perifériák is kikapcsolnak, kivéve azok a perifériák, amelyek saját, belső oszcillátorral rendelkeznek, mint például az A/D konverter. A mikrokontrollert a mikrokontrollerhez tartozó fejlécállományában definiált **Sleep()**; makró segítségével tudjuk **Sleep** üzemmódba helyezni.
- **Doze** üzemmód – Egy frekvenciaosztó segítségével lehetőségünk nyílik a processzort a perifériákhoz képest alacsonyabb frekvencián üzemeltetni. Ilyen esetben lassabb lesz az utasítás-végrehajtás, de kisebb lesz a mikrokontroller energiefelvételle. A **Doze** üzemmód engedélyezését és a frekvenciaosztó értékét a **CLKDIV** regiszterben lehet beállítani.

A mikrokontroller külső lábainak állapota **Sleep** és **Idle** üzemmódban nem változik meg. A mikrokontrollert az engedélyezett megszakítások, a **Watch Dog Timer** túlcsordulása vagy **Reset** feltétel teljesülése tudja a **Sleep** és **Idle** üzemmódból felébreszteni. **Sleep** és **Idle** üzemmódba kapcsoláskor a **Watch Dog Timer** számláló értéke automatikusan törlödik.

A **Watch Dog Timer** értékét a **ClrWdt()**; makró segítségével tudjuk törölni a programról.

A futófényprogramunkban úgy konfiguráltuk be a **Timer1** időzítőt az **InitT1()** függvényben a **T1CONbits.TSIDL = 0;** utasítás segítségével, hogy **Idle** üzemmód alatt is dolgozzon, ezért a processzort a főprogramban nyugodtan „el lehet altatni”, mert az időzítő túlcsordulására fel fog ébredni. Ha a főprogram **while** ciklusában a processzort minden **Idle** üzemmódba helyezzük, akkor csak annyi ideig fog futni, amíg módosítja a LED-sor állapotát, majd újból „elalaszik”. A módosított főprogram **while** ciklusa a következőképpen alakul:

17.3. mintaprogram

```

while(1)                 // Végtelen ciklus
{
    LATA=chLedek;        // chLedek változó értékét kitesszük a kimenetre
    Idle();                // A mikrokontroller Idle üzemmódba kapcsolása
}

```

17.5. INLINE ASSEMBLY

Programírás közben előfordulhat olyan eset, amikor rövid assembly kódot kell beillesztenünk a programunkba. Tipikusan ilyen eset, amikor feloldó assembly kódot kell a programunkban elhelyezni, mint például az **OSCCON** regiszter írása előtt. De olyan eset is előfordulhat, amikor a C nyelvben bonyolultabban elkészíthető algoritmust váltunk ki egyszerűbb assembly utasításokkal. Már mi is használtuk a fordító inline assembly funkcióját, igaz, el volt előlünk rejte. Ilyen volt például az **Idle()** vagy a **Nop()** makró használata. A **Nop()** makró például egy **nop** assembly utasítást helyez el a kódunkba. Ha kikeressük a mikrokontroller fejlécállományában a **Nop()** makró definícióját, akkor a következő sort találjuk:

```
#define Nop()  __asm__ volatile ("nop");
```

Az assembly utasítást `_asm_` előtaggal kell kezdeni, majd paraméterként a beiljesztendő assembly utasítást kell írni. Az `_asm_` utasítás általános alakja a következő, képpen néz ki:

```
_asm_ ("utasítás");
```

Most újból módosítuk a futófényprogramunkat. Az időzítő megszakítás rutinjában a `chLedek` változó új értékét egy `if-else` szerkezet segítségével számoljuk ki. Erre a megoldásra azért volt szükség, mert a nyelv nem rendelkezik rotáló funkcióval, csak eltolával. A bináris rotálás és eltolás közötti különbség az, hogy rotálás esetén a felszabaduló bit helyére a másik oldalon kiesett bit kerül, amíg eltolás esetén nulla vagy az előjelbit (lásd 13.3. fejezet). Mivel a futófénynél pont arra van szükségünk, hogy az egyik oldalon kiesett bit a másik oldalon megjelenjen, az `if-else` szerkezetet egy `rlnc.b` assembly utasítással ki tudjuk váltani.

Most már csak egy dologra van szükségünk: Hogyan érjük el a `chLedek` változót az assembly utasításban? Mivel a `chLedek` változó globális, aláhúzás jellel kiegészítve (`_chLedek`) elérhető az assembly kódban is.

Módosítuk a `_T1Interrupt` megszakítás rutinját úgy, hogy az `if-else` szerkezetet cseréljük le az `rlnc.b` assembly utasításra. Az így kapott kód jóval rövidebb és gyorsabb lesz.

17.4. mintaprogram

```
/* Timer1 megszakítás rutinja */
void _ISR _T1Interrupt ( void )
{
    _asm_ ("rlnc.b _chLedek"); // chLedek változó eggyel balra rotálása
    IFS0bits.T1IF = 0;          // Timer1 megszakításbitjének törlése
}
```

Ha egy változónk lokális, akkor is el tudjuk érni a beágyazott assembly utasításokban is, de ilyenkor már kiterjesztett inline assembly utasítást kell használnunk. Terjedelmi okokból az alábbi kis példában nem részletezzük az inline assembly utasítás összes paraméterét. A teljes leírás megtalálható a Microchip C30 fordító felhasználói kézikönyvének kilencedik fejezetében.

Nézzük, hogyan tudjuk megoldani egy lokális változó balra rotálását. Ilyenkor meg kell adni az utasítás után, hogy milyen változón szeretnénk elvégezni a műveletet. A változó típusát, illetve hogy be- vagy kimeneti érték, azt a változó előtt álló karakterek írják le. Általános esetben a `chLedek` változó rotálását a következő utasítással lehet megadni:

```
_asm_ volatile ("rlnc.b %0, %0" : "+r" (chLedek));
```

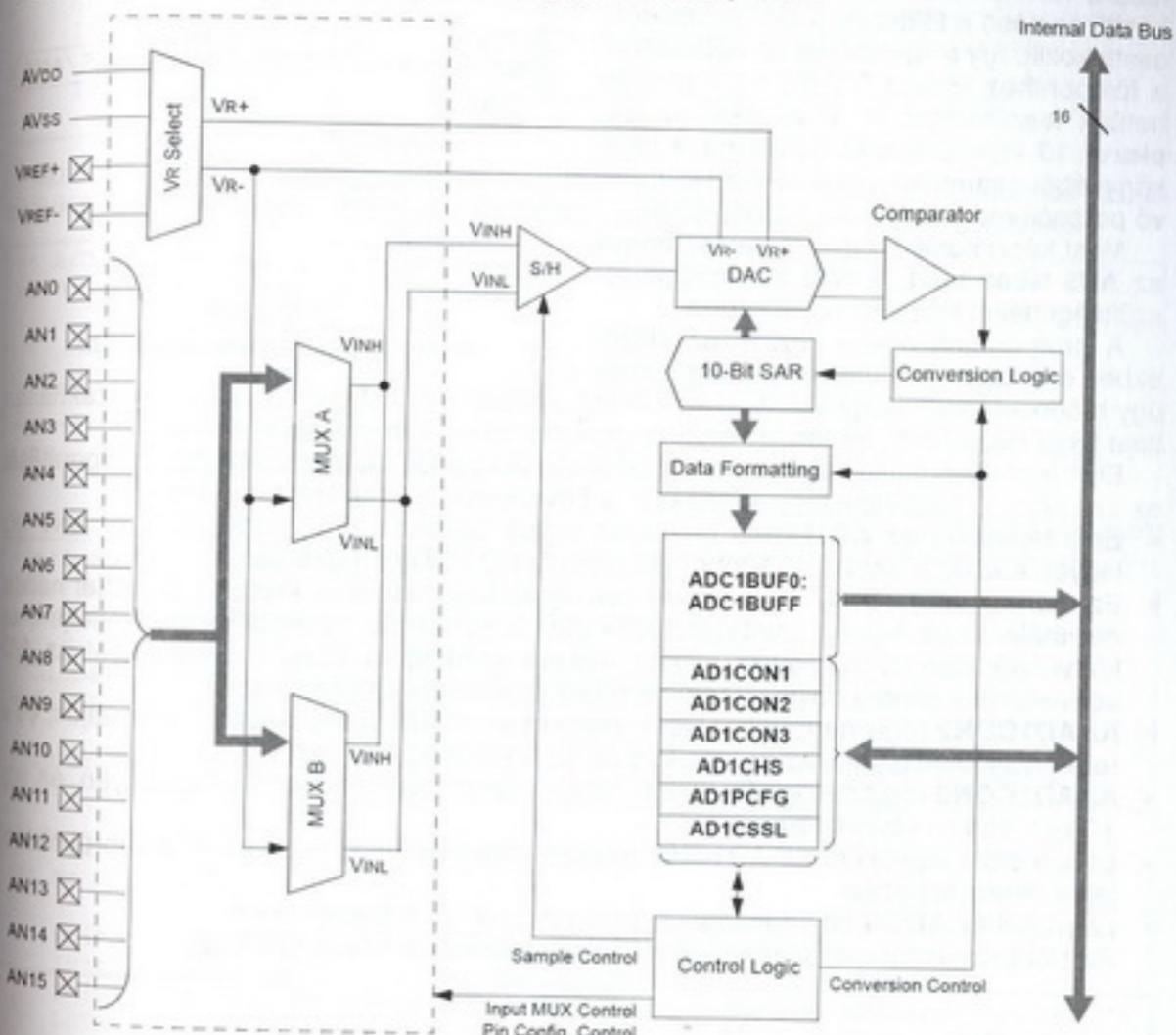
Az utasítás hatására a `chLedek` változó értékét bemozgatja a `w0` regiszterbe, majd a rotálás után a `w0` regiszter értékét visszamásolja a `chLedek` változóba.

17.6. ANALÓG-DIGITÁLIS ÁTALAKÍTÓ HASZNÁLATA

Az eddig használt bemeneteink csak digitálisak voltak, értékük nulla vagy egy lehetett. Most ismerkedjünk meg a mikrokontroller analóg-digitális átalakítójának használatával. Az analóg-digitális átalakítók működésével a könyv *Analóg perifériák* című fejezete foglalkozik, ezért mielőtt nekiállunk az analóg-digitális átalakító beállításának, érdemes az előbb említett fejezetet átolvassni.

A PIC24FJ128GA010 mikrokontrollerben egy 10 bites analóg-digitális átalakító modul van. A modul bemenetére két analóg multiplexer segítségével lehet ráhelyezni a mérim kívánt analóg feszültséget. A **MUX A** elnevezésű multiplexer segítségével a mintavételező

áramkör magas (V_{INH}) bemeneti szintjét tudjuk kiválasztani, míg a **MUX B** elnevezésű multiplexer segítségével a mintavételező áramkör alacsony (V_{INL}) bemeneti szintjét tudjuk kiválasztani. A két multiplexer a 16 bemeneti csatornán kívül a negatív referenciafeszültséget is ki tudja választani. Ezzel a megoldással lehetőségünk van a bemeneti csatornák feszültségszintjét a negatív referenciahoz vagy egy másik bemeneti csatornához képest méni. Az analóg-digitális átalakító referenciafeszültsége vagy külső referenciajel (V_{REF+} , V_{REF-}), vagy az analóg föld és tápfeszültség (AV_{DD} , AV_{SS}) lehet.



17.4. ábra
Az analóg-digitális átalakító felépítése

A 17.4. ábra a PIC24FJ128GA010 mikrokontroller adatlapjából származik, és a beépített analóg-digitális átalakító felépítését mutatja.

Az analóg-digitális átalakító működését az **AD1CON1**, **AD1CON2**, **AD1CON3** regiszter segítségével lehet beállítani. Az **AD1CHS** regiszter segítségével a két analóg multiplexer állapota konfigurálható. Azt, hogy az egyes bemeneti lábak analóg vagy digitális lábként működjenek, az **AD1PCFG** regiszter segítségével tudjuk beállítani. Programjaink írásakor minden vegyük figyelembe, hogy az analóg bemeneti funkciót is ellátó lábak a reset feltétel után analóg bemenetként működnek, és nem digitálisként! Az analóg-digitális modul képes akár minden a 16 bemenetét egymás után automatikusan végigpásztázni. Azt, hogy

melyik bemenetek vegyenek részt az automatikus pásztázásban, az **AD1CSSL** regiszter segítségével lehet megadni. Az egyes mérések eredményei, amelyek 10 biten vannak ábrázolva, az **ADC1BUFO–ADC1BUFF**-ig terjedő 16 regiszterben kerülnek tárolásra.

Az **Explorer 16** fejlesztőpanelen található egy potenciometter. A potenciometter középső kivezetése a mikrokontroller **AN5** lábára van rátöltve. A potenciometter másik két kivezetése a földre és a tápfeszültségre csatlakozik, így forgatásával az **AN5** lábon, a földponthoz képest 0 V-tól 3,3 V-ig mérhetünk feszültséget. A 17.5. ábra az **Explorer 16** fejlesztőpanel felhasználói kézikönyvből származik, és az **AN5** lábon lévő potenciometter bekötését szemlélteti.

Most készítünk el egy programot, amely az **AN5** lábon mér, 8 bitre lecsomkított feszültségértékét kihelyezi a LED-sorra.

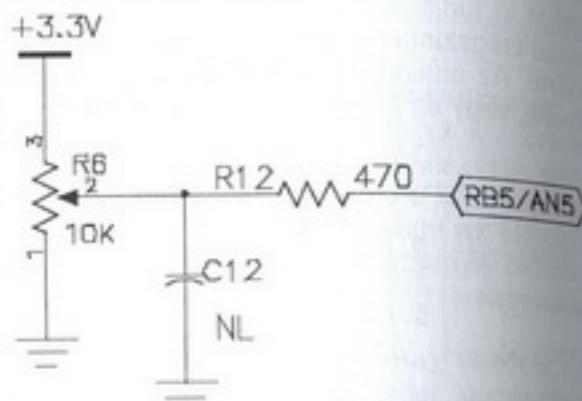
A programunk alapja legyen az előzőekben elkészített futófényünk. A programunk úgy módosul majd, hogy a **chLedek** változó értékét nem a Timer1 modul megszakítás rutinja fogja kiszámolni, hanem az analóg–digitális átalakító megszakítás rutinja.

Első lépésként állítsuk be az analóg–digitális átalakítót. Az A/D átalakító konfigurálását az **InitAD1()** függvény fogja elvégezni, a következő lépések segítségével:

- Első lépésként az **AD1PCFG** regisztert fogjuk beállítani úgy, hogy csak az **AN5** láb legyen analóg a bemenet, a többi láb digitális I/O lábként működjön.
- Ez után az **AD1CON1** regiszterben beállítjuk, hogy a mérés eredménye előjel nélküli decimális szám legyen, mintavételezés után a konverzió automatikusan elinduljon, és konverzió után pedig egy új mintavételezés kezdődjön. Ezzel a beállítással az A/D konverterünk minden dolgozni fog, nem kell szoftverből újraindítanunk.
- Az **AD1CON2** regiszterben beállítjuk, hogy csak a **MUX A** multiplexert használjuk, és a referenciafeszültségek az analóg föld és tápfeszültség legyenek.
- Az **AD1CON3** regiszterben beállítjuk, hogy a mintavételezési idő 31 T_{AD} legyen, és egy 1 T_{AD} = 125 ns ideig tartson.
- Utolsó előtti lépésként az **AD1CHS** regiszterben beállítjuk, hogy az **AN5** lábról történjen a mintavételezés.
- Legvégül az **ADON** bit segítségével bekapsoljuk az A/D konvertort.

Az előbbi beállításokat elvégző függvény forráskódja a következő lesz:

```
/* Analóg Digitális Konvertort inicializáló függvény */
void InitAD1 ( void )
{
    AD1PCFG = 0xFFDF; // AN5 analóg bemenet engedélyezése,
                       // a többi bemenet digitális I/O lesz
    AD1CON1 = 0x00E4; // Automatikus konverzió indítása
                       // mintavételezés után, konverzió után
                       // új mintavételezés indítása
    AD1CON2 = 0; // MUXA használata, a Vref+/- feszültségekkel
                  // a AVss és a AVdd használata.
    AD1CON3 = 0x1F00; // Tsamp = 31 x Tad; Tad=125ns (Tad = 1 Tcy)
    AD1CHS = 5; // AN5 bemenet kiválasztása
    AD1CON1bits.ADON = 1; // A/D konverter bekapsolása
}
```



17.5. ábra
Az R6 potenciometter bekötése

17. fejezet: Amikor több szálön futnak az események...

A következő lépésben módosítsuk a megszakítások beállítását végző **InitInterrupt()** függvényt is. A függvényben megadjuk az A/D konverter megszakításbitjének prioritását, és engedélyezzük a megszakítást.

```
/* Megszakítások konfigurálását végző függvény */
void InitInterrupt ( void )
{
    SRbits.IPL = 0; // Processzor prioritásszintjének beállítása
    IPC3bits.AD1IP = 3; // AD1 megszakítás prioritásszintjének
                         // beállítása
    IEC0bits.AD1IE = 1; // AD1 megszakításának engedélyezése
}
```

Most már csak a megszakítás rutin megírása van hátra. A megszakítás rutinban kiszámlujuk a **chLedek** változó új értékét, és töröljük az A/D konverter megszakításbitjét.

```
/* A/D konverter megszakítás rutinja */
void _ISR _AD1Interrupt ( void )
{
    chLedek = ADC1BUFO >> 2; // ADC felső nyolc bitje
    IFS0bits.AD1IF = 0; // AD1 megszakításbitjének törlése
}
```

És most nézzük meg egyben a programunk teljes forráskódját!

17.5. mintaprogram

```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTEN_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

/* Globális változó, futófény állapotát tárolja */
volatile unsigned char chLedek = 1;

void InitAD1 ( void ); // A/D konvertert inicializáló függvény
void InitInterrupt ( void ); // Megszakítások konfigurálása

/* Főprogram */
int main ( void )
{
    InitAD1(); // A/D konvertert inicializálása
    InitInterrupt(); // Megszakítások konfigurálása
    TRISA = 0xFF00; // PORTA alsó nyolc lába kimenet lesz.
    LATA = 0x0000; // PORTA törlése

    while(1) // Végtelen ciklus
    {
        LATA=chLedek; // chLedek változó értékét kitesszük a kimenetre
        Idle(); // A mikrokontroller Idle üzemmódba kerül
    }

    /* Analóg Digitális Konvertort inicializáló függvény */
    void InitAD1 ( void )
    {
        AD1PCFG = 0xFFDF; // AN5 analóg bemenet engedélyezése,
                           // a többi bemenet digitális I/O lesz
    }
}
```

```

AD1CON1 = 0x00E4; // Automatikus konverzió indítása
// mintavételezés után, koverzió után
// új mintavételezés indítása
AD1CON2 = 0; // MUXA használata, a Vref+/- feszültségként
// a AVss és a AVdd használata.
AD1CON3 = 0x1F00; // Tsamp = 31 x Tad; Tad=125ns (Tad = 1 Tcy)
AD1CHS = 5; // AN5 bemenet kiválasztása
AD1CONbits.ADON = 1; // A/D konverter bekapcsolása
}

/* Megszakítások konfigurálását végző függvény */
void InitInterrupt ( void )
{
    SRbits.IPL = 0; // Processzor prioritásszintjének beállítása
    IPC3bits.AD1IP = 3; // AD1 megszakítás prioritásszintjének
    // beállítása
    IEC0bits.AD1IE = 1; // AD1 megszakításának engedélyezése
}

/* A/D konverter megszakításrutinja */
void _ISR _ADC1Interrupt ( void )
{
    chLedek = ADC1BUF0 >> 2; // ADC felső nyolc bitje
    IFS0bits.AD1IF = 0; // AD1 megszakításbitjének törlése
}

```

Próbáljuk ki a programot! Ha helyesen állítottuk be az A/D konvertort, akkor a LED-sor állapota a potenciométer forgatásával együtt változik.

Elegánsabb programot kapunk, ha a megszakításrutinban a chLedek változó értékét a következőképpen számoljuk ki:

17.6. mintaprogram

```

/* A/D konverter megszakításrutinja */
void _ISR _ADC1Interrupt ( void )
{
    chLedek = 1 << (ADC1BUF0 >> 7);
    IFS0bits.AD1IF = 0; // AD1 megszakításbitjének törlése
}

```

Jó szórakozást a program teszteléséhez!

17.7. TÖBB MEGSZAKÍTÁSFORRÁS EGYIDEJŰ HASZNÁLATA

A programok nagy többségében nem csak egy megszakításforrás eredményezhet megszakítást. Ilyenkor minden egyes megszakításforrásnak el kell készíteni a megszakításrutinját, és biztosítani kell az esetleges megszakításrutinok közötti helyes kommunikációt is. Több megszakításforrás engedélyezése esetén már fontos szerepet kapnak az egyes megszakításforrások prioritásszintjei. A programírás során minden figyelnünk kell arra, hogy melyik megszakításforrás képes megszakítani egy másik megszakításrutin végrehajtását.

A fejezet végén egyesítük a fejezetben elkészített két programunkat. Készitsünk egy olyan futófényprogramot, melynek sebességét a potenciométer segítségével tudjuk változtatni.

Azért, hogy meg tudjuk változtatni a futófénünk sebességét egy új, globális változót kell bevezetnünk, az uiIdo-t.

```

/*Globális változó, futófény sebessége*/
volatile unsigned int uiIdo = 1;

```

A változó értékét az A/D konverter megszakításrutinja fogja beállítani, az utoljára mért analóg érték alapján. Mivel az A/D konverter 10 bites, ezért az eredményregiszter értéke 0 és 1023 közötti szám lehet. Azért, hogy az időzítésünk ne legyen nulla, az uiIdo változó értékét az ADC1BUF0 regiszter eggyel növelt értékével tegyük egyenlővé.

```

/* A/D konverter megszakításrutinja */
void _ISR _ADC1Interrupt ( void )
{
    uiIdo = ADC1BUF0+1; // Időzítés beállítása
    IFS0bits.AD1IF = 0; // AD1 megszakításbitjének törlése
}

```

Az időzítés értékét a Timer1 időzítő megszakításrutinja fogja beállítani. A megszakításrutinban minden egyes időzítőmegszakítás után újból be kell állítani a PR1 regiszter értékét az A/D konverter által kiszámolt változóérték alapján.

```

/* Timer1 megszakításrutinja */
void _ISR _T1Interrupt ( void )
{
    __asm__ ("rlnc.b _chLedek"); // chLedek változó eggyel balra rotálása
    PR1 = uiIdo; // Időzítés beállítása
    IFS0bits.T1IF = 0; // Timer1 megszakításbitjének törlése
}

```

A Timer1 időzítő modul beállításánál két apró módosítást érdemes elvégezni. A futófény lassabb sebessége érdekében, érdemes a Timer1 időzítő előosztóját 1:64 helyett 1:256-ra állítani. A második módosítás PR1 első értékére vonatkozik, ami az uiIdo változó kezdőértékét fogja felvenni.

```

/* Timer1 modult inicializáló függvény */
void InitT1 ( void )
{
    T1CONbits.TCS = 0; // Belső oszcillátor használata (Fosc/2)
    T1CONbits.TCKPS = 0b11; // 1:256 frekvenciaosztó használata
    T1CONbits.TGATE = 0; // Kapuzó áramkör letiltása
    T1CONbits.TSIDL = 0; // IDLE állapotban is fussen az áramkör
    TMRI = 0; // Timer1 számlálójának törlése
    PR1 = uiIdo; // Első időzítés
    IFS0bits.T1IF = 0; // Timer1 megszakításbitjének törlése
    T1CONbits.TON = 1; // Timer1 bekapcsolása
}

```

A két program egyesítésekor az InitInterrupt() függvényt egyesíteni kell, az A/D konvertort inicializáló függvényt csak át kell másolni. A két mintaprogram egy programmá olvasztásánál figyeljünk arra, hogy inicializáláskor ne felejtse el meghívni a main() függvényben a Timer1 időzítőt és az A/D konvertort inicializáló függvényt.

Az előbb említettek alapján a programunk véleges forráskódja a következő lesz:

17.7. mintaprogram

```
#include <p24fj128ga010.h>
_CONFIG1( JTAGEN_OFF & FWDTE_N_OFF )
_CONFIG2( POSCMOD_HS & FNOSC_PRI )

/* Globális változó, futófény állapotát tárolja */
volatile unsigned char chLedek = 1;
/* Globális változó, futófény sebessége */
volatile unsigned int uiIdo = 1;

void InitT1 ( void ); // Timer1 modult inicializáló függvény
void InitAD1 ( void ); // A/D konvertert inicializáló függvény
void InitInterrupt ( void ); // Megszakítások konfigurálása

/* Főprogram */
int main ( void )
{
    InitT1(); // Timer1 inicializálása
    InitAD1(); // A/D konverter inicializálása
    InitInterrupt(); // Megszakítások konfigurálása
    TRISA = 0xFF00; // PORTA alsó nyolc lába kimenet lesz.
    LATA = 0x0000; // PORTA törlése

    while(1) // Végtelen ciklus
    {
        LATA=chLedek; // chLedek változó értékét kitesszük a kimenetre
        Idle(); // A mikrokontroller Idle üzemmódba kerül
    }
}

/* Timer1 modult inicializáló függvény */
void InitT1 ( void )
{
    T1CONbits.TCS = 0; // Belső oszcillátor használata (Fosc/2)
    T1CONbits.TCKPS = 0b11; // 1:256 frekvenciaosztó használata
    T1CONbits.TGATE = 0; // Kapuzó áramkör letiltása
    T1CONbits.TSIDL = 0; // IDLE állapotban is fusson az áramkör
    TMR1 = 0; // Timer1 számlálójának törlése
    PR1 = uiIdo; // Első időzítés
    IFS0bits.T1IF = 0; // Timer1 megszakításbitjének törlése
    T1CONbits.TON = 1; // Timer1 bekapcsolása
}

/* Analóg Digitális Konvertert inicializáló függvény */
void InitAD1 ( void )
{
    AD1PCFG = 0xFFDF; // AN5 analóg bemenet engedélyezése,
    // a többi bemenet digitális I/O lesz
    AD1CON1 = 0x00E4; // Automatikus konverzió indítása
    // mintavételezés után, koverzió után
    // új mintavételezés indítása
    AD1CON2 = 0; // MUXA használata, a Vref+/ - feszültségekkel
    // a AVss és a AVdd használata.
    AD1CON3 = 0x1F00; // Tsamp = 31 x Tad; Tad=125ns (Tad = 1 Tcy)
    AD1CHS = 5; // AN5 bemenet kiválasztása
    AD1CON1bits.ADON = 1; // A/D konverter bekapcsolása
}
```

```
/* Megszakítások konfigurálását végző függvény */
void InitInterrupt ( void )
{
    SRbits.IPL = 0; // Processzor prioritásszintjének beállítása
    IPC0bits.T1IP = 4; // Timer1 megszakítás prioritásszintjének
    // beállítása
    IEC0bits.T1IE = 1; // Timer1 megszakításának engedélyezése
    IPC3bits.AD1IP = 3; // AD1 megszakítás prioritásszintjének
    // beállítása
    IEC0bits.AD1IE = 1; // AD1 megszakításának engedélyezése
}

/* Timer1 megszakításrutinja */
void _ISR _T1Interrupt ( void )
{
    __asm__ ("rlnc.b _chLedek"); // chLedek változó eggyel balra rotálása
    PR1 = uiIdo; // Időzítés beállítása
    IFS0bits.T1IF = 0; // Timer1 megszakításbitjének törlése
}

/* A/D konverter megszakításrutinja */
void _ISR _ADC1Interrupt ( void )
{
    uiIdo = ADC1BUF0+1; // Időzítés beállítása
    IFS0bits.AD1IF = 0; // AD1 megszakításbitjének törlése
}
```

Frissen elkészített programunk az elvárásaink szerint működik. A fejlesztőpanelen lévő potenciométer elcsavarásával megváltozik a futófény sebessége. Abban az esetben, ha lassítani szeretnénk a futófényünket, akkor érdemes az A/D konverter megszakításrutinjában az uiIdo változó értékét nagyobb határok között változtatni, például az ADC1BUF0 regiszter értékének megszorzásával.

Megszakítások használatánál minden figyelnünk kell arra, hogy a magasabb prioritású megszakítások bármikor megszakíthatják a főprogram vagy egy alacsonyabb szintű megszakítás végrehajtását. Ez a tény addig nem okoz problémát, amíg a megszakításrutin nem olyan kritikus szakaszban szakítja meg egy másik megszakításrutin vagy a főprogram futását, amikor a megszakító által használt változót éppen a megszakított is használja, írja vagy olvassa.

Nézzük meg az utolsó programunkban a megszakításokhoz tartozó prioritásszinteket. A főprogramnak van a legkisebb prioritásszintje, mert nulla értékre van állítva. A főprogram futását minden a két megszakításforrás meg tudja szakítani. Az A/D konverter megszakításának a prioritásszintje három, a Timer1 időzítő megszakítás prioritásszintjének értéke négy. Ebből következik, hogy az időzítő meg tudja szakítani az A/D konverter megszakításutinját, de az A/D konverter megszakításutinja nem tudja megszakítani az időzítő megszakításutinjának végrehajtását.

Mi történik akkor, ha a Timer1 időzítő megszakításutinja abban a pillanatban szakítja meg az A/D konverter megszakításutinját, amikor az uiIdo változó új értékét állítja be. Abban az esetben, ha a program nem tudja egy assembly lépéssben megváltoztatni egy változó értékét, előfordulhat, hogy a megszakítás akkor történik meg, amikor a program még csak a változó egyik értékét módosította, a másik felét viszont még nem. Ilyenkor a megszakító rutin egy hibás változóértékkel fog dolgozni, ami a program helytelen működéséhez vezethet. Természetesen ez a hiba olyankor is fennállhat, ha a végrehajtás alatt

álló program a változó egyik felét beolvasta, majd a megszakításra kerül a program, ahol az olvasás alatt álló változó értéke módosul, így a megszakított rutin hibás változóértékkel fog tovább dolgozni.

Mostani programunkban nem kell számolnunk ezzel a veszéllyel, mert az uiIdo változó unsigned int típusú, azaz 16 bit hosszúságú, így a program egy assembly utasítás-sal meg tudja változtatni az értékét. Így minden helyes érték lesz a változóban. Az A/D konverter megszakításrutinjában az uiIdo változó értékének módosítását a következő assembly sorokkal oldja meg a fordító:

```
uiIdo = ADC1BUF0+1; // Időzítés beállítása
mov.w 0x0300,0x0000
inc.w 0x0000,0x0000
mov.w 0x0000,0x0802
```

Az ADC1BUF0 regiszter a 0x0300, az uiIdo változó a 0x0802 memóriacímen található. Az assembly kódóból megfigyelhető, hogy a program először beolvassa az ADC1BUF0 regiszter értékét a W0 regiszterbe, majd megnöveli a W0 regiszter értékét eggyel, és végül a W0 regiszter értékét átmásolja az uiIdo változóba. Az assembly utasításokból látszik, hogy az uiIdo változó értéke egy assembly utasítás hatására változik meg.

Mi történne akkor, ha az uiIdo változó nem 16 bites, hanem például egy 32 bites long típusú változó lenne? Ebben az esetben már nem egy assembly utasítással történne meg a változó beállítása, ahogy azt a következő assembly sorok is mutatják:

```
uiIdo = ADC1BUF0+1; // Az uiIdo változó unsigned long típusú!
mov.w 0x0300,0x0000
inc.w 0x0000,0x0000
mov.w #0x0,0x0002
mov.w 0x0000,0x0802
mov.w 0x0002,0x0804
```

Az assembly utasításlistából látható, hogy a változó új értékét az utolsó két utasítás állítja be, először az alsó 16 bit módosul, majd a felső 16 bit. Ilyenkor, ha a két utasítás végrehajtása közé ékelődik be a Timer1 időzítő megszakításutinja, a rutin hibás változó-értékkel fog dolgozni.

Ennek a hibának a kiküszöbölésére szolgál a disi assembly utasítás. Az utasítás a paramétereként megadott utasításciklusig kikapcsolja a hetes prioritásnál kisebb megszakításokat, így a programok nem tudják egymást megszakítani a kritikus szakaszokban.

Abban az esetben, ha unsigned long típus uiIdo változót szeretnénk használni, akkor az A/D konverter megszakításutinját a következőképpen kell módosítani:

17.8. mintaprogram

```
/* A/D konverter megszakításutinja */
void _ISR _ADC1Interrupt ( void )
{
    __asm__ ("disi #5"); // 7 prioritásnál kisebb megszakítások
                          // 5 assembly utasításig kikapcsolva
    uiIdo = ADC1BUF0+1; // Az uiIdo változó unsigned long típusú!
    IFS0bits.AD1IF = 0; // AD1 megszakításbitjének törlése
}
```

Azért kell a disi utasítás paramétereként minimum az #5 értéket megadni, mert a következő kifejezést a fordító öt assembly utasítás segítségével oldja meg, ahogy azt a következő disassembly sorok is mutatják:

17. fejezet: Amikor több szálön futnak az események...

```
_asm__ ("disi #5"); // 7 prioritásnál kisebb megszakítások
                      // 5 assembly utasításig kikapcsolva
disi #5
uiIdo = ADC1BUF0+1; // Az uiIdo változó unsigned long típusú!
mov.w 0x0300,0x0000
inc.w 0x0000,0x0000
mov.w #0x0,0x0002
mov.w 0x0000,0x0802
mov.w 0x0002,0x0804
```

Most már nyugodtan hátradőlhetünk, mert a programunk nem produkál majd helytelen változóértékek következtében hibás működést.

18. ÁTJÁRÁS A VILÁGOK KÖZÖTT

A fejezet három, különböző architektúrához készült fordító használatát mutatja be. minden egyes fordító ismertetése a fordító könyvtárszerkezetének bemutatásával kezdődik, majd az első projekt létrehozásának lépéseivel folytatódik. minden egyes architektúrán ugyanazt a négy mintaprogramot fogjuk elkészíteni:

- Az első program bekapcsolja az adott mikrokontrollerhez tartozó gyakorlópanelen lévő LED-eket.
- A második mintapélda szoftveres késleltetés segítségével futófényt valósít meg a LED-soron.
- A harmadik mintapélda az időzítő megszakítás segítségével vezérli a futófényt.
- A negyedik mintapéldában a futófény időzítésének időalapját a gyakorlópanelen lévő potenciometré segítségével állíthatjuk be.

18.1. ÁTTÉRÉS A MICROCHIP C18 FORDÍTÓJÁRA

8/16 A Microchip MPLAB C18 fordítója a PIC 18 mikrokontroller-családhoz készített ANSI C kompatibilis fordító. A PIC 18 mikrokontrollerek a 8 bites adathosszúságú mikrokontrollerek családjába tartoznak, ezért a regisztereik 8 bit szélességűek.

Az MPLAB C18 mikrokontrollerekhez való C fordító tanuló-, ingyenes változata megtalálható a könyv DVD-mellékletében, vagy a legfrissebb változata elérhető a <http://www.microchip.com/c18> webcímen.

18.1.1. A fordító könyvtárszerkeze

A fordító alapértelmezett esetben a c:\MCC18\ könyvtárba települ fel, föbb alkonyvtárai a következők:

- A C:\MCC18\bin\ könyvtárban a fordító futtatható állományai találhatók.
- A C:\MCC18\doc\ könyvtárban a fordítóhoz tartozó dokumentáció található.
- A C:\MCC18\hl\ könyvtárban a standard C és a mikrokontroller-specifikus fejlécállományok találhatók.
- A C:\MCC18\lib\ könyvtárban a standard C és a mikrokontroller-specifikus előfordított könyvtárak találhatók, lib kiterjesztéssel.
- A C:\MCC18\lkr\ könyvtárban a mikrokontroller-specifikus linker állományok találhatók, lkr kiterjesztéssel.

18.1.2. Az első projekt elkészítése

A következő pár lépés segítségével készítsük el az első 8 bites C programunk környezetét. A kontroller kiválasztásánál a PIC18F46K20 típust fogjuk választani. Ennek oka az, hogy a jövendőbeli programjaink a PIC18FXK20 Starter Kit gyakorlópanelre készülnek majd, amely az előbb említett mikrokontrollert tartalmazza. A projekt létrehozásának megnete a következő:

- 1) Indítsuk el az MPLAB IDE-t, majd hozzunk létre egy új projektet a varázsló segítségével, amelyet a Project → Project Wizard... menüpont alatt találunk.
- 2) Az üdvözlőszöveget tartalmazó oldal után az eszköz (mikrokontroller) típusának kiválasztása következik. Itt a legördülő menüben a PIC18F46K20 típus válasszuk ki.
- 3) A következő oldalon a fordítóeszközt kell kiválasztani, itt a Microchip C18 Toolsuite-t válasszuk ki.
- 4) Következő lépés a projektállomány létrehozása. Hozzunk létre egy ElsoC18 nevű könyvtárat és abban egy projektet elso névvel.

5) Ezek után a projekthez hozzá kell adni a 18F46K20i.lkr szerkesztő (linker) állományt. Abban az esetben, ha a C18-as fordító az alapértelmezett helyre lett feltelepítve, akkor az előbb említett állomány a C:\MCC18\lkr\ könyvtárban található.

Természetesen itt is igaz az, hogy az új MPLAB változatoknál már elhagyható az utolsó lépés, mivel a fejlesztői környezet automatikusan hozzárendeli a projekthez a szükséges szerkesztőállományt.

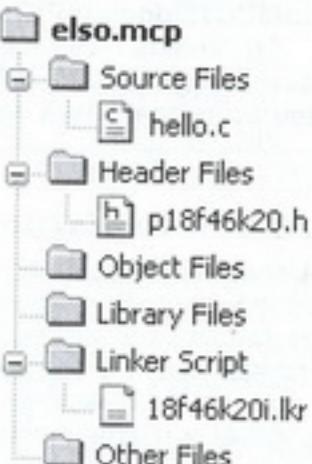
Minden egyes PIC 18-as mikrokontrollerhez négy különböző szerkesztőállomány tartozik. Lehet olyan, amely csak a mikrokontroller megnevezését tartalmazza, ilyen például a 18F46K20.i.lkr állomány. A kiegészítés nélküli állományok végeleges, már nem tesztelésre fordított állományok készítésére valók. Az „l” végződéssel ellátott szerkesztőállományok a programok valós idejű tesztelésére szolgálnak, memóriát biztosítanak az ICD2 vagy a PicKit2 debugger számára.

Az „_e” végződéssel ellátott szerkesztőállományt abban az esetben kell használnunk, ha a fordítót kiterjesztett (extended) assembly utasításokat is használó kód generálására állítottuk be. Abban az esetben, ha a szerkesztőállomány „l_e” végződéssel van ellátva, a linker biztosítja a debugger modul számára a memóriaterületet, és egyúttal az extended kód használatát is támogatja.

Ezzel az öt lépéssel létre is hoztunk egy új projektet. Következő lépésként adjunk hozzá a projektünkhez egy üres hello.c nevű állományt. Érdemes még a projektünkhez a mikrokontroller típusához tartozó fejlécállományt is hozzáadni, amelynek neve: p18f46k20.h Az állomány megtalálható a C:\MCC18\hl könyvtárban.

Ha minden helyesen végeztünk el, akkor a projektünk a 18.1. ábrán látható állományokat tartalmazza.

A PIC18FXK20 Starter Kit gyakorlópanelén a D port összes lábán LED-ek találhatók. Az első programunk feladata legyen az, hogy a gyakorlópanelen lévő LED-eket bekapcsolja.



18.1. ábra
C18 projekt

18.1. mintaprogram

```

#include <p18f46k20.h>

#pragma config FOSC = INTIO67 // Belső oszcillátor használata
#pragma config WDTE = OFF // Watch Dog Timer kikapcsolása
#pragma config LVP = OFF // Low Voltage Programming kikapcsolása

main ( )
{
    TRISD = 0x00; // Főprogram kezdete
    while(1)
    {
        LATD = 0xFF; // PORTD kimenet lesz.
    }
} // PORTD összes bitje egy lesz.

```

Ha olyan kódot szeretnénk készíteni, amelyet több típusú mikrokontrollerre is le szeretnénk fordítani, az `#include` utasítás paramétereként az adott mikrokontrollerhez tartozó fejlcállomány helyett a `p18cxx.h` állományt érdemes megadnunk. A `p18cxx.h` fejlcállomány előfordítói makrók segítségével betöltik a projektben általunk beállított, mikrokontrollerhez tartozó fejlcállományt.

18.1.3. Konfigurációs bitek beállítása

A frissen elkészített programunk kódjának olvasásánál látható, hogy csak egy helyen tér el az eddig megszokott programkódtól, mégpedig a konfigurációs bitek beállításánál.

A C18 fordító a konfigurációs biteket a `#pragma config` előfordítói utasítás segítségével állítja be. Az utasítás paramétereként meg kell adni, hogy az adott konfigurációs tulajdonság milyen értéket vegyen fel. Például a `FOSC = INTIO67` kifejezés azt jelenti, hogy a mikrokontroller belső oszcillátort használjon, és az oszcillátorlábak I/O lábként funkcionáljanak.

Az egyes mikrokontroller-típusokhoz tartozó konfigurációs beállítások megtalálhatók a `C:\MCC18\doc\hlp\pic18ConfigSet.chm` állományban.

Egy konfigurációs utasításban nemcsak egy érték állítható be, hanem visszövel elválasztva egyszerre több beállítás is elvégezhető. Például az előző példában elvégzett konfigurációs beállítások egy sorban is leírhatók:

```
#pragma config FOSC = INTIO67, WDTEN = OFF, LVP = OFF
```

18.1.4. Regiszterek és bitek használata

A speciális funkcióregiszterek, így például a portok használata is megegyezik a C30 fordítónál használt módszerrel. Természetesen azzal a különbséggel, hogy a PIC 18 családba tartozó mikrokontrollerek regisztereinek szélessége nyolc bit, ellentétben a 16 bites kontrollerek 16 bit szóhosszúságú regisztereivel.

Az egyes funkcióregiszterek bitjei a C30 fordítónál már megismert bitstruktúrák segítségével a mikrokontroller fejlcállományában vannak definiálva. Például a D port ötödik bitjét a következő kifejezéssel állíthatjuk egybe:

```
LATDbits.LATD5 = 1;
```

18.1.5. Változótípusok

A 18.2. ábrán látható táblázat a Microchip C18 fordító által használt, lebegőpontos számok tárolására alkalmas változótípusokat foglalja össze. A táblázatból kiderül, hogy a fordító nem tesz különbséget a `float` és `double` változótípusok között, mind a két típus 32 biten ábrázolja a lebegőpontos számokat.

| Típus | float | double |
|----------------------------------|---|---|
| Méret | 32 bit | 32 bit |
| Legkisebb kitevő | -126 | -126 |
| Legnagyobb kitevő | 128 | 128 |
| Minimum (normalizált alakban) | 2^{-126} $\approx 1.17549435e - 38$ | 2^{-126} $\approx 1.17549435e - 38$ |
| Maximum (normalizált alakban) | $2^{128} \cdot (2 - 2^{-15})$ $\approx 6.80564693e + 38$ | $2^{128} \cdot (2 - 2^{-15})$ $\approx 6.80564693e + 38$ |

18.2. ábra

A Microchip C18 fordító lebegőpontos típusú változói

18. fejezet: Átjárás a világok között

A 18.3. ábrán látható táblázat a Microchip C18 fordító által használt, egész számok tárolására alkalmas változótípusokat foglalja össze.

| Típus | Méret | Minimum | Maximum |
|---------------------|--------|----------------|---------------|
| char | 8 bit | -128 | 127 |
| signed char | 8 bit | -128 | 127 |
| unsigned char | 8 bit | 0 | 255 |
| int | 16 bit | -32,768 | 32,767 |
| unsigned int | 16 bit | 0 | 65,535 |
| short | 16 bit | -32,768 | 32,767 |
| unsigned short | 16 bit | 0 | 65,535 |
| short long | 24 bit | -8,388,608 | 8,388,607 |
| unsigned short long | 24 bit | 0 | 16,777,215 |
| long | 32 bit | -2,147,483,648 | 2,147,483,647 |
| unsigned long | 32 bit | 0 | 4,294,967,295 |

18.3. ábra

A Microchip C18 fordító egész típusú változói

Az ANSI C specifikációhoz képest a változó deklarációknál használható előtagok az `overlay` előtaggal bővülnek. Különböző függvényekben az `overlay` előtaggal deklarált változókat a fordító megpróbálja ugyanarra a memóriaterületre lehelyezni, így a legkisebb memóriahasználatot elérni. Abban az esetben, ha a függvények meghívják egymást, akkor a fordító nem fogja végrehajtani az optimalizációt.

A következő példában két függvénydefiníció található. Mivel a két függvény nem hívja meg egymást, ezért a két lokális változó nagy valószínűséggel ugyanarra a memóriaterületre fog kerülni.

```
int f (void)
{
    overlay int x = 1;
    return x;
}
int g (void)
{
    overlay int y = 2;
    return y;
}
```

18.1.6. Mutatók

A Microchip C18 fordító, ellentétben a C30 fordítóval, megkülönbözteti a `near` és `far` mutatóosztályokat. Adatmemória címzése esetén a `far` típusú mutatók az egész adatmemoriát meg tudják címezni, míg a `near` típusú mutatók csak az `access` memóriát tudják elérni. Ebből következik, hogy az adatmemória-mutató nagysága `near` típus esetén 8 bit, míg `far` típus esetén 16 bit.

Programmemória címzése esetén a `far` típusú mutatók az egész programmemóriát meg tudják címezni, míg a `near` típusú mutatók csak a programmemória első 64 kibocsátáját tudják megcímezni. Ebből következik, hogy a programmemória-mutató nagysága `near` típus esetén 16 bit, míg `far` típus esetén 24 bit.

Azt, hogy egy mutató az adatmemóriára mutasson, a `ram` előtag használatával tudjuk elérni. Abban az esetben, ha nem írjuk ki, hogy milyen memóriaterületre mutasson a mutató, akkor automatikusan az adatmemóriára fog mutatni. Ha azt szeretnénk, hogy egy mutató a programmemóriára mutasson, akkor `rom` előtagot kell használni.

Abban az esetben, ha egy mutató elé nem írjuk ki az osztályazonosítóját, akkor alapértelmezett esetben az adatmemóriára fog mutatni, és `far` típusú lesz.

A következő táblázat összefoglalóan bemutatja az egyes mutatóosztályok hatását.

| | <code>rom</code> | <code>ram</code> |
|-------------------|------------------------------------|---------------------------------|
| <code>far</code> | Teljes programmemória megcímezhető | Teljes adatmemória megcímezhető |
| <code>near</code> | Csak 64 kibisző címezhető meg | Csak az access ram érhető el |

18.4. ábra
Mutatók címzési tartományai

Végül nézzünk pár mintát a mutatók deklarálására:

```
/* Adatmemóriára mutató */
char *dmp; // Nagysága: 16 bit

/* Programmemóriára mutató (near típus) */
rom near char *npmp; // Nagysága: 16 bit

/* Programmemóriára mutató (far típus) */
rom far char *fpmp; // Nagysága:24 bit
```

18.1.7. Tárolási hely meghatározása

Az előző fejezetben bevezetett `ram` és `rom` kulcsszót nemcsak mutatók előtt lehet használni, hanem változók vagy tömbök deklarálása esetén is. A `ram` előtag hatására az adott változó az adatmemóriába kerül. A változók deklarálásakor az alapértelmezett memóriaterület az adatmemória, így abban az esetben, ha nem írunk előtagot, a változó az adatmemóriába kerül.

A `rom` előtag használatával a változó vagy a tömb a programmemóriába kerül, így nem lehet módosítani. A programmemóriában elhelyezett változókat a program táblaolvasó assembly utasításokkal éri el.

A következő két deklaráció az adat- és a programmemóriában hoz létre változót. Abban az esetben, ha programmemóriában hozunk létre változót, akkor ki kell írnunk a `const` és a `static` előtagot is. A `const` előtag szimbolizálja azt, hogy a váltoozó csak olvasható, a `static` előtag pedig arra utal, hogy a változó fix helyen található a memóriában.

```
ram int x = 1;
static const rom str[] = "Hello World";
```

18.1.8. Inline Assembly

Az Microchip C18 fordító is lehetőséget nyújt arra, hogy assembly utasításokat ágyazunk be a C kódunkba. A beágyazott assembly részt `_asm` és az `_endasm` kifejezések között lehet elhelyezni.

Pár fontos tudnivaló a beágyazott assemblyvel kapcsolatban:

- A beágyazott assemblyben lehetőségünk van címkék használatára is. A címkéket ket-tösponttal kell lezárnai.
- Az assembly részben is a C fordító által használt megjegyzésekkel kell használni, nem lehet pontosvesszővel kezdődő megjegyzést használni.

- A teljes utasításneveket kell használni, nincs lehetőség rövidítésre.
- Az utasításoknál az összes paramétert meg kell adni, nincs alapértelmezett paraméter.
- Konstansok esetén a tízes számrendszer az alapértelmezett.
- Konstansok előtt a C nyelv által használt számrendszer előtagot kell használni. Pl. a `H'1234'` helyett `0x1234` kell használni.

A következő mintapélda a beágyazott assembly utasítások használatára mutat példát:

```
_asm
/* Beágyazott assembly kód */
MOVLW 10 // A count változó értéke legyen egyenlő 10 decimális
értékkel.
MOVWF count, 0
/* Ciklus, amíg a count változó értéke nem lesz nulla */
start:
    DECFSZ count, 1, 0
    GOTO done
    BRA start
done:
_endasm
```

A C18 fordító mikrokontroller-specifikus fejlécállományában definiálásra kerültek bizonyos alap assembly utasítások inline assembly makrói. Ezek a makrók a következők:

- | | |
|------------------------|---|
| • Nop() | NOP utasítás beillesztése |
| • ClrWdt() | Watchdog Timer törlése (CLRWDUT utasítás) |
| • Sleep() | SLEEP utasítás beillesztése |
| • Reset() | Szoftveres reset (RESET utasítás) |
| • Rlc(f,dest,access) | File regiszter carryn keresztsüli balra rotálása |
| • Rln(f,dest,access) | File regiszter carry nélküli balra rotálása |
| • Rrc(f,dest,access) | File regiszter carryn keresztsüli jobbra rotálása |
| • Rrn(f,dest,access) | File regiszter carry nélküli jobbra rotálása |
| • Swapf(f,dest,access) | Két regiszter értékének kicserélése |
- Nézzünk egy példát az előbbi makrók használatára. Készítsük el a már jól ismert futófényprogramunkat az `Rln` makró használatával. A futófény lassítására szoftveres késleltetést használunk.

18.2. mintaprogram

```
#include <pl18f46k20.h>
/* Konfigurációs bitek beállítása */
#pragma config FOSC = INTIO67, WDTEN = OFF, LVP = OFF

unsigned char chLedek = 1; // A futófény állapota

void main ( )
{
    unsigned int uiIdo; // Késleltetéshez használt változó

    TRISD = 0x00; // PORTD kimenet lesz
    LATD = 0x00; // PORTD törlése

    while(1)
    {
        Rln(chLedek, 1, 1); // chLedek globális változó eggyel balra
        LATD = chLedek; // rotálása
        // ledék változó értékét
        // kitesszük a kimenetre
```

```

for(uiIdo=0; uiIdo<1000; uiIdo++)
{
    Nop();           // Szoftveres késleltetés:
    // 1000-szer fut le a nop utasítás.
}

```

Az Rlncf(f,dest,access) makró a következő assembly utasításokat helyezi be a programkódba: _asm movlb f rlncf f,dest,access _endasm A C18 fordító a lokális változókat a szoftveres stackbe helyezi, ezért azt csak indirekt címzéssel lehet elérni, így a mikrokontroller fejlécállományában definiált makrók által behelyettesített utasítások nem tudnak a lokális változón műveletet elvégezni. Azért, hogy elkerüljük a fordító által alkalmazott szoftveres stack használatát, az Rlncf makró által behelyettesített assembly utasítás használatakor a chLedek változót érdemes globálisra állítani. Olyan esetekben, ha lokális változóval szeretnénk inline assembly műveletet végezni, akkor először a fordító által használt stack mutatót rá kellett állítani az adott változóra. Példaképp, ha a chLedek változó lokális lett volna, azt a következő két utasítással tudtuk volna eggyel balra rotálni:

```

_asm movlw chLedek _endasm
Rlncf(PLUSW2, 1, 0);

```

18.1.9. Megszakítások használata

A PIC18 mikrokontroller-család két megszakításvektorral rendelkezik. Abban az esetben, ha magas prioritással rendelkező megszakításforrásból érkezik megszakítás, akkor a vezérlés a 0x0008 programmemória-címre, míg alacsony szintű megszakítás esetén a vezérlés a 0x0018 címre kerül. Az alacsony és a magas szintű megszakításvektor között csak 16 utasításnyi hely van, ezért ennél nagyobb megszakításrutin nem fér el a két vektor közé. Általában olyan kevés utasításból álló megszakításrutint, amely elfér a két vektor között, ritkán szoktunk készíteni, ezért a megszakításrutinunkat érdemes két részre bontani.

A megszakításrutinunk első részét a megszakításvektor címére kell fordítanunk. Ezt a fordítónak a #pragma code utasítás segítségével tudjuk megadni. A megszakításvektora fordított függvény csak egy inline assembly utasítást tartalmaz, amelynek segítségével a program végrehajtását a valóságos, a programmemória más területén elhelyezkedő megszakításkezelő rutinra helyezzük át.

Hogy egy függvény a megszakításkezelő rutin szerepét látja el, azt jelezünk kell a fordító felé. A fordítónak ezt a #pragma interrupt utasítás segítségével tudjuk megadni. Ilyen esetben a fordító automatikusan menti a rutin által használt regisztereit a megszakításrutin elején. Regiszterek mentésekor a fordító kihasználja, hogy a megszakításrutin meghívásakor a W, Status és BSR regiszterek hardverszinten mentésre kerülnek. A megszakításrutin végén a fordító visszaállítja a rutin elején elmentett regiszterértékeket, majd a RETFIE utasítással fejezi azt be.

Az ugyanolyan prioritással rendelkező megszakításforrások ugyanarra a megszakításvektorra adják át a vezérlést. Ezért a megszakításrutint érdemes egy if szerkezetre felépíteni. Az elágazás feltételében megadjuk az adott megszakításhoz tartozó megszakításbitet, a feltétel igaz ágában elvégezzük az adott megszakításhoz tartozó feladatot, majd az ág végén töröljük a megszakításhoz tartozó megszakításbitet.

Egyszintű megszakítások használata esetén érdemes a következő programkódot használni:

```

/* Egy kicsi megszakításkezelő függvény 0x0008 programcímre kerül */
#pragma code InterruptVectorHigh = 0x08
void InterruptVectorHigh (void)
{
    /* A függvény a valódi megszakításkezelő rutinra ugrik */
    _asm
        goto InterruptHandlerHigh
    _endasm
}

/* A fordító újból saját maga helyezi el a függvényeket a
programmemoriába*/
#pragma code

/* Magas prioritású megszakítások kezelő függvénye */
#pragma interrupt InterruptHandlerHigh
void InterruptHandlerHigh ()
{
    if(INTCONbits.TMR0IF)      /* Timer0 megszakítás */
    {
        /* - */
        INTCONbits.TMR0IF = 0; // Timer0 megszakításbitjének törlése
    }
}

```

A következő mintapélda az előző mintapélda továbbfejlesztett változata. A futófény új állapotát a Timer0 időzítőhöz tartozó megszakításrutin fogja kiszámítani. Az időzítöt úgy állítjuk be, hogy a megszakítás 250 ms-onként érkezzen meg. Ehhez a belső oszcillátor frekvenciáját 8 MHz-re állítjuk be. A főprogram a változó állapotát kihelyezi a LED-sorra, majd a processzort Idle, energiatakarékos állapotba helyezi.

18.3. mintaprogram

```

#include <p18f46k20.h>
/* Konfigurációs bitek beállítása */
#pragma config FOSC = INTIO67, WDTE = OFF, LVP = OFF
/* Globális változó, futófény állapotát tárolja*/
volatile unsigned char chLedek = 1;

/* Megrakításkezelő függvény deklarációja */
void InterruptHandlerHigh (void);

void InitT0 ( void );           // Timer0 modult inicializáló függvény
void InitInterrupt ( void );    // Megrakítások konfigurálása

void main ( )
{
    OSCCON = 0b01100000;          // Fosc = 8 MHz beállítása
    InitT0();                     // Timer0 inicializálása
    InitInterrupt ();             // Megrakítások konfigurálása
    TRISD = 0x00;                 // PORTD kimenet lesz
    LATD = 0x00;                  // PORTD törlése

    while(1)                      // Végtelen ciklus
    {
        LATD = chLedek;           // Ledek változó értékét
    }
}

```

```

        // kitesszük a kimenetre
OSCCONbits.IDLEN = 1;      // Idle üzemmód előkészítése
Sleep();                    // Energiatakarékos üzemmód bekacsolása
}

/* Timer0 modult inicializáló függvény */
void InitT0 ( void )
{
    /* 16 bites üzemmód, belső oszcillátor (Fosc/4), 1:32 előosztó */
T0CON = 0b00000100;
TMR0H = 0xC2;                // Időzítési ciklus: 250ms
TMR0L = 0xF7;                // TMR0 = 65536 - 15625;
INTCONbits.TMR0IF = 0;        // Timer0 megszakításbitjének törlése
T0CONbits.TMR0ON = 1;        // Timer0 bekapcsolása
}

/* Megszakítások konfigurálását végző függvény */
void InitInterrupt ( void )
{
    INTCON2bits.TMR0IP = 1;    // Timer0 megszakítás legyen magas prioritású
INTCONbits.TMR0IE = 1;        // Timer0 megszakításának engedélyezése
INTCONbits.GIEH = 1;         // Magas prioritású megszakítások engedélyezése
}

/* Egy kicsi megszakításkezelő függvény 0x0008 programcímre kerül */
#pragma code InterruptVectorHigh = 0x08
void InterruptVectorHigh (void)
{
    /* A függvény a valódi megszakításkezelő rutinra ugrik */
    _asm
        goto InterruptHandlerHigh
    _endasm
}

/* A fordító újból saját maga helyezi el a függvényeket a
programmemoriába*/
#pragma code

/* Magas prioritású megszakítások kezelő függvénye */
#pragma interrupt InterruptHandlerHigh
void InterruptHandlerHigh ()
{
    if(INTCONbits.TMR0IF)      /* Timer0 megszakítás */
    {
        TMR0H = 0xC2;          // Időzítési ciklus újbóli beállítása
        TMR0L = 0xF7;          // TMR0 = 65536 - 15625;
        Rlncf(chLedek, 1, 1); // chLedek globális változó eggyel balra
                               // rotálása
        INTCONbits.TMR0IF = 0;  // Timer0 megszakításbitjének törlése
    }
}

```

18.1.10. Kétszintű megszakításrendszer használata

Ebben a részben két, különböző prioritású megszakításrutint fogunk elkészíteni. Az alacsony prioritású megszakítások kezelőfüggvénye nagymértékben hasonlít a magas szintű megszakítások kezelőfüggvényéhez. Egy különbség van a két megszakításrutin között: a #pragma interruptlow utasítás, amellyel jelezük a fordítónak, hogy alacsony prioritású megszakításrutint készítünk.

A #pragma interruptlow utasítással azt jelezük a fordítónak, hogy a megszakításrutin regisztermentési és -visszaállítási fázisában ne használja az árnyékregisztert, hanem minden regiszter mentését szoftverből oldja meg. A magas prioritású megszakítások bármikor megszakíthatják az alacsony szintű megszakításrutin végrehajtását, így az árnyékregiszter értéke bármikor felülíródhhat a magas szintű megszakításrutin meghívásakor a hardver által elvégzett árnyékmentéssel, ezért nem használhatjuk az árnyékregisztereket.

Az alacsony szintű megszakításrutint is két részre bontjuk, ahogy azt a magas szintű megszakításrutinnál tettük. A megszakításrutin első része a 0x0018 programcímre kerül, amely átugrik a valódi megszakításkezelő függvényre. Az alacsony szintű megszakításrutinok általános alakja a következő:

```

/* Alacsony prioritású megszakítások belépő függvénye */
#pragma code InterruptVectorLow = 0x18
void InterruptVectorLow (void)
{
    /* A függvény a valódi megszakításkezelő rutinra ugrik */
    _asm
        goto InterruptHandlerLow
    _endasm
}

/* A fordító újból saját maga helyezi el a függvényeket a
programmemoriába*/
#pragma code

/* Alacsony prioritású megszakítások kezelő függvénye */
#pragma interruptlow InterruptHandlerLow
void InterruptHandlerLow ()
{
    if (PIR1bits.ADIF)      /* A/D konverter megszakítás */
    {
        /* ... */
        PIR1bits.ADIF=0;    // A/D megszakításbitjének törlése
    }
}

```

A következő mintapélda az előző mintapélda folytatása. Módosítsuk az előző programot úgy, hogy a program futása közben a Timer0 időzítő időalapját az AN5 lábra kötött potenciometér segítségével meg tudjuk változtatni. Az analóg–digitális átalakító megszakítását alacsony prioritásúra fogjuk állítani azért, hogy az időzítésünk pontos legyen.

Az aktuális időalapot az ui1do változó tárolja. Az ui1do változó értékét a Timer0-hoz tartozó megszakításrutin olvassa, és az analóg–digitális átalakítóhoz tartozó megszakításrutin írja. Az ui1do változó unsigned int típusú, azaz 2 bájt nagyságú, így a kontroller nem tudja egyszerre megváltoztatni az értékét. Azért, hogy a Timer0 időzítő megszakításrutinja ne olvasson be helytelen értéket, az ui1do változó írása idejére a magas szintű megszakításokat ki kell kapcsolni.

Az előző programban elkészített Timer0 időzítő konfigurációjához képest módosítsuk az előosztó nagyságát 1:256-ra, hogy nagyobb időzítési határok között mozogjon a pro-

ramunk. A megszakítások konfigurálásánál pedig ne felejtsük el a kétprioritásos megszakításrendszer engedélyezését, az RCON regiszter IPEN bitjének egyesbe állításával.

18.4. mintaprogram

```
#include <pl18f46k20.h>
/* Konfigurációs bitek beállítása */
#pragma config FOSC = INTIO67, WDTEN = OFF, LVP = OFF

/* Globális változó, futófény állapotát tárolja */
volatile unsigned char chLedek = 1;
/* Globális változó, futófény sebessége */
volatile unsigned int uiIdo = 0xF85F; // Első időzítés: ~250ms

/* Megszakításkezelő függvények deklarációja */
void InterruptHandlerHigh (void);
void InterruptHandlerLow (void);

void InitT0 ( void ); // Timer0 modult inicializáló függvény
void InitAD ( void ); // A/D konvertert inicializáló függvény
void InitInterrupt ( void ); // Megszakítások konfigurálása

void main ( ) // Főprogram kezdete
{
    OSCCON = 0b01100000; // Fosc = 8 MHz beállítása
    InitT0(); // Timer0 inicializálása
    InitAD(); // A/D konverter inicializálása
    InitInterrupt (); // Megszakítások konfigurálása
    TRISD = 0x00; // PORTD kimenet lesz
    LATD = 0x00; // PORTD törlése

    while(1) // Végtelen ciklus
    {
        LATD = chLedek; // Ledek változó értékét
        // kitesszük a kimenetre
        OSCCONbits.IDLEN = 1; // Idle üzemmód előkészítése
        Sleep(); // Energiatakarékos üzemmód bekapcsolása
    }
}

/* Timer0 modult inicializáló függvény */
void InitT0 ( void )
{
    /* 16 bites üzemmód, belső oszcillátor (Fosc/4), 1:256 előosztó */
    T0CON = 0b00000111;
    TMROH = (uiIdo>>8); // uiIdo felső 8 bitje
    TMROL = uiIdo & 0xFF; // uiIdo alsó 8 bitje
    INTCONbits.TMR0IF = 0; // Timer0 megszakításbitjének törlése
    T0CONbits.TMR0ON = 1; // Timer0 bekapcsolása
}

/* Analóg Digitális Konvertert inicializáló függvény */
void InitAD ( void )
{
    ANSEL = 0; // Az összes analóg bemenet kikapcsolása,
    ANSELH = 0b00100000; // kivéve az RE0/AN5 bemenetet.
    ADCON2 = 0b10000111; // 10 bites A/D, időzítés: belső RC osc.
}
```

```
ADCON1 = 0b00000000; // Vref+/- legyen a Vss és a Vdd
ADCON0 = 0b00010101; // AN5 csatorna kiválasztása
ADCON0bits.GO = 1; // Első A/D konverzió indítása
}

/* Megszakítások konfigurálását végző függvény */
void InitInterrupt ( void )
{
    INTCON2bits.TMR0IP = 1; // Timer0 megszakítás legyen magas prioritású
    INTCONbits.TMR0IE = 1; // Timer0 megszakításának engedélyezése
    IPR1bits.ADIP=0; // A/D megszakítás legyen alacsony prioritású
    PIE1bits.ADIE=1; // A/D megszakításának engedélyezése
    RCONbits.IPEN=1; // Prioritásos megszakítás engedélyezése
    INTCONbits.GIEL = 1; // Alacsony prioritású megszakítások engedélyezése
    INTCONbits.GIEH = 1; // Magas prioritású megszakítások engedélyezése
}

/* Magas prioritású megszakítások belépő függvénye */
#pragma code InterruptVectorHigh = 0x08
void InterruptVectorHigh (void)
{
    _asm
        goto InterruptHandlerHigh
    _endasm
}

/* Alacsony prioritású megszakítások belépő függvénye */
#pragma code InterruptVectorLow = 0x18
void InterruptVectorLow (void)
{
    /* A függvény a valódi megszakításkezelő rutinra ugrik */
    _asm
        goto InterruptHandlerLow
    _endasm
}

/* A fordító újból saját maga helyezi el a függvényeket a
programmemóriába */
#pragma code

/* Magas prioritású megszakítások kezelő függvénye */
#pragma interrupt InterruptHandlerHigh
void InterruptHandlerHigh ()
{
    if(INTCONbits.TMR0IF) /* Timer0 megszakítás */
    {
        TMROH = (uiIdo>>8); // uiIdo felső 8 bitje
        TMROL = uiIdo & 0xFF; // uiIdo alsó 8 bitje
        Rlncf(chLedek, 1, 1); // chLedek globális változó eggyel balra
        // rotálása
        INTCONbits.TMR0IF = 0; // Timer0 megszakításbitjének törlése
    }
}

/* Alacsony prioritású megszakítások kezelő függvénye */
#pragma interruptlow InterruptHandlerLow
```

```

void InterruptHandlerLow ()
{
    if (PIR1bits.ADIF) /* A/D konverter megszakítás */
    {
        INTCONbits.GIEH = 0; // Magas szintű megszakítások
        kikapcsolása
        /* Időzítés kiszámítása */
        uiDo = 0xFFFF - ((unsigned int)ADRESH << 8) | ADRESL;
        INTCONbits.GIEH = 1; // Magas szintű megszakítások
        engedélyezése
        PIR1bits.ADIF = 0; // A/D megszakításbitjének törlése
        ADCON0bits.GO = 1; // A/D konverzió indítása
    }
}

```

18.2. ÁTTÉRÉS A HI-TECH PIC-C FORDÍTÓJÁRA

8/12 8/14 A Hi-Tech C for the PIC10/12/16 MCU Family fordítója a PIC10/12/16 mikrokontroller-családhoz készített ANSI C kompatibilis fordító. A PIC10/12/16 mikrokontrollerek a 8 bites adathosszúságú mikrokontrollerek családjába tartoznak, ezért a regisztereik 8 bit szélességűek. Az architektúrát eredetileg assembly programozáshoz fejlesztettek ki, így kevés utasítást tartalmaz. Az architektúra nem C nyelvre optimalizált, de azért bizonyos megkötések betartása mellett kényelmesen lehet rá C nyelvű programokat készíteni. Ilyen megkötés például a belső, nyolc elemből álló hardver stack, amit nem szabad túllépni, így az egybeágyazott függvényhívások sem haladhatják meg a nyolcas mélységet.

Az Hi-Tech PIC-C Lite PIC10/12/16 kontrollerekhez való C fordító tanuló-, ingyenes változata megtalálható a könyv DVD-mellékletében, vagy a legfrissebb változata elérhető a <http://microchip.hisoft.com/webcimen> keresztül.

18.2.1. A fordító könyvtárszerkezete

A fordító alapértelmezett esetben a C:\Program Files\HI-TECH Szoftver\PICCLITE\9.60\ könyvtárba települ fel, főbb alkönyvtárai a következők:

- A bin könyvtárban a fordító futtatható állományai találhatók.
- A docs könyvtárban a fordítóhoz tartozó dokumentáció található.
- Az include könyvtárban a standard C és a mikrokontroller-specifikus fejlécállományok találhatók.
- A lib könyvtárban a standard C és a mikrokontroller-specifikus előfordított könyvtárak találhatók, lib kiterjesztéssel.

18.2.2. Az első projekt elkészítése

A következő pár lépés segítségével az első, kis lábszámú 8 bites C programunk környezetét készítjük el. A kontroller kiválasztásánál a PIC16F690 típust fogjuk kiválasztani. A választás oka az, hogy a programokat a PICkit™ 2 Starter Kit segítségével fogjuk tesztelni, ami az előbb említett mikrokontrollert tartalmazza. A projekt létrehozásának a menete a következő:

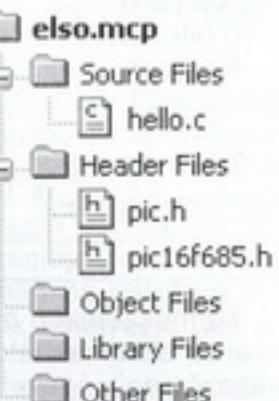
- 1) Indítsuk el az MPLAB IDE-t, majd hozunk létre egy új projektet a varázsló segítségével, amelyet a Project → Project Wizard... menüpont alatt találunk.
- 2) Az üdvözlőszöveget tartalmazó oldal után az eszköz (mikrokontroller) típusának kiválasztása következik. Itt a legördülő menüben a PIC16F690 típust válasszuk ki.
- 3) A következő oldalon a fordítóeszközt kell kiválasztani, itt a HI-TECH Universal Tool-Suite-ot válasszuk ki.

- 4) A következő lépés a projektállomány létrehozása. Hozunk létre egy ElsoC16 nevű könyvtárat és abban egy projektet elso névvel.
- 5) A projekthez nem kell linker állományt hozzáadni, mert a memóriaelosztást a fordító kezeli.

Ezzel a pár lépéssel létre is hoztunk egy új projektet. Következő lépésként adjunk hozzá a projektünkhez egy üres hello.c nevű állományt. Érdemes még a projektünkhez az általános, pic.h nevű és a mikrokontroller típusához tartozó fejlécállományt is hozzáadni, amelynek neve: p16f685.h Nem elírás, hogy a fejlécállomány neve nem egyezik meg a kiválasztott kontroller nevével, mert a Hi-Tech PIC C fordítóban nem tartozik minden egyes mikrokontroller-típushoz fejlécállomány. Egy fejlécállomány egy családra vonatkozik. Azt, hogy az adott mikrokontroller melyik család tagja, legegyszerűbben a pic.h állományban található előfordítói makrókból lehet kideríteni.

Ha minden helyesen végeztünk el, akkor a projektünk a 18.5. ábrán látható állományokat tartalmazza.

A PICkit™ 2 Starter Kithez tartozó gyakorlópanelen a C port alsó négy lábán LED-ek találhatók. Az első programunk feladata legyen az, hogy a gyakorlópanelen lévő LED-eket bekapsolja.



18.5. ábra
Hi-Tech PIC C projekt

18.5. mintaprogram

```

#include <pic.h>
/* Belső osc. használata, WDT kikapcsolása, RESET láb engedélyezése */
__CONFIG (INTIO & WDTON & MCLREN);

main ( ) // Főprogram kezdete
{
    TRISC = 0xF0; // PORTC alsó négy bitje kimenet lesz.
    while(1)
    {
        PORTC = 0x0F; // PORTC alsó négy bitje egy lesz.
    }
}

```

A Hi-Tech C fordítónál minden esetben a pic.h állományt kell meghívni, mert ez az állomány tartalmazza az általános definíciókat, és az állomány előfordítói makró segítségével automatikusan betölti a mikrokontroller típusához tartozó fejlécállományt is. A többi mikrokontroller-családhoz képest egy eltérés jelentkezik a PIC10/12/16 családhoz tartozó mikrokontrollereknél, ez pedig a LAT regiszter hiánya, így a portokra csak a PORT regiszterek segítségével lehet írni.

18.2.3. Konfigurációs bitek beállítása

A HI-TECH PIC C fordító által használt konfigurációs beállítás hasonlít a Microchip C30 fordító által alkalmazott megoldáshoz. A konfigurációs biteket a __CONFIG() makró segítségével lehet megadni. A makró paramétereiként az egyes konfigurációs beállításokat bináris és kapcsolatba állítva, kell felsorolni. Az egyes beállítások elnevezését az adott mikrokontroller-család fejlécállományában találjuk meg.

18.2.4. Regiszterek és bitek használata

A speciális funkcióregiszterek használata, így például a portoké is, megegyezik a C30 fordítónál használt módszerrel. Természetesen az a különbség megvan, hogy a PIC10/12/16 családba tartozó kontrollerek regisztereinek nagysága nyolc bit, ellentétben a 16 bites kontrollerek 16 bit szóhosszúságú regisztereivel.

Az egyes funkcióregiszterek bitjeinek elérése eltér a C30 fordítónál már megismert módszertől. A HI-TECH PIC C fordítóban a regiszterek egyes bitjei külön makrók segítségével kerültek definiálásra az adott mikrokontroller adatlaphájában megtalálható elnevezések használatával. Például a C port harmadik bitjét (RC3 bit) a következő kifejezéssel állíthatjuk egybe:

```
RC3 = 1;
```

Természetesen a fordító ismeri a bitstruktúrákat is, így a saját változóinknál ez a módszer is alkalmazható. Ezenfelül a fordító ismeri a bit típusú változót is. A bit típusú változó segítségével memóriatakarékos módon tudunk létrehozni kétállapotú változókat. A bit típusú változókkal kapcsolatban egy megkötése van a fordítónak, miszerint vagy globálisnak, vagy statikusnak kell lennie. Például egy lokális, statikus bitet a következő deklaráció segítségével tudunk létrehozni:

```
static bit vege;
```

18.2.5. Változótípusok

A 18.6. ábra a HI-TECH PIC C fordító által használt, egész számok tárolására alkalmas változótípusokat foglalja össze.

| Típus | Méret | Minimum | Maximum |
|----------------|--------|----------------|---------------|
| bit | 8 bit | 0 | 1 |
| char | 8 bit | -128 | 127 |
| signed char | 8 bit | -128 | 127 |
| unsigned char | 8 bit | 0 | 255 |
| int | 16 bit | -32,768 | 32,767 |
| unsigned int | 16 bit | 0 | 65,535 |
| short | 16 bit | -32,768 | 32,767 |
| unsigned short | 16 bit | 0 | 65,535 |
| long | 32 bit | -2,147,483,648 | 2,147,483,647 |
| unsigned long | 32 bit | 0 | 4,294,967,295 |

18.6. ábra

Hi-Tech C fordító egész típusú változói

A HI-TECH PIC C fordító két, a lebegőpontos számok ábrázolására alkalmas változótípust ismer. A float változótípus 24 bit hosszúságú. A double változótípus fordítási paramétertől függően lehet 24 vagy 32 bit hosszúságú is (alapértelmezett esetben 24 bites lesz).

| Típus | Méret |
|--------|--------|
| float | 24 bit |
| double | 32 bit |

18.7. ábra
Hi-Tech C fordító lebegőpontos változói

18.2.6. Tárolási hely meghatározása

A változók automatikusan a RAM-területen kerülnek elhelyezésre. Ha a programmemóriaiban szeretnénk változót elhelyezni, azt a const kulcsszó használatával tudjuk elérni. Például egy karakterláncot a következő deklaráció segítségével tudunk elhelyezni a programmemoriában:

```
const unsigned char szText[] = "Szoveg";
```

A PIC10/12/16 család mikrokontrollerein futó program az adatmemóriát maximum 4 darab 128 bájt nagyságú memóriabankon keresztül tudja elérni. Abban az esetben, ha a fordító számára nem jelezük, hogy melyik memóriaterületre szeretnénk változót létrehozni, akkor a nullás bankban jön létre a változó. Egy változó deklarációjakor a bank1, bank2 és bank3 előtaggal tudjuk megadni azt, hogy a fordító melyik bankba helyezze az adott változót. Például ha egy változót a hármas bankban szeretnénk elhelyezni, azt a következő kifejezéssel tehetjük meg:

```
static bank3 unsigned char valtozo;
```

A fordító közvetlen nyelvi elemmel biztosítja a belső EEPROM memória írását és olvasását. Az EEPROM memória írása sokkal lassabb és bonyolultabb, mint a hagyományos memória írása, ezért az EEPROM memóriában lévő változókkal csak alapműveleteket lehet elvégezni. Az EEPROM memóriában tárolt változót a következő kifejezéssel tudjuk létrehozni:

```
eeprom int number = 0x1234;
```

A fordító lehetőséget biztosít arra, hogy az adott változó memóriacímét meghatázzuk. Egy változó tárolási memóriacímét a változó deklarációjában adhatjuk meg, a @ jelet követően. Például ha egy változót a 0x06 memóriacímre szeretnénk helyezni, azt a következő deklarációval tehetjük meg:

```
volatile unsigned char Portvar @ 0x06;
```

18.2.7. Mutatók

A HI-TECH PIC C fordító a hardver által biztosított SFR regiszterek segítségével tudja elérni a RAM memóriát. Ebből következik, hogy az adatmemóriára mutató nagysága 8 bit, így egy mutató két bank változót tudja elérni. Abban az esetben, ha kettes vagy hármas bankban található változóra szeretnénk mutatni, akkor azt a bank3 előtaggal jelezhetjük a fordító felé.

```
unsigned char* mutatol; // Bank0-ra és Bank1-re tud mutatni
bank3 unsigned char* mutato2; // Bank2-re és Bank3-ra tud mutatni
```

Lehetőségünk van EEPROM-területen lévő változóra is mutatót készíteni úgy, hogy a mutató deklarációjában kiírjuk az eeprom kulcsszót.

```
eeprom int * nptr;
```

A ROM-területre mutató nagysága 16 bit. Mivel az architektúra nem támogatja a táblavezető utasításokat, a fordító a RETLW assembly utasítással oldja meg a ROM-területen lévő tömbök elérését.

18.2.8. Inline Assembly

Az HI-TECH PIC C fordító is lehetőséget biztosít arra, hogy assembly utasításokat ágyazunk be a C kódunkba. Beágyazott assemblyrész vagy a `#asm` és a `#endasm` kifejezések között, vagy az `asm("")` kifejezés paraméterében tudunk elhelyezni.

A két lehetőség közötti fő különbség az, hogy a `#asm` és a `#endasm` kifejezések között elhelyezett assembly utasításokat egy az egyben elhelyezi a fordító a kódban, míg az `asm("")` kifejezés paraméterében szereplő assembly kód a C kód szerves része, így a fordító figyel arra, hogy összeegyeztethető legyen az öt körülvevő C kóddal. Feltételek ágaiban, ciklusok törzsében, stb. ajánlott az `asm("")` megoldását alkalmazni.

A globális változókat aláhúzásjellel plusz a változó nevével tudjuk elérni a beágyazott assemblyrészben. A következő példa, minden inline assembly megoldással, egy `unsigned int` típusú változót binárisan tol el egy bittel balra:

```
unsigned int var;
void main(void)
{
    var = 1;
    // Első megoldás
    #asm
        bcf 3,0
        rlf _var
        rlf _var+1
    #endasm
    // Második megoldás
    asm("bcf 3,0");
    asm("rlf _var");
    asm("rlf _var+1");
}
```

A Hi-Tech C fordító mikrokontroller pic.h fejlécállományában definiálásra kerültek bizonyos alap assembly utasítások inline assembly makrói. Ezek a makrók a következők:

- `NOP()` NOP utasítás beillesztése
- `CLRWDT()` Watchdog Timer törlése (CLRWDT utasítás)
- `SLEEP()` SLEEP utasítás beillesztése

18.2.9. Megszakítások használata

Mielőtt megismerkedünk a megszakítások használatával, nézzünk egy példát egy szoftveres késleltetés elkészítéséhez. Készítsük el a már jól ismert futófényprogramunkat. A futófényprogram lesz az alapja a későbbi időzítő megszakítással ellátott programunknak. A futófényprogramunk azzal a bonyolódik, hogy a **C port** alsó négy lábán van csak LED, így négy LED alkotja a futófényt.

18.6. mintaprogram

```
#include <pic.h>
/* Belső osc. használata, WDT kikapcsolása, RESET láb engedélyezése */
__CONFIG (INTIO & WDTON & MCLREN);

main ( ) // Főprogram kezdete
{
    unsigned char chLedek = 1; // A futófény állapota
    unsigned int uiIdo; // Késleltetéshez használt változó
    TRISC = 0xF0; // PORTC alsó négy bitje kimenet lesz.
```

18. fejezet: Átjárás a világok között

```
PORTC = 0x00; // PORTC törlése
while(1) // Végtelen ciklus
{
    /*chLedek változó alsó négy bitjén balra rotálás*/
    chLedek = (chLedek&0x08) ? 1 : chLedek << 1;
    PORTC = chLedek; // ledék változó értékét
                      // kitesszük a kimenetre
    for(uiIdo=0; uiIdo<10000; uiIdo++)
    {
        NOP(); // Szoftveres késleltetés:
    }
}
```

A PIC12/16 mikrokontroller-család bizonyos mikrokontrollerei egy megszakításvektorral rendelkeznek. Abban az esetben, ha egy megszakításforrásból egy megszakításkérés érkezik, akkor a vezérlés a 0x0004 programmemória-címre kerül.

Azt, hogy egy függvény a megszakításkezelő rutin szerepét látja el, a függvény definíciójában elhelyezett `interrupt` kulcsszó használatával jelezhetjük a fordítónak. Ebben az esetben a fordító automatikusan menti a rutin által használt regisztereit a megszakításrutin elején. A megszakításrutin végén a fordító visszaállítja a rutin elején elmentett regiszterértékeit, és a `RETFIE` utasítással fejezi be a rutin végrehajtását.

Mivel a mikrokontroller architektúrája csak egy megszakításvektorral rendelkezik, az összes megszakításforrás ugyanarra a megszakításvektorra adja át a vezérlést. A megszakításrutint érdemes egy `if` szerkezetre felépíteni. Az elágazás feltételében megadjuk az adott megszakításhoz tartozó megszakításbitet, a feltétel igaz ágában elvégezzük az adott megszakításhoz tartozó feladatot, majd az ág végén töröljük a megszakításhoz tartozó megszakításbitet.

A következő programkód a megszakításrutin általános formáját mutatja be:

```
/* Megrakításkezelő függvény */
void interrupt InterruptHandler ()
{
    if(T0IF) /* Timer0 megszakítás */
    {
        /* ... */
        T0IF = 0; // Timer0 megszakításbitjének törlése
    }
}
```

A következő mintapélda az előző mintapélda továbbfejlesztett változata. A futófény új állapotát a **Timer1** időzítőhöz tartozó megszakításrutin fogja kiszámítani. Az időzítöt úgy állítjuk be, hogy a megszakítás 250 ms-onként érkezzen meg. A belső oszcillátor frekvenciája 4 MHz.

18.7. mintaprogram

```
#include <pic.h>
/* Belső osc. használata, WDT kikapcsolása, RESET láb engedélyezése */
__CONFIG (INTIO & WDTON & MCLREN);

/* Globális változó, futófény állapotát tárolja*/
volatile unsigned char chLedek = 1;

void InitT1 ( void ); // Timer1 modult inicializáló függvény
void InitInterrupt ( void ); // Megszakítások konfigurálása
```

```

main ( )                                // Főprogram kezdete
{
    InitT1();                            // Timer1 inicializálása
    InitInterrupt ();                   // Megszakítások konfigurálása
    TRISC = 0xF0;                      // PORTC alsó négy bitje kimenet lesz.
    PORTC = 0x00;                       // PORTC törlése

    while(1)                            // Végtelen ciklus
    {
        PORTC = chLedek;               // ledek változó értékét
        // kitesszük a kimenetre
    }

/* Timer1 modult inicializáló függvény */
void InitT1 ( void )
{
    /* Belső oszcillátor (Fosc/4), 1:8 előosztó */
    T1CON = 0b00110000;
    TMR1H = 0x85;                      // Időzítési ciklus: 250ms
    TMR1L = 0xEE;                      // TMR1 = 65536 - 31250;
    TMR1IF = 0;                        // Timer1 megszakításbitjének törlése
    TMR1ON = 1;                        // Timer1 bekapcsolása
}

/* Megszakítások konfigurálását végző függvény */
void InitInterrupt ( void )
{
    TMRIIE = 1;                        // Timer1 megszakításának engedélyezése
    PEIE = 1;                          // Perifériamegszakítások engedélyezése
    GIE = 1;                           // Összes megszakítás engedélyezése
}

/* Megszakításkezelő függvény */
void interrupt InterruptHandler (void)
{
    if(TMRIIF) /* Timer1 megszakítás */
    {
        // Időzítési ciklus újból beállítása
        TMR1H = 0x85;                  // Időzítési ciklus: 250ms
        TMR1L = 0xEE;                  // TMR1 = 65536 - 31250;
        /*chLedek változó alsó négy bitjén balra rotálás*/
        chLedek = (chLedek&0x08) ? 1 : chLedek << 1;
        TMR1IF = 0;                   // Timer1 megszakításbitjének törlése
    }
}

```

18.2.10. Több megszakításforrás használata

A következő mintapélda az előző mintapélda folytatása. Módosítsuk az előző programunkat úgy, hogy a program futása közben a **Timer1** időzítő időalapját az **AN0** lábra kötött potenciometré segítségével meg tudjuk változtatni. Ehhez be kell konfigurálnunk az **analóg-digitális átalakítót** is.

Az aktuális időalapot az **uiIldo** változó fogja tárolni. Mivel a változó **Timer1** és az analóg-digitális átalakító megszakítása közötti kommunikációjára szolgál, ezért nem muszáj globális változóként, hanem elég a megszakítás rutinra lokális, statikus változóként deklarálni.

```

#include <pic.h>
/* Belső osc. használata, WDT kikapcsolása, RESET láb engedélyezése */
_CONFIG (INTIO & WDTON & MCLREN);

/*Globális változó, futófény állapotát tárolja*/
volatile unsigned char chLedek = 1;

void InitT1 ( void );           // Timer1 modult inicializáló függvény
void InitAD ( void );          // A/D konvertert inicializáló függvény
void InitInterrupt ( void );   // Megszakítások konfigurálása

main ( )                      // Főprogram kezdete
{
    InitT1();                  // Timer1 inicializálása
    InitAD();                  // A/D konverter inicializálása
    InitInterrupt ();          // Megszakítások konfigurálása
    TRISC = 0xF0;              // PORTC alsó négy bitje kimenet lesz.
    PORTC = 0x00;               // PORTC törlése

    while(1)                  // Végtelen ciklus
    {
        PORTC = chLedek;       // ledek változó értékét
        // kitesszük a kimenetre
    }

/* Timer1 modult inicializáló függvény */
void InitT1 ( void )
{
    /* Belső oszcillátor (Fosc/4), 1:8 előosztó */
    T1CON = 0b00110000;
    TMR1H = 0x85;              // Időzítési ciklus: 250ms
    TMR1L = 0xEE;              // TMR1 = 65536 - 31250;
    TMR1IF = 0;                // Timer1 megszakításbitjének törlése
    TMR1ON = 1;                // Timer1 bekapcsolása
}

/* Analóg Digitális Konvertert inicializáló függvény */
void InitAD ( void )
{
    ANSEL = 1;                 // Az összes analóg bemenet kikapcsolása,
    ANSELM = 0;                 // kivéve az AN0 bemenetet.
    ADCON1 = 0b01110000;         // Időzítés: belső RC osc.
    ADCON0 = 0b10000001;         // 10 bites A/D, AN0 csatorna kiválasztása
    GODONE = 1;                 // Első A/D konverzió indítása
}

/* Megszakítások konfigurálását végző függvény */
void InitInterrupt ( void )
{
    TMRIIE = 1;                // Timer1 megszakításának engedélyezése
    ADIE = 1;                   // A/D megszakításának engedélyezése
    PEIE = 1;                   // Perifériamegszakítások engedélyezése
    GIE = 1;                    // Összes megszakítás engedélyezése
}

```

```

/* Megszakításkezelő függvény */
void interrupt InterruptHandler (void)
{
    static unsigned int uiIdo; // Késleltetés időalapja
    if(TMR1IF) // Timer1 megszakítás
    {
        TMR1H = (uiIdo>>8); // uiIdo felső 8 bitje
        TMR1L = uiIdo & 0xFF; // uiIdo alsó 8 bitje
        /*chLedek változó alsó négy bitjén balra rotálás*/
        chLedek = (chLedek&0x08) ? 1 : chLedek << 1;
        TMR1IF = 0; // Timer1 megszakításbitjének törlése
    }
    if (ADIF) // A/D konverter megszakítás
    {
        /* Időzítés kiszámítása */
        uiIdo = 0xFFFF - ((unsigned int)ADRESH << 8) | ADRESL;
        uiIdo <= 4;
        ADIF = 0; // A/D megszakításbitjének törlése
        GODONE = 1; // A/D konverzió indítása
    }
}

```

18.3. ÁTTÉRÉS A MICROCHIP C32 FORDÍTÓJÁRA

32/32 A Microchip MPLAB C32 fordítója a PIC 32 bites mikrokontroller-családhoz készített ANSI C kompatibilis fordító. A PIC 32 bites mikrokontrollerei a MIPS32® M4K™ 32-Bit processzormagon alapulnak, a regiszterei 32 bitesek.

A Microchip C32 fordító tanuló-, ingyenes változata megtalálható a könyv DVD-mellékletében, vagy a legfrissebb változata elérhető a <http://www.microchip.com/c32> webcímen keresztül.

18.3.1. A fordító könyvtárszerkezete

A fordító alapértelmezett esetben a C:\Program Files\Microchip\MPLAB C32 Suite könyvtárba települ fel, föbb alkönyvtárai a következők:

- A **bin** könyvtárban a fordító futtatható állományai találhatók.
- A **doc** könyvtárban a fordítóhoz tartozó dokumentáció található.
- A **pic32mx\include** könyvtárban a standard C és a mikrokontroller-specifikus fejlécállományok találhatók.
- A **pic32mx\lib** könyvtárban a standard C és a mikrokontroller-specifikus előfordított könyvtárak találhatók, o kiterjesztéssel.
- A **pic32mx\lib\proc** könyvtár mikrokontroller-specifikus alkönyvtáraiban az adott mikrokontroller linker állománya található, Id kiterjesztéssel.

18.3.2. Az első projekt elkészítése

A következő pár lépés segítségével az első 32 bites C programunk környezetét fogjuk elkészíteni. A programokat, a 16 bites mikrokontrollerekre írt programok tesztelésénél, használt Explorer 16 fejlesztőpanel segítségével fogjuk tesztelni, azzal a különbséggel, hogy az eredeti mikrokontroller helyett a **PIC32 Plug-in Module-t** fogjuk használni. A **PIC32 Plug-in Module** a **PIC32MX360F512L** típusú mikrokontrollert tartalmazza.

A projekt létrehozásának menete a következő:

- 1) Indítsuk el az MPLAB IDE-t, majd hozunk létre egy új projektet a varázsló segítségével, amelyet a **Project → Project Wizard...** menüpont alatt találunk.

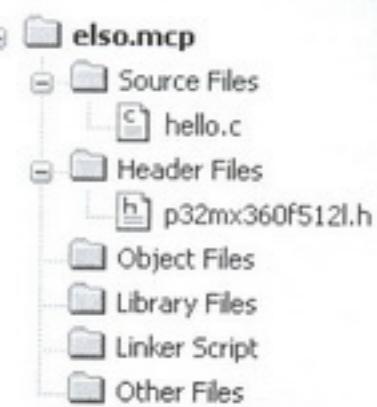
- 2) Az üdvözlőszöveget tartalmazó oldal után az eszköz (mikrokontroller) típusának kiválasztása következik. Itt a legördülő menüben a **PIC32MX360F512L** típust válasszuk ki.
- 3) A következő oldalon a fordítóeszközöt kell kiválasztani, itt a **Microchip PIC32 Compiler Toolsuite**-ot válasszuk ki.
- 4) Következő lépés a projektállomány létrehozása. Hozunk létre egy **ElsőC32** nevű könyvtárat és abban egy projektet **első** névvel.

A Microchip C32 fordítójának használatakor nem szükséges a linker állományt a projekthez csatolni, mert a fejlesztőkörnyezet automatikusan betölti a mikrokontroller típusához tartozó linker állományt.

Ezzel a négy lépéssel létre is hoztunk egy új projektet. Következő lépésként adjunk hozzá a projektünkhez egy üres **hello.c** nevű állományt. Érdemes még a projektünkhez a mikrokontroller típusához tartozó fejlécállományt is hozzáadni, amelynek neve: **p32mx360f512I.h**. Az állomány megtalálható a fordító **pic32mx\include\proc** alkönyvtárban.

Ha minden helyesen végeztünk el, akkor a projektünk a 18.8. ábrán látható állományokat tartalmazza.

Ahogy az már az előző fejezetekből kiderült, az Explorer 16 fejlesztőpanelen, a mikrokontroller **A portjának** alsó nyolc helyi értékű lábán LED-ek találhatók. Az első programunk feladata legyen az, hogy a gyakorlópanelen lévő LED-eket bekapsolja.



18.8. ábra
C32 projekt

18.9. mintaprogram

```

#include <p32xxxx.h>

#pragma config POSCMOD = XT // XT oszcillátor használata(3.5 MHz-10
MHz)
#pragma config FNOSC = PRIPLL // Orajel forrása: Külső oszcillátor + PLL
#pragma config FPLLDIV = DIV_2 // 1:2 előosztó használata (PLL)
#pragma config FPLLMUL = MUL_18 // 18xPLL szorzó használata
#pragma config FPLLODIV = DIV_1 // 1:1 utóosztó használata (PLL)
#pragma config FWDTEN = OFF // Watch Dog Timer kikapcsolása

main ( )
{
    DDPCONbits.JTAGEN = 0; // JTAG port kikapcsolása
    TRISA = 0xFF00; // PORTA alsó 8 bitje kimenet lesz.
    while(1)
    {
        LATA = 0xFF; // PORTA alsó 8 bitje egy lesz.
    }
}

```

A C32 fordító használata esetén a mikrokontroller regisztereinek használatakor kötelezően az általános mikrokontroller-állományt kell meghívni. Az általános fejlécállomány a közös definíciókat tartalmazza, és egyben betölti az adott típushoz tartozó fejlécállományt is. Az általános mikrokontroller-fejlécállomány neve: **p32xxxx.h**.

18.3.3. Konfigurációs bitek beállítása

A C32 fordító esetén a konfigurációs bitek beállításának módszere nagyban hasonlít a Microchip C18 fordítójánál alkalmazott megoldáshoz. A fordítónak a konfigurációs biteket a #pragma config előfordítói utasítás segítségével tudjuk megadni. Az utasítás paramétereként meg kell adni, hogy az adott konfigurációs tulajdonság milyen értéket vegyen fel. Például a POSCMOD = XT kifejezés azt jelenti, hogy az mikrokontroller külső, XT típusú (3,5 MHz és 10 MHz közötti) oszcillátort használjon.

Az egyes mikrokontroller-típusokhoz tartozó konfigurációs bitek beállításai megtalálhatók a fordító doc alkönyvtárának **hlpPIC32MXConfigSet.chm** nevű állományában.

Egy konfigurációs utasításban nemcsak egy érték állítható be, hanem vesszővel elválasztva több beállítás is elvégezhető egyszerre. Például az előző példában elvégzett konfigurációs beállítások egy sorban is leírható:

```
#pragma config POSCMOD = XT, FNOSC = PRIPLL // Külső XT oszcillátor + PLL
```

Az Explorer 16 fejlesztőpanel egy 8 MHz-es külső kvarcot tartalmaz. A konfigurációs bitek beállításánál a PLL modul bemenetére érkező frekvenciát megfelezzük az FPLLIDIV = DIV_2 beállítás segítségével. Az így kapott frekvenciát, a FPLLMUL = MUL_18 konfigurációs beállítás hatására, a PLL modul 18-szoros szorzó használatával megnöveli. A felsorozott frekvencia az FPLLODIV = DIV_1 beállítás hatására 1:1 utóosztón keresztül a mikrokontroller belső órajele lesz. A most alkalmazott beállítással a mikrokontroller 72 MHz-es órajel segítségével hajtja végre az utasításokat.

A JTAG port be- vagy kikapcsolását a 16 bites kontrollerek esetén a konfigurációs bitek segítségével lehet beállítani. A 32 bites mikrokontrollerek esetén a JTAG port állapota a program futtatása közben is módosítható. A JTAG portot a DDPCON regiszter JTAGEN bitjének segítségével lehet be- vagy kikapcsolni.

18.3.4. Regiszterek és bitek használata

A speciális funkcióregiszterek használata, így például a portok is, megegyezik a C30 fordítónál használt módszerrel. A PIC 32 bites családjába tartozó mikrokontrollerek regisztereinek nagysága 32 bit, de sok funkcióregiszterben csak az alsó 16 bit van felhasználva, mint például a portok regiszterei esetén is.

Az egyes funkcióregiszterek bitjei – a C30 fordítónál már megismert bitstruktúrák segítségével – a mikrokontroller fejlécállományában vannak definiálva. Például az A port ötödik bitjét a következő kifejezéssel állíthatjuk egybe:

```
LATAbits.LATA5 = 1;
```

18.3.5. Változótípusok és mutatók

A Microchip MPLAB C32 fordítója által használt változók hasonlítanak a C30 fordítónál megismert változótípusokhoz. A különbség a két fordító által használt típusok között az int és long változótípusnál van, mert a C32 fordító esetén az int típusú változók 32 bit, a long típusú változók pedig 64 bit szóhosszúságúak. A következő táblázat a Microchip C32 fordító által használt, egész számok tárolására alkalmas változótípusokat foglalja össze.

18. fejezet: Átjárás a világok között

| Típus | Méret | Minimum | Maximum |
|------------------------------------|--------|-----------|------------|
| char, signed char | 8 bit | -128 | 127 |
| unsigned char | 8 bit | 0 | 255 |
| short, signed short | 16 bit | -32,768 | 32,767 |
| unsigned short | 16 bit | 0 | 65,535 |
| int, signed int, long, signed long | 32 bit | -2^{31} | $2^{31}-1$ |
| unsigned int, unsigned long | 32 bit | 0 | $2^{32}-1$ |
| long long, signed long long | 64 bit | 2^{63} | $2^{63}-1$ |
| unsigned long long | 64 bit | 0 | $2^{64}-1$ |

18.9. ábra

A Microchip C32 fordító egész típusú változói

Elérés van a két fordító által használt, lebegőpontos számok ábrázolására alkalmas változótípusok között is. A C30 fordító esetében a double típusú változó hossza állítható volt, a C32 fordító esetén a double változótípus hossza fixen 64 bit, így a double és a long double típusú változók bithosszúsága megegyezik. A következő táblázat a Microchip C32 fordító által használt, lebegőpontos számok tárolására alkalmas változótípusokat foglalja össze.

| Típus | Méret |
|-------------|--------|
| float | 32 bit |
| double | 64 bit |
| long double | 64 bit |

18.10. ábra

A Microchip C32 fordító lebegőpontos típusú változói

A PIC 32 bites családba tartozó mikrokontrollerek címbusza 32 bit szélességű, ezért a mutató típusú változók nagysága 32 bit.

18.3.6. Inline Assembly

A Microchip C32 fordító is lehetőséget biztosít arra, hogy assembly utasításokat ágyazzunk be a C nyelvű kódunkba. Az assembly kódok beillesztése megegyezik a C30 fordítónál megismert szintaktikával.

A PIC 32 bites mikrokontrollerjeinek utasításkészlete jóval összetettebb, mint a 16 bites mikrokontrollereké, ezért a beágyazott assembly utasítások használatát lehetőleg kerüljük.

Ahogy a C30 fordító mikrokontroller-specifikus fejlécállományában is definíálva voltak, a C32 fordító általános mikrokontroller-fejlécállományában (**p32xxxx.h**) is definíálásra kerültek bizonyos alap assembly utasítások inline assembly makrói.

Az inline assembly makrók elnevezése eltér a C30 fordítónál megismert elnevezési szabálytól. A C32 fordító inline assembly makrói aláhúzásjellel kezdődnek, és végig kisbetűvel vannak írva. Például egy **ssnop** assembly utasítást a **_nop()** makró segítségével tudunk a programkódunkban elhelyezni. A következő sor a **p32xxxx.h** állományból származik, és a **_nop()** makró definícióját szemlélteti:

```
#define _nop() __asm__ __volatile__ ("%(ssnop%)" : :)
```

Egy következő példa egy 32 bites, long típusú változót eggyel balra forgat:

```
_asm volatile (*rotr %0, 1 : "+r"(longTipusuValtozo));
```

18.3.7. Megszakítások használata

Mielőtt megismerkedünk a megszakítások használatával, nézzük egy példát a szoftveres késleltetés elkészítéséhez. Implementáljuk a már jól ismert futófényprogramunkat a 32 bites mikrokontrollerre is. A futófényprogram lesz az alapja a későbbi, időzítő megszakítással ellátott programunknak. A futófény állapotának megváltoztatására nem tudjuk a hardver által biztosított rotálást kihasználni, mert rotálni csak 32 bites változót tudunk, nekünk pedig 8 bites változón kell elvégeznünk.

18.10. mintaprogram

```
#include <p32xxxx.h>

#pragma config POSCMOD = XT, FNOSC = PRIPLL // Külső XT oszcillátor + PLL
// 72 MHz órajel előállítása

#pragma config FPLLMUL = MUL_18, FPLLDIV = DIV_1
#pragma config FWDTEN = OFF // Watch Dog Timer
kikapcsolása

main ( ) // Főprogram kezdete
{
    unsigned char chLedek = 1; // A futófény állapota
    unsigned int uiIdo; // Késleltetéshez használt változó

    DDPCONbits.JTAGEN = 0; // JTAG port kikapcsolása
    TRISA = 0xFF00; // PORTA alsó 8 bitje kimenet lesz.
    PORTA = 0; // PORTA törlése

    while(1)
    {
        /*chLedek változó alsó nyolc bitjén balra rotálás*/
        chLedek = (chLedek&0x80) ? 1 : chLedek << 1;
        LATA = chLedek; // ledek változó értékét
        // kitesszük a kimenetre

        for(uiIdo=0; uiIdo<30000; uiIdo++)
        {
            _nop(); // 30000-szer fut le a nop utasítás.
        }
    }
}
```

A PIC32 mikrokontroller-család architektúrája maximum 96 megszakításforrás kezelésére alkalmas. A megszakításkezelő rendszert két üzemmódban lehet használni. Lehetőség van egy megszakításvektor használatára, ilyen esetben az összes engedélyezett megszakítás egy megszakításvektorra érkezik meg. Ez a megoldás a 8 bites architektúránál alkalmazott megoldáshoz hasonlít. A második lehetőség szerint az összes megszakításforráshoz (megszakításforrás-családhoz) külön megszakításvektort rendelünk.

Először ismerkedjünk meg az egyvektoros megszakításkezelő rendszer használatával. Ha jelezük a fordítókörnyezet számára, hogy egy függvényt megszakításkezelő rutinnak szeretnénk használni, gondoskodik az adott függvény megfelelő címre történő elhelyezéséről (hasonlóan, mint a C30 fordítókörnyezet).

A megszakításkezelő függvényünk definíciójakor meg kell adnunk a fordító számára, hogy a függvény melyik megszakításvektor aktivizálódásakor kerüljön meghívásra, és hogy az adott megszakításrutin milyen prioritású. A fordító két szintaktikai megoldást

biztosít a programozó számára ahhoz, hogy a függvényt megszakításkezelő rutinná módosítsa.

Az első megoldás hasonlít a C18 fordító által használt, #pragma interrupt kulcsszó használatával történő definiáláshoz. A #pragma interrupt utasítás után meg kell adni a megszakításrutin nevét és a prioritását, az utasítás vektor paramétere után a megszakításrutin helyét. A megszakításrutin prioritása ipl0 – ipl7-ig terjedhet.

Egy olyan megszakításrutin definícióját, amely egyes prioritással rendelkezik, és a nullás vektorra ül rá, a következőképpen kell megadni:

```
#pragma interrupt InterruptHandler ipl1 vector 0
void InterruptHandler ( void )
{
    /* Megszakításkezelő függvény belseje */
}
```

Abban az esetben, ha egy megszakításrutint használunk, nem kell prioritást rendelnünk a megszakításkezelő függvényünkhez, hanem elég helyette egy single kifejezést használnunk. Ha a mikrokontroller egy megszakításvektoros üzemmódban van, akkor az összes megszakítás a nullás megszakításvektorra érkezik meg. Ilyen esetben a megszakításkezelő függvényünk definíciója a következőképpen alakul:

```
#pragma interrupt InterruptHandler single vector 0
void InterruptHandler ( void )
{
    /* Megszakításkezelő függvény belseje */
}
```

A másik fajta szintaktika a C30 fordítónál használt, __attribute__ kulcsszón alapszik. Ilyen esetben is meg kell adni a megszakításrutin prioritását és a vektorcímét is.

```
void __attribute__ (( interrupt(ipl1), vector(0) ))
InterruptHandler ( void )
{
    /* Megszakításkezelő függvény belseje */
}
```

Azért, hogy ne kelljen állandóan ilyen hosszú kifejezést használni, lehetőségünk van előre definiált __ISR(,) makró segítségével kiváltani az __attribute__ kifejezést. A makró definíciója a sys/attribs.h fejlécállományban található. Az __ISR(,) makró használatával a megszakításrutin definíciója a következőképpen alakul:

```
void __ISR( 0, ipl1 ) InterruptHandler ( void )
{
    /* Megszakításkezelő függvény belseje */
}
```

Megszakítások használatakor érdemes megismerkedni még egy fejlécállománnyal, a peripheral\int.h-val. A programban érdemes közvetett módon meghívni, méghozzá a plib.h segítségével. Ezek a fejlécállományok kényelmesen használható megszakításkezelő függvények deklarációját tartalmazzák. A függvények és makrók elnevezései meglehetősen hosszúak, de egyértelműen beazonosíthatóak. A következő felsorolás, a teljesség igénye nélkül, néhány fontosabb függvény (F), illetve makró (M) nevét és működését írja le:

- INTEnableSystemSingleVectoredInt(); F Egyvektoros megszakításrendszer engedélyezése
- INTEnableSystemMultiVectoredInt(); F Multivektoros megszakításrendszer engedélyezése
- mXXSetIntPriority(x); M Adott megszakításforrás prioritásának beállítása. Az xx helyére a módosítani kívánt megszakításforrás nevét kell beírni. Például ha a Timer1 tűlcsordulásbitjéhez tartozó megszakításnak szeretnénk megváltoztatni a prioritását, akkor azt az mT1SetIntPriority(x); makró segítségével tudjuk megadni. A lista további makróit is az adott megszakításhoz tartozó behelyettesítéssel kell értelmezni.
- mXXIntEnable(x); M Adott megszakításforrás engedélyezését vagy tiltását hajtja végre. Engedélyezés: x=1, tiltás: x=0.
- mXXGetIntFlag(); M Adott megszakításhoz tartozó jelzőbit értékével tér vissza.
- mXXClearIntFlag(); M Az adott megszakításhoz tartozó jelzőbitet törli.

Most, hogy minden ismeret a rendelkezésünkre áll ahhoz, hogy elkészítsük az első egyvektoros megszakításrutinunkat, módosítsuk a futófényprogramunkat úgy, hogy a futófény új állapotát a **Timer1** időzítőhöz tartozó megszakításrutin számítsa ki. Az időzítő úgy állítjuk be, hogy a megszakítás **250 ms**-onként érkezzen meg. A PIC 32 bites PIC mikrokontrollerek perifériákészlete nagyon hasonlít a PIC 16 bites PIC mikrokontroller-család perifériakészletéhez. Azért, hogy a **Timer1** időzítését ne kelljen megváltoztatni a 17.3. fejezetben elkészített programunkhoz képest, a mikrokontroller órajel-frekvenciáját módosítsuk **32 MHz**-re. A PIC 32 bites kontrollereiben állítható, hogy a perifériák a processzor órajeléhez képest hányadrásszel dolgozzanak. Ez az érték alapesetben **1:8** osztót jelent, amit a programunk nem változtat meg.

Mielőtt a programot elkészítenénk, érdemes pár szót ejteni arról, hogyan lehet a PIC 32 bites mikrokontrollereit energiatakarékos üzemmódba kapcsolni. Az energiatakarékos üzemmódokat beállító függvények deklarációi a *peripheral/power.h* fejlécállományban találhatók. A fejlécállományt nem kell külön behívni a programunkban, mert a *plib.h* fejlécállomány ezt a fejlécállományt is betölti. A fejlécállomány két függvénydeklarációt tartalmaz:

- PowerSaveSleep(); A mikrokontroller **alvó (sleep)** üzemmódba kerül.
- PowerSaveIdle(); A mikrokontroller **idle** üzemmódba kerül.

18.11. mintaprogram

```
#include <p32xxxx.h>
#include <plib.h>

#pragma config POSCMOD = XT, FNOSC = PRIPLL // Külső XT oszcillátor + PLL
// 32 MHz órajel előállítása
#pragma config FPLLIDIV = DIV_2, FPLLMUL = MUL_16, FPLLORDIV = DIV_2
#pragma config FWDTEN = OFF // Watch Dog Timer kikapcsolása
```

```
/*Globális változó, futófény állapotát tárolja*/
volatile unsigned char chLedek = 1;

void InitT1 ( void ); // Timer1 modult inicializáló függvény
void InitInterrupt ( void ); // Megszakítások konfigurálása

main ( ) // Főprogram kezdete
{
    DDPCONbits.JTAGEN = 0; // JTAG port kikapcsolása
    InitT1(); // Timer1 inicializálása
    InitInterrupt (); // Megszakítások konfigurálása
    TRISA = 0xFF00; // PORTA alsó 8 bitje kimenet lesz.
    PORTA = 0; // PORTA törlése
    while(1)
    {
        LATA = chLedek; // ledék változó értékét
        PowerSaveIdle(); // Kitesszük a kimenetre
    }
}

/* Timer1 modult inicializáló függvény */
void InitT1 ( void )
{
    T1CONbits.TCS = 0; // Belső órajel használata (Fosc/8)
    T1CONbits.TCKPS = 2; // 1:64 frekvenciaosztó használata
    T1CONbits.TGATE = 0; // Kapuzó áramkör letiltása
    T1CONbits.TSIDL = 0; // IDLE állapotban is fusson az áramkör
    TMRI = 0; // Timer1 számlálójának törlése
    PR1 = 15625; // Időzítési ciklus: 250ms
    mT1ClearIntFlag(); // Timer1 megszakításbitjének törlése
    T1CONbits.TON = 1; // Timer1 bekapcsolása
}

/* Megszakítások konfigurálását végző függvény */
void InitInterrupt ( void )
{
    INTEnableSystemSingleVectoredInt(); // Egyvektoros megszakításrendszer
    mT1SetIntPriority( 4 ); // Timer1 megszakítás prioritásszintjének
    // beállítása
    mT1IntEnable(1); // Timer1 megszakításának engedélyezése
}

/* Megszakításrutin */
#pragma interrupt InterruptHandler single vector 0
void InterruptHandler ( void )
{
    /*chLedek változó alsó nyolc bitjén balra rotálás*/
    chLedek = (chLedek&0x80) ? 1 : chLedek << 1;
    mT1ClearIntFlag(); // Timer1 megszakításbitjének törlése
}
```

A megszakításkezelő makrók használata csak a programunk átláthatóságát növeli, nem kell kötelező jelleggel alkalmazni. Például az *mT1ClearIntFlag()*; makróhívás helyett a PIC24 mikrokontrollereknél megszokott *IFS0bits.T1IF = 0;* megoldás is használható.

Abban az esetben, ha a programot ICD2-vel futtatjuk, előfordulhat, hogy a program futása debugolás alatt, Idle üzemmódban megszakad. Ilyen esetben a program debugolása idejére érdemes a főprogramban található PowerSaveIdle(); függvényhívást kivenni.

18.3.8. Több megszakításvektor használata

Ebben a fejezettrészben két, különböző megszakításvektoron található, különböző prioritású megszakításról fogunk elkezdeni. Módosítsuk az előző programot úgy, hogy a program futása közben a Timer1 időzítő időalapját az AN5 lábra kötött potenciometter segítségével módosítani tudjuk. Az analóg–digitális átalakító megszakítását alacsony prioritásra fogjuk állítani azért, hogy az időzítésünk pontos legyen.

Az előző programban elkészített Timer1 időzítő konfigurációjához képest módosítsuk az előosztó nagyságát 1:256-ra, hogy nagyobb időzítési határok között mozogjon a programunk. Az előző programunk átírásakor ne feledkezzünk meg a több megszakításvektoros üzemmód aktiválásáról. A több megszakításvektoros üzemmódot az INTEnableSystemMultiVectoredInt(); függvény meghívásával tudjuk bekapsolni. A PIC32MX360F512L mikrokontroller adatlapja 8. fejezetének végén található, a megszakításvektorok címét tartalmazó táblázatból kiderül, hogy a Timer1 időzítőhöz tartozó megszakításvektor a 4-es, az AD1 konverterhez tartozó megszakításvektor pedig a 27-es.

18.12. mintaprogram

```
#include <p32xxxx.h>
#include <plib.h>

#pragma config POSCMOD = XT, FNOSC = PRIPLL // Külső XT oszcillátor + PLL
// 32 MHz órajel előállítása
#pragma config FPLLIDIV = DIV_2, FPLLMUL = MUL_16, FPLLODIV = DIV_2
#pragma config FWDTEN = OFF // Watch Dog Timer
kikapcsolása

/* Globális változó, futófény állapotát tárolja */
volatile unsigned char chLedek = 1;
/* Globális változó, futófény sebessége */
volatile unsigned long uiIdo = 1;

void InitT1 ( void ); // Timer1 modult inicializáló függvény
void InitAD1 ( void ); // A/D konvertert inicializáló függvény
void InitInterrupt ( void ); // Megszakítások konfigurálása

main ( )
{
    DDPCONbits.JTAGEN = 0; // JTAG port kikapcsolása
    InitT1(); // Timer1 inicializálása
    InitAD1(); // A/D konverter inicializálása
    InitInterrupt (); // Megszakítások konfigurálása
    TRISA = 0xFF00; // PORTA alsó 8 bitje kimenet lesz.
    PORTA = 0; // PORTA törlése
    while(1)
    {
        LATA = chLedek; // ledek változó értékét
        // kitesszük a kimenetre
        PowerSaveIdle(); // A mikrokontroller Idle üzemmódba kerül
    }
}
```

```
)
/* Timer1 modult inicializáló függvény */
void InitT1 ( void )
{
    T1CONbits.TCS = 0; // Belső órajel használata (Fosc/8)
    T1CONbits.TCKPS = 3; // 1:256 frekvenciaosztó használata
    T1CONbits.TGATE = 0; // Kapuzó áramkör letiltása
    T1CONbits.TSIDL = 0; // IDLE állapotban is fussen az áramkör
    TMRI = 0; // Timer1 számlálójának törlése
    PR1 = uiIdo; // Első időzítés
    mT1ClearIntFlag(); // Timer1 megszakításbitjének törlése
    T1CONbits.TON = 1; // Timer1 bekapsolása
}

/* Analóg Digitális Konvertert inicializáló függvény */
void InitAD1 ( void )
{
    AD1PCFG = 0xFFDF; // AN5 analóg bemenet engedélyezése,
    // a többi bemenet digitális I/O lesz
    AD1CON1 = 0x00E4; // Automatikus konverzió indítása
    // mintavételezés után, konverzió után
    AD1CON2 = 0; // Új mintavételezés indítása
    // MUXA használata, a Vref+/- feszültségekkel
    // a AVss és a AVdd használata.
    AD1CON3 = 0x1F00; // Tsamp = 31 x Tad; Tad=125ns (Tad = 1 Tcy)
    AD1CHSbits.CH0SA = 5; // MUXA: AN5 bemenet kiválasztása
    AD1CHSbits.CH0NB = 0; // MUXB: Negatív feszültségszint kiválasztása
    AD1CON1bits.ADON = 1; // A/D konverter bekapsolása
}

/* Megszakítások konfigurálását végző függvény */
void InitInterrupt ( void )
{
    INTEnableSystemMultiVectoredInt(); // Egyvektoros megszakításrendszer
    mT1SetIntPriority( 4 ); // Timer1 megszakítás prioritásszintjének
    // beállítása
    mT1IntEnable(1); // Timer1 megszakításának engedélyezése
    mAD1SetIntPriority( 3 ); // AD1 megszakítás prioritásszintjének
    // beállítása
    mAD1IntEnable(1); // AD1 megszakításának engedélyezése
}

/* Timer1 megszakításrutinja */
#pragma interrupt T1Interrupt ipl4 vector 4
void T1Interrupt ( void )
{
    /* chLedek változó alsó nyolc bitjén balra rotálás */
    chLedek = (chLedek&0x80) ? 1 : chLedek << 1;
    PR1 = uiIdo; // Időzítés beállítása
    mT1ClearIntFlag(); // Timer1 megszakításbitjének törlése
}

/* A/D konverter megszakításrutinja */
#pragma interrupt ADC1Interrupt ipl3 vector 27
```

```
void ADC1Interrupt ( void )
{
    uiIdo = ADC1BUF0+1;           // Időzítés beállítása
    mAD1ClearIntFlag();           // AD1 megszakításbitjének törlése
}
```

Ezzel a mintapéldával elérkeztünk a könyv C nyelvű programozással foglalkozó fejezeteinek végére. Bízunk benne, hogy a kötet elérte a célját, megfelelő segítséget nyújtott a C nyelvvel ismerkedő mikrokontroller-programozók számára ahhoz, hogy az elolvasása után minél hasznosabban tudjanak C nyelvű programokat fejleszteni.

Mindenkinék ajánljuk, hogy a későbbi programfejlesztés során látogasson el a Microchip cég honlapjára (<http://www.microchip.com>), ahol az egyes alkalmazásokhoz sok, dokumentációval ellátott C nyelvű mintapéldát talál.

19. MELLÉKLETEK

19.1. A MICROCHIP INTERNET-OLDALÁN KÖZVETLENÜL ELÉRHETŐ TÉMÁK GYŰJTEMÉNYE

A mai internetes világban a legfontosabb irodalomjegyzék a Microchip cég honlapja:

<http://www.microchip.com>

Az előző kiadásban szereplő irodalomjegyzék is fontos, de mivel az ott (és a DVD-melékleben) is szerepel, ezért itt nem ismételjük meg.

Néhány további forrás, a teljesség igénye nélkül:

<http://mikrovezerlo.lap.hu/>

<http://www.t-es-t.hu/index2.htm>

<http://piclist.com>

<http://www.microchipc.com/sourcecode> C nyelvű mintapéldák gyűjteménye.

PIC mikrovezérlők:

<http://www.microchip.com/8bit>
<http://www.microchip.com/pic10>
<http://www.microchip.com/versatile>
<http://www.microchip.com/pic12>
<http://www.microchip.com/pic18>
<http://www.microchip.com/pic16>
<http://www.microchip.com/pic24>
<http://www.microchip.com/pic32>
<http://www.microchip.com/datasheets>
<http://www.microchip.com/flash>

Analóg:

<http://www.microchip.com/analog>
<http://www.microchip.com/linear>
<http://www.microchip.com/mechatronics>
<http://www.microchip.com/nanowatt>
<http://www.microchip.com/3v>

Fejlesztő eszközök:

<http://www.microchip.com/tools>
<http://www.microchip.com/icd2help>
<http://www.microchip.com/icd2>
<http://www.microchip.com/pickit2>
<http://www.microchip.com/pickit1>
<http://www.microchip.com/ide>
<http://www.microchip.com/realice>
<http://www.microchip.com/mplab>
<http://www.microchip.com/mindi>

Memoriák:

<http://www.microchip.com/memory>
<http://www.microchip.com/serial>
<http://www.microchip.com/eeprom>

Oktatás:

<http://www.microchip.com/seminars>
<http://www.microchip.com/webseminars>
<http://www.microchip.com/masters>

Perifériák:

<http://www.microchip.com/temp>
<http://www.microchip.com/battery>
<http://www.microchip.com/rf>
<http://www.microchip.com/can>
<http://www.microchip.com/usb>
<http://www.microchip.com/mtouch>
<http://www.microchip.com/interface>
<http://www.microchip.com/tcpip>
<http://www.microchip.com/connectivity>
<http://www.microchip.com/wired>
<http://www.microchip.com/lin>
<http://www.microchip.com/lcd>
<http://www.microchip.com/ethernet>
<http://www.microchip.com/zigbee>

C nyelv:

<http://www.microchip.com/c18>
<http://www.microchip.com/c30>
<http://www.microchip.com/c32>

19.2. PIC MIKROVEZÉRLŐKKEL KAPCSOLATOS FOGALMAK GYŰJTEMÉNYE

A következőkben a PIC mikrovezérlőkkel kapcsolatos fogalomgyűjteményt közlünk, amelynek az eredetije a Microchip weblapján is megtalálható. A magyaráztatagyűjtemény angol nyelvű fogalmait meghagyuk, és megadtuk azok magyar nyelvű meghatározását.

Absolute Section Egy szekció rögzített címmel, amit a szerkesztő nem tud megváltoztatni.
Access RAM (PIC18XXX eszközökönél) Egy adatmemória-terület, ami az éppen aktuális kiválasztott banktól függetlenül elérhető. Az SFR regiszterek ilyen módon címzhetők, anélkül, hogy az aktuális bankot BSR bitekkel váltani kellene. Az Access RAM tartalmaz még a felhasználó által használt regisztereit (*General Purpose Registers – GPRs*) is. Ez nagyon hasznos, mert megőrizhetjük a változóink értékét, mikor átkapcsolunk egy másik programrészre (pl. megszakításkor).

Acquisition Time (TACQ) Ez a fogalom az Analog-Digital (A/D) átalakításhoz kapcsolódik. Az az idő, míg a PIC A/D tartó kapacitását (*holding capacitor*) tölti a rákapcsolt bemeneti feszültség. Mikor a GO bit 1 lesz, az analóg bemenet leválasztódik a kapacitásról, és elkezdődik az A/D átalakítás.

A/D Az Analog-Digital átalakító rövidítése.

ALU Arithmetical Logical Unit – aritmetikai-logikai egység. Digitális áramkörök, amelyek a matematikai (összeadás, kivonás...), logikai (és, vagy...) és eltolási műveleteket végzik.

Analog Olyan mennyiség, amelynek értéke nem diszkrét lépcsőkből áll, s amelyre azt mondjuk, hogy folyamatosan bármilyen értéket felvehet.

Analog-to-Digital (A/D) Analóg bemeneti feszültség átalakítása a vele arányos digitális számértékké.

Assembly Language Szimbólumokat használó programozási nyelv, amivel a gépi kódú bináris programot olvashatóan és érthetően tudjuk leírni.

AUSART Addressable Universal Synchronous Asynchronous Receiver Transmitter – címmezhető univerzális aszinkron adó-vevő. Ez a modul működhet egyszerre kétirányú (*full duplex*) aszinkron kommunikációs portként vagy váltakozóan egy irányú (*half duplex*) szinkron kommunikációs portként. Amikor aszinkron módban működik, akkor a PC soros portjával teremthet kapcsolatot.

Alphanumeric Alfanumerikus karakterek, a betük és a számok együtt.

Application Alkalmazás. A felhasználó által fejlesztett áramkör (hardver) és program (szoftver). Egy mikrovezérlővel működtetett berendezés.

Assemble Az assembler által végzett tevékenység.

Assembler Egy nyelvi eszköz, ami a felhasználó által készített assembly forráskódot (.asm) átalakítja gépi kódú utasítások sorozatára. Ilyen például a MPASM™ a Microchip's assemblerre.

Assembly Gépi kódú programok az utasításokat kódoló bitcsoportokból és az operandusokat ábrázoló bináris alakú számokból állnak, és az ember számára igen nehezen olvashatók. Az assembler programnak ugyanolyan a felépítése, mint a gépi kódú programnak, de az utasítások neveit könnyen megjegyezhető formában mnemonikokkal ábrázoljuk, míg a bináris alakú számok helyett szimbólumokat használunk.

Assigned Section Hozzárendelt szekció. Olyan programrész, amit a (linker) szerkesztő parancsfájlból (*.lkr) egy adott fizikai memóriablokkhoz rendel.

Bank Az adatmemória egyik címzési megoldása. Mivel az utasításokban szereplő adatmemória-cím hosszát az utasításhossz korlátozza, a fizikai memóriát bankokra osztjuk, és bankválasztó regiszter (BSR – *Bank Selection Register*) segítségével címzzük meg a fizikai adatmemoriát. Fizikai cím = BSR tartalma + az utasításban szereplő cím.

Baud Diszkrét jelek esetén a jelváltások száma másodpercenként. Ez határozza meg a kommunikáció sebességét, vagyis a másodpercenként átvitt információ mennyiségét. Kétállapotú jelek esetén a baud számértékileg a bit/sec-ban kifejezett sebességet jelenti.

BCD Lásd a „Binary Coded Decimal (BCD)” bejegyzésnél.

Binary Coded Decimal (BCD) minden négy bites egység (*nibble*) egy 0–9 számjegy bináris értékét jelenti. Általában egy bájt két ilyen számjegyet tartalmaz, így az értéke 0–99 között van.

BOR Lásd „Brown-out Reset (BOR)”.

Brown-out Egy esemény, mikor egy tok tápfeszültsége időlegesen egy minimális működési feszültség alá csökken. Például ez akkor következhet be, mikor a tok egy nagyobb terhelést kapcsol be.

Brown-out Reset (BOR) Áramkör, amely a tokot RESET-be kényszeríti, ha a tápfeszültsége egy megadott minimális működési feszültség alá csökken.

Bus width Buszsélesség, azon bitek száma, amit a busz egy időben, párhuzamosan továbbít. Adatmemoriánál ez általában 8, 16 vagy 32 bit. PIC mikrovezérlőknél a programbusz 12/14/16/24/32 bit szélességű lehet. Jelölése: adatbuszsélesség/programbuszsélesség. Például a PIC 18-as család 8/16-os eszköz.

Breakpoint – Hardware Egy esemény, aminek végrehajtása megállást okoz.

Breakpoint – Software Egy cím, amit elérve a program végrehajtása megáll, és a rendszer változói megtekinthetők.

Build Egy alkalmazást alkotó forrásfájlok újrafordítása.

C Magas szintű programozási nyelv, a mikrovezérlőknél is használt. Segítségével hozzuk létre (fordítás = kompilálás) a program gép kódú formáját.

Calibration Memory Egy SFR regiszter vagy regiszterek, amelyek a PICmicro® mikrovezérlő tokjában található RC oszcillátor beállítási (kalibrálási, jusztírozási) értékét tartalmazza.

CAN Controller Area Network Periféria-interfész, az autóiparban és ipari alkalmazásokban használt.

Capture „Elkapás” A CCP modul azon funkciója, amikor előre meghatározott esemény bekövetkezésének pillanatában a hozzárendelt időzítő/számláló számértékét egy tároló (*capture*) regiszterbe írja.

Capture Register Capture eseménykor az értéket tárolja.

CCP Capture, Compare, Pulse Width Modulation (PWM) Három funkciót egyesítő modul, számlálót és regisztereket tartalmaz. Egyszerre csak egy funkciót tud megvalósítani.

COFF Common Object File Format. Közös tárgykód-fájlformátum. Az MPLINK™ LINKER által generált közbenső fájlformátum, amely kód- és hibavadász (debugger) információt tartalmaz.

Command Line Interface Parancssoros interfész. Segítségével, paraméterek megadásával egy programot a DOS-ból indíthatunk. Kötegelt (*batch*) programvégrehajtást tesz lehetővé.

Common RAM Közös RAM. Az a RAM-terület, amely ugyanaz, függetlenül a bankválasztástól. Például 8/14-es PIC mikróknál ez a 70H–7FH című RAM-terület. Hasznos különböző bankokat használó taszkok közötti változóátadásnál.

Compare „Hasonlítás” A CCP modul azon funkciója, mikor a compare regiszterbe írt értékel a számláló értéke megegyezik, akkor a tok valamelyik kimenetén egy esemény következik be (pl. a láb állapotot vált).

Compare Register A számlálóhoz hasonlítandó értéket tároló regiszter.

Compile Fordítás, amit a compiler végez. Lásd compiler.

Compiler Egy magas szintű nyelven megírt programot gépi kódra fordító program. Pl. MPLAB® C18 a Microchip C fordítója a PIC18XXX család számára.

Configuration Bits A PICmicro® mikrovezérlő működési módját beállító bitek megadása. Egy konfigurációs bit bizonyos esetekben újraprogramozható. Ezek a bitek az MPLAB Options → Development Mode párbeszédbablakában vagy az MPASM assembler CONFIG direktívája segítségével állítható be.

Configuration Word A konfigurációs bitek, bitcsoporthoz vannak szervezve.

Conversion Time (Tconv) Analóg-digitál átalakítókkal kapcsolatos fogalom. Az az idő, ami szükséges az átalakítónak, hogy a tartókapacitáson (*holding capacitor*) megjelenő feszültséget bináris jelcsoporthoz alakitsa.

CPU Central Processing Unit. Központi feldolgozóegység: dekódolja az utasításokat, meghatározza az operandusokat, ami a program végrehajtásához szükséges. Az aritmétikai, logikai és léptető utasításokat az ALU (aritmétikai-logikai egység) hajtja végre.

D/A Lásd a „Digital to Analog” bejegyzésnél.

DAC Digital-to-Analog Converter. Digitális-analóg átalakító.

Data Bus Az adatmemoriához kapcsolódó adatforgalmat biztosító jelvezetékek összessége.

Data EEPROM Data Electrically Erasable Programmable Read Only Memory. A mikrovezérlőben lévő, olyan adatokat tároló memória, amiben programutasítások segítségével adatokat tudunk tárolni, illetve kiolvasni. A tápfeszültség kikapcsolása után is megőrzi a tartalmát.

Data Memory Az adatbuszhoz kapcsolódó írható/olvasható memória, más néven fájlregisztertömb. A tápfeszültség kikapcsolása után elveszíti a tartalmát. Két részből áll: a mikrovezérlő működését meghatározó SFR-ekből (*Special Function Registers* – speciális funkciójú regiszterek) és a programozó által felhasználható GPR regiszterekből (*General Purpose Registers* – általános célú regiszterek). A szimulátor File Register, illetve SFR ablakában nézhetjük meg a regiszterek tartalmát.

Digital-to-Analog Egy digitális érték átalakítása analóg értékké (általában feszültséggé).

Direct Addressing Mikor egy adatmemória-cím szerepel az utasításban. Ilyenkor az utasítás operandusa az így megadott címen lévő tartalom.

Directives Direktívák, az assembler programnak szóló utasítások. Arra utal, hogyan kezeljük a mnemonikokat, hogyan adjuk meg az adatokat, hogyan vezéreljük az assembler működését. Segítségével rugalmasan irányíthatjuk az assembleret.

Download Letöltés. Az a folyamat, amikor a PC felől adatokat, illetve programot töltünk egy célramkörbe, programozóba, emulátorba, hibavadász (debugger) eszközbe.

EEPROM Electrically Erasable Programmable Read Only Memory. Elektromosan törölhető, csak olvasható memória. A csak programozható memória speciális típusa, ami elektromosan törölhető. A bájtokat egyenként írhatjuk vagy törölhetjük. Az EEPROM tartalmát megőrzi a tápfeszültség kikapcsolása után is.

Embedded System Beágyazott rendszer. Egy mikroszámitógép, amit egy eszközbe építene. Ez vezérli, figyeli vagy az eszköz működését irányítja. A legegyszerűbb beágyazott rendszer egy mikrovezérlő áramkör. Működtető programja a firmware, általában cserélhető és módosítható. Ilyen például a PC ROM BIOS-a.

Emulation Az a folyamat, amikor a végrehajtandó firmverprogramot az emulációs memoriába töltjük, és az ellenőrzendő programot hardver- és szoftverfelügyelet mellett futtatjuk.

Emulation Memory Az emulátoreszközben lévő programmemória.

Emulator Az emulációt végző hardvereszköz.

Emulator System Ilyen például az MPLAB® ICE emulátorrendszer, ami tartalmaz egy podhak hívott, a vizsgált rendszerre csatlakozó elektromos illesztőegységet, egy processzormodult, egy eszközillesztőt, kábeleket és az MPLAB programot.

EPROM Electrically Programmable Read Only Memory. Elektromosan programozható memória, ultraibolyai fénnnyel törölhető.

Event Esemény. A buszciklus során cím-, adat- egyéb információk: a ciklus típusa (programlehívás (*fetch*) írás, olvasás, időbelyeg megadása. Az indítás (*trigger*) és a töréspontok megadására használt.

Executable Code Lásd „Hex Code”.

Export Az MPLAB® IDE környezetből az adatok kivitele szabványos formában.

Expressions Kifejezések. Szimbólumokkal megadott állandókkal aritmétikai-logikai műveleteket lehet végezni az assembler forrásmezőjében.

External RAM Tokon kívül elhelyezett írható/olvasható memória.

EXTRC External Resistor-Capacitor (RC). Külső ellenálláskapacitás-áramkör, rendszerrajel előállítására. Néhány típusnál ezt RC módnak hívjuk.

Flash Olyan EEPROM típus, ahol az adatok írása vagy törlése nem egyenként (bájtonként), hanem bájtcsoportonként (blokkonként) történik.

FLASH Memory Ezt a memóriát programozhatjuk és törölhetjük beforrasztás után. Ez általában EEPROM memória. **Enhanced Flash**: 40 év adatmegőrzési idő, önprogramzási képesség, működési feszültsége 2 V–5,5 V között, ICSP (*In-Circuit Serial Programming* – soros programozás beforrasztott állapotban) 5 V vagy 12 V feszültséggel; az adat EEPROM élettartama max. egymillió törlési/írási ciklus. **Standard Flash**: max. 10 000 törlési/írási ciklus, 40 év adatmegőrzési idő, ICSP-képesség csak 12 V-on.

FOSC A tok működtető oszcillátorának a frekvenciája.

File Registers A PIC mikrovezérlőkben lévő GPR és SFR regiszterek összessége.

GIO General Input/Output – általános bemenet/kimenet.

GPIO General Purpose Input/Output – általános célú bemenet/kimenet.

GPR General Purpose Register (RAM). Az adatmemoriának az a része, ahol a felhasználói program változóit tároljuk.

Harvard Architecture Olyan számítógép-felépítés (architektúra), ahol a programmemória-busz és az adatmemória-busz szét van választva. Ez lehetővé teszi a program- és adatmemória egyidejű elérését, ami növeli a rendszer teljesítményét. Az összes PIC mikrovezérlő ilyen felépítésű.

Hex Code A gépi kódú programok szabványos ábrázolására szolgál. Egy hex fájlban található.

Hex File Szövegfájl, amely hexadecimális címeket és értékeket tartalmaz karakteres formában. Ezt tudja fogadni egy programozó vagy hibavadász eszköz.

High Level Language Magas szintű nyelv. Összetett struktúrákat alkalmazó programnyelv (pl. C). Az ezen a nyelven megírt programot egy fordító (*compiler*) segítségével alakítjuk át a processzor által végrehajtandó utasítások sorozatára.

Holding Capacitor Tartó kapacitás. Az Analog-to-Digital (A/D) modulban lévő kondenzátor, amely megőrzi (tartja) a bemeneten lévő, mérendő feszültségrőlétet a konverzió ideje alatt. A konverzió kezdete előtt kivezetései a külső analóg lábra (és a földre) kapcsolódnak, a konverzió kezdetekor innen leválnak, és a feltöltött értéket tartja az A/D konverzió ideje alatt.

HS (High Speed) Nagy sebességű. A rendszerrajel egyik típusa. A kapcsolódó oszcillátor-áramkört ennek megfelelően állítják be. Sebesség 4 MHz–40 MHz között (jelenleg).

I²C™ Inter-Integrated Circuit. Kétvezetékes kommunikáció.

ICD In-Circuit Debugger. Az MPLAB® ICD a Microchip hibavadász eszköze, jelenleg az ICD3 a legújabb.

ICE In-Circuit Emulator. Az MPLAB® ICE a Microchip emulátora, ami képes együttműködni az MPLAB® IDE-vel.

IDE Integrated Development Environment. Egybeépített fejlesztői környezet. Programfejlesztéshez használt, több funkciót egyesítő programrendszer. Az MPLAB® IDE tartalmaz fordítót, assemblert, projektkezelőt, szövegszerkesztőt, hibavadászt, szi-

mulátort és egyéb programokat. A felhasználó úgy fejleszthet programot (írhat kódot, fordíthatja és ellenőrizheti), hogy nem lép ki az IDE asztalról.

Identifier Azonosító. Egy változó vagy szubrutin neve.

Import Egy külső (pl. hex) fájlból olvasunk be adatokat az MPLAB® IDE környezetbe.

Indirect Addressing Indirekt címzés. Az utasításban egy olyan adatmemória-cím, ami az utasításdekódolónak azt jelzi, hogy egy adott másik című regiszter tartalmát tekintse címmek, és az onnan kiolvasott tartommal végezze el a műveletet. Ez 8/12, 8/14 felépítésű PIC mikrovezérlőknél a nullás cím (INDF). Ha ez szerepel az utasítás címmezejében, akkor a 4-es című FSR regiszter tartalma által megcímezett regiszter tartalmával végezzük el a műveletet.

Initialized Data Olyan adatok, amelyeket azonnal egy kezdeti érték megadásával definiálunk. Pl. C-ben az int myVar=5; definíció egy olyan integer adatot definiál, aminek a kezdőértéke 5.

Instruction Bus Utasításbusz: a programmemóriából kiolvasott utasítások ezen a buszon közlekednek.

Instruction Cycle Utasításciklus. Az utasítás végrehajtásakor végrehajtott események sorozata. Általában négy esemény: dekódolás, olvasás, végrehajtás és az írás alakot egy utasítást. A végrehajtása során nem minden szerepel minden a négy. Négy órajel (TOSC) kell egy utasítás végrehajtásához. Ez egy utasításciklus (TCY).

Instruction Fetch Utasításlehívás. A Harvard felépítés miatt, még egy utasítást végrehajtunk, a következő programmemória-helyen lévő utasítás már lehívásra kerül.

Interrupt Megszakítás. A CPU-hoz érkező jel, amelynek hatására a normál programvégrehajtás megszakad, a következő utasítás címe a verem tetejére töltődik, és a program 04h (*Interrupt Vector Address*) címen folytatódik. A megszakítást kiszolgáló rutin végén lévő RETI utasítás hatására a verem tetjén lévő elmentett cím visszaíródik az utasításszámlálóba, és a megszakadt program tovább folytatódik.

INTRC Belső RC oszcillátor. A tokba integrált belső oszcillátor-áramkör.

KEELOQ® KEELOQ® egy szabadalmaztatott kódugrásos technológia. A Microchip KEELOQ® termékei egyirányú adatátviteli titkosítást és visszaállítást biztosítanak, ami minden adatátvitel esetén más kódosorozatot generál.

LCD Liquid Crystal Display. Az LCD kijelzéssel az információk megjeleníthetők.

LED Light Emitting Diode. Világító dióda. Ilyen kijelzéssel az információk megjeleníthetők.

Librarian Könyvtárkezelő. Programnyelvi eszköz, amivel könyvtárat kezelhetünk és hozhatunk létre. Az MPLIB™ a Microchip könyvtárkezelőjének a neve.

Library Könyvtár. Áthelyezhető tárgymodulok gyűjteménye. Több forrásfájl assemblálása után az elkészült tárgykódokat a könyvtárkezelő segítségével egy könyvtárfájllá alakítjuk. A könyvtárban szereplő tárgymodulokat a programunkból készített tárgykódokkal összeszerkeszthetjük egy futtatható programmá.

LIN Protocol Specification LIN protokoll specifikációja. Olcsó, kis távolságot áthaladó kis sebességű hálózat, ami kapcsolók állapotváltozásait és a rájuk adott válaszokat viszi át. Az autóiparban használt.

Link Szerkesztés.

Linker Egy nyelvi eszköz, amivel az általunk készített tárgykódot összeilleszti a könyvtármódulokkal, létrehozva a futtatható kódot. A szerkesztést a Microchip MPLINK™ LINKER programja segítségével végezzük.

Linker Script Files Ezek az MPLINK™ LINKER *.LKR kiterjesztésű parancsfájlok, benne adjuk meg a szerkesztési opciókat, leírja a használható memóriát.

Literal Konstans, más néven állandó, az utasításba van ágyazva.

Listing File Szövegfájl, ami a C forrás- vagy assembler programból létrehozott gépi kódot együtt mutatja a forrásprogramban szereplő szimbolikus utasításokkal.

Logic Probes Maximum 14 logikai jelet kezelő érintkezőt lehet az emulátorhoz kapcsolni. Ezek biztosítják a külső, nyomon követendő jelekhez való bemeneti csatlakozást, az indító (*trigger*) kimenő jeleket és a tápfeszültségeket.

Long Word Instruction Olyan utasítástípus, ami együtt tartalmazza a műveleti kódot és az operandust. Ez teszi lehetővé, hogy minden utasítás egy utasításciklus alatt végrehajtható.

LP Low Power. Egy oszcillátor típus. Alacsony fogyasztású, lassú oszcillátor, max. 200 kHz-ig.

LSb (or LSB) Least Significant Bit. Legkisebb helyi értékű bit. Pl. B'01100010' esetén 0.

Machine cycle Gépi ciklus. PIC mikrovezérlőknél a rendszerrajel négyszerese (4TOSC). TCY rövidítéssel jelölik.

Machine Code Amit a processzor végrehajt.

Macro Névvel és paraméterrel adott assembler utasítások együttese, amire ha az assembler programban hivatkozunk, akkor az aktuális paramétereivel a programba másolódik. A makrókat az előtt kell definálni, mielőtt használnánk; a fordított irány – hivatkozás, utána definíció – nem megengedett.

Master Synchronous Serial Port Az MSSP-nek két működési módja van: az első a Serial Peripheral Interface (SPI™), a második az Inter-Integrated Circuit (I²C).

MCU Microcontroller Unit, más néven a mikrovezérlő.

Memory Models Az objektumkönyvtárak és/vagy előre lefordított objektumfájlok, amelyek az eszköz program- és adatmemoriájának méretétől függnek.

Microcontroller Mikrovezérlő. Egy nagy integráltságú áramkör, ami egy mikroszámítógép működéséhez szükséges minden részegységet tartalmazza. Ez a CPU, program- és adatmemória, időzítő elemek, valamint a perifériák.

MIPS Million Instruction Per Second. Millió utasítás másodpercenként. Az utasítások végrehajtási sebességének egyik mértéke.

Mnemonics Utasítások könnyen megjegyezhető, általában az angol nyelv igékből származtatott neve. Pl. MOV (*move* – mozgatás). Egy utasítást formailag egy név (mnemonik) és az operandusok szimbolikus nevét tartalmazza. Szokták műveleti kódnak is nevezni.

MPASM™ Assembler A Microchip makró assembler. Az MPASM™ lehet DOS vagy Windows alapú PC-n futó program, amivel assembly nyelven írt programot fejleszthetünk. A makró kifejezés azt jelzi, hogy benne makrókat is elhelyezhetünk.

MPLAB® ICD In-Circuit Debugger. A Microchip beépített hibavadász eszköze, gyakorlatilag a 8/14-es PIC16F87X eszközöktől kezdve használható a 8/16, a 16/24 és 32/32 PIC mikrovezérlőknél. Az MPLAB® ICD az MPLAB® IDE környezetbe van ágyazva. Az MPLAB® ICD rendszer áll egy modulból, a mikrovezérlőhöz csatlakozó illesztő fejből (header), a demókártyából (esetleg), kábelekből és az MPLAB® programból.

MPLAB® ICE A Microchip in-circuit emulátora, ami az MPLAB® IDE rendszerbe ágyazva működik.

MPLAB® IDE A fejlesztői környezet fő programjának a neve, irányítja az Editor, Project Manager és az Emulator/Simulator Debugger modulok együttműködését. A teljes MPLAB® program a PC-n fut. Az indítandó program neve: MPLAB.EXE, és ez hívja meg a többi programot.

MPLAB® SIM Az MPLAB® IDE részeként működő, Microchip által készített szimulátor.

MPLIB™ Librarian Az MPLIB™ könyvtárkezelő a COFF tárgymodulokat (*.o), amelyeket vagy az MPASM™ v2.0, MPASMWIN v2.0, vagy MPLAB® C v2.0, vagy későbbi programokkal hoztak létre, az MPLAB™ könyvtárkezelő egyetlen könyvtárfájllá alakítja. Ezenkívül a programot az így létrehozott könyvtárlállományok kezelésére is használhatjuk.

MPLINK™ LINKER MPLINK™ LINKER az MPASM™ assemblerének a szerkesztője. A Microchip C fordítónál az MPLINK™ LINKER szintén használható az MPLAB™

Librarian könyvtárkezelővel. Az MPLINK™ LINKER-t úgy tervezték, hogy az MPLAB® IDE környezetben működjön, de nélküle is használható. Az MPLINK™ LINKER tárgykódokat és könyvtárat egyetlen végrehajtható fájllá egyesít. MPSIM™ Simulator Egy DOS alá készített Microchip szimulátor. Az újabb szimulátor neve: MPLAB® SIM.

MSb Legnagyobb helyi értékű bit. Pl. '01110101' esetén 0.

MSB Most Significant Byte. A legnagyobb értékű bájt. Pl. 0XF5D3 esetén: F.

MSSP Lásd a „Master Synchronous Serial Port” bejegyzésnél.

Non-Return to Zero (NRZ) Adatátvitelnél használt két szintet használó kódolás. Az 1-es bitérték jelöli a magasabb feszültségű jelet, míg a 0-s bitérték jelöli az alacsonyabb jelszintet.

Object Code Tárgykód. Egy forráskóból származtatott ideiglenes kód, amit az assembler vagy a fordító hoz létre. Ezt a kódot használja fel a szerkesztőprogram (*linker*), létérehozva a futatható kódot. A tárgykód az objektumfájlból található.

Opcode Műveleti kód. Az utasításnak az a része, ami az utasítás által elvégzendő műveletet határozza meg. Ennek a résznek a hossza az utasítás típusától függ. Az utasítás fennmaradó része tartalmazza a művelet operandusát, ami lehet egy állandó (literál), egy adatmemória-cím, aminek a tartalmával végezzük el a kijelölt műveletet, vagy egy programmemória-cím, amit a PC-be töltve meghatározhatjuk a következőnek végrehajtandó utasítás címét (pl. GOTO).

Oscillator Start-up Timer (OST) Ez az időzítő 1024 órajel-periódus eltelte után engedélyezi a RESET állapotból történő kilépést.

OST Lásd az „Oscillator Start-up Timer (OST)” bejegyzésnél.

OTP One-Time-Programmable. Egyszer programozható.

Pages Lapok. A programmemória egy címzési módszere. Mivel az utasításban szereplő programmemória címének hossza rövidebb a fizikailag címezhető memória címének hosszánál, ezért lapozó bitekkel egészítjük ki az utasításban szereplő cím részt. Vagyis a lapozó bitek határozzák meg, hogy az utasításban szereplő cím melyik fizikai memóriaterületen – lapon – található. Pl. a 8/14-es mikrovezérlőknél az utasításban csak 11 bites programmemória-cím szerepelhet. Mivel ezt kiegészítették két lapozó bittel, így négy memórialapból áll a fizikai memória.

Parallel Slave Port (PSP) Párhuzamos kommunikációs port, amivel egy periféria (pl. karakteres LCD-kijelző) vagy egy másik mikrovezérlő nyolcbites adatbuszához csatlakozhatunk.

PC Personal Computer (személyi számítógép) vagy Program Counter (programszámláló).

PC Host Személyi számítógép.

PICmicro® MCUs Microchip mikrovezérlő-családok összefoglaló neve.

PICSTART® Plus Device Programmer A Microchip egyik programozó eszköze. Képes 8, 14, 28, és 40 lábú PIC mikrovezérlők programozására. Az MPLAB® programból tudjuk kezelni.

POP Kifejezés (utasítás), amivel egy, a veremben eltárolt információ kiolvasható.

Postscaler Utóosztó. A bemenetére érkező jel (gyakoriságát) frekvenciáját osztja le. A megszakítások gyakoriságát vagy a WDT túlcsordulási idejét csökkenthetjük.

Power-on Reset (POR) Tápfeszültség növekedését érzékelő áramkör. Ha a 0 V-ról induló feszültség elér egy értéket, akkor eszköz RESET jön létre, és a PWRT számláló elkezdi a számlálást.

Power-up Timer (PWRT) Egy számláló, ami a belső RESET jelet alacsony szinten tartja a számláló túlcsordulásáig, ezzel lehetővé téve, hogy a RESET fennmaradjon, amíg a mikrovezérlő el nem ér egy érvényes működési feszültséget. Ha bármelyik krisztályrezonátor oszcillátort formáló választjuk, akkor PWRT túlcsordulásakor az OST időzítő kerül engedélyezésre.

Prescaler Előosztó. Számlálóból felépített áramkör, ami a bemenetén megjelenő jel frekvenciáját leosztja az osztásviszonynak megfelelően.

Program Bus A programmemoriából az utasítások kiolvasását lehetővé tévő vezetérendszer, ami az utasításdekódoló regiszterhez csatlakozik.

Program Counter Regiszter, ami meghatározza a programmemoriából kiolvasandó következő utasítás címét.

Program Memory A microkontroller memóriájának az utasításokat tároló része.

Programmer Olyan berendezés, amivel az elektromosan programozható áramkörök, pl. a mikrovezérlőket programozzák.

Project Fájlok csoporthja, amely egy alkalmazói gépi kódú program elkészítéséhez szükséges. A fájlok kezelését, nyilvántartását a projektkezelő (Project Manager) végzi.

PRO MATE® II Device Programmer A Microchip egyik PIC programozója. Képes az összes PICmicro® mikrovezérlő és a KEELOOQ® tokokat programozni. Önállóan vagy az MPLAB IDE részeként használható.

Prototype System A felhasználó alkalmazás másik megnevezése.

PWM Signals Lásd a „Pulse Width Modulation (PWM)” bejegyzésnél.

PSP Lásd a „Parallel Slave Port (PSP)” bejegyzésnél.

Pulse Width Modulation (PWM) Állandó frekvenciájú jelfolyam, ahol az információt a magas szintű impulzusszakasz hossza hordozza. A legegyszerűbb digitális-analóg átalakító.

PUSH Információk a verem tetejére történő írása.

PWM Lásd a „Pulse Width Modulation (PWM)” bejegyzésnél.

Q-cycles Ugyanaz, mint a rendszer oszcillátor ciklusideje. minden utasításciklus 4 Q-ciklusból áll.

RAM Random Access Memory. Véletlen elérésű írható/olvasható adatmemória.

Raw Data Nyers adat. Egy program vagy adatszekcióban lévő adatok bináris formája.

RC Resistor-Capacitor Ellenállás-kapacitás. A rendszeroszcillátor alapértelmezés szerinti oszcillátora.

Read-Modify-Write Utasításvégrehajtási forma, amikor egy regisztert kiválasztunk, módosítuk a tartalmát, és utána visszaírjuk az eredeti helyére.

Register File Az adatmemória Microchip szerinti elnevezése. SFR és GPR elnevezésű részeiből áll.

Real-Time Valós idejű. Mikor egy emulátor vagy hibavadász eszköz parancsunkra az álló halt állapotot elhagyja, a processzor teljes sebességgel fut, és pontosan úgy működik, mint normál futáskor. Ilyenkor – ha engedélyezett – a futási adatokat gyűjtő tárolóban (*real-time trace buffer*) gyűlnek a futás közbeni adatok. Ez befejeződik, ha érvényes töréspontra fut a program, vagy ha kézzel leállítjuk a program futását.

ROM Read Only Memory. Memória, aminek a tartalma nem módosítható, csak kiolvasható.

Sampling Time Mintavételi idő. Az időtartam, amíg a konverzió eredményét megkapjuk. Két részből áll: a gyűjtési időből (*acquisition time*) és a konverzióhoz szükséges időtől (*conversion time*).

Serial Peripheral Interface (SPI™) Az SSP vagy MSSP modulok egyik működési módja. Hívják háromvezetékes interfésznek is (adat ki, adat be, órajel). Mivel órajel van, az átvitel szinkron. Lényegében a shift (toló) regiszterek tartalma cserélődik ki.

Section Szekció. Névvel, nagysággal, és címmel jellemzhető program- vagy adatterület.

SFR Special Function Register. Regiszter, amit a mikrovezérlő használ a működése során, vagy egy periféria állapot- (státus-) vagy vezérlőbitjeit tartalmazza.

Shared Section Megosztott szekció. Az adatmemória bankváltás nélkül elérhető közös része.

Simulator Program, ami modellez a PICmicro® mikrovezérlő működését.

Single Cycle Instruction Utasítás, aminek végrehajtása egy gépi ciklust (TCY) igényel.

SLEEP A mikrovezérlő alacsony fogyasztású állapotba, a rendszerórájel is leáll. Ez nagymértékben csökkenti az eszköz fogyasztását. Bizonyos perifériák folytathatják a működésüket.

Source Code – Assembly Forráskód. A PICmicro® utasításait és az MPASM™ Assembler direktíváit és makróit tartalmazza, és ezt az assembler alakítja át gépi kóddá.

Source Code – C Forráskód. Egy C-ben írt program, amit a fordító (*compiler*) alakít át végrehajtható gépi kóddá.

Source File – Assembly ASCII szövegfájl, a PICmicro® utasításait és az MPASM™ Assembler direktíváit és makróit tartalmazza (forráskód), és ezt az assembler alakítja át gépi kóddá. Bármilyen szövegszerkesztővel elkészíthető.

Source File – C ASCII szövegfájl, amely a C nyelven megírt forráskódot tartalmazza, és amit a fordító gépi kódra fordít. Bármilyen szövegszerkesztővel elkészíthető.

Special Function Registers (SFR) Olyan regiszterek, amelyek a mikrovezérlő egységeinek és perifériáinak a vezérlő- és státusbitjeit tartalmazzák.

SPI™ Lásd a „Serial Peripheral Interface (SPI™)” definíciójánál.

SSP Synchronous Serial Port. A szinkro soros portnak (SSP) két működési módja van: Az egyik a Serial Peripheral Interface (SPI™), a másik az Inter-Integrated Circuit (I²C™). A PIC 16/24-es családnál már két önálló, külön egység.

Stack – Hardware A PIC mikrovezérlő egy különálló RAM memoriája, amiben függvények operandusait, változókat és visszatérési címeket tartalmaz. Szubrutinhíváskor a CALL utáni utasítás címe a veremmutató által megcímezett helyre kerül, majd a szubrutin végrehajtása végén lévő RETURN utasítás hatására a PC-be visszatölök, és folytatódik a programvégrehajtás a CALL utasítást követő utasítással. A megszakítás kiszolgálása is ehhez hasonló.

Stack Software Programozáskor ez az írható/olvasható memória használható változók, visszatérési címek tárolására. A változók a program függvényeinek egymás közötti paraméterátadására is használható.

Static RAM or SRAM Static Random Access Memory. Olyan memóriatípus, ami írható és olvasható, és tartalmának megőrzése nem igényel periodikus frissítést.

TAD A/D átalakítóknál az átalakítás során egy bit meghatározásához szükséges idő.

Target A fejlesztett áramköri kártya.

Target Application Fejlesztés alatt álló kártya programja.

Target Board Fejlesztés alatt álló áramköri kártya.

Target Processor Célprocesszor. A mikrovezérlő a céláramköri kártyán, amit emulálunk.

TCY Az az időtartam, ami egy utasítás végrehajtásához szükséges. Ez az idő a tok órajel periódusidejének a négyeszerese (Fosc/4) és négy Q-ciklusra van osztva.

Tosc A tok oszcillátorperiódusának az ideje.

USART Univerzális szinkron-aszinkron vevő-adó. Egy olyan perifériamodul, amely képes egyidejű aszinkron soros adásra és vételre (*full duplex*) vagy váltakozónan egyirányú (*half duplex*) aszinkron portként. Aszinkron módban az USART jelszintillesztő áramkörrel a PC soros vonalára kapcsolható.

Upload Feltöltés a programozóból vagy emulátorból a PC-be, vagy a céláramkörből az emulátorba adatokat töltünk fel.

Voltage Reference (VREF) Egy feszültségszint, amit az A/D átalakításnál referenciának tekintünk. A digitális értékhez tartozó tényleges feszültséget ez határozza meg. Ez lehet a tápfeszültség (AVDD és AVSS) vagy egy külső feszültség.

Von Neumann Architecture Ennél a felépítésnél, a programmemória és az adatmemória egy közös területen van. Ez azt jelenti, hogy a program, illetve az adatmemória elérése csak egymás után valósítható meg, ami a rendszer számítási teljesítményét csökkenti.

Watchdog Timer (WDT) Olyan időzítő, ami egy választható idő múlva reszponzív a mikrovezérlőt. A WDT engedélyezése, illetve tiltása a konfigurációs bitekkel történik.

19.3. EGYÉB IRODALOM

- A könyv írásakor sokszor vettünk igénybe más szerzők által leírt tudásanyagot, ezért ajánljuk minden olvasónak, hogy e könyvön kívül a következő könyveket is bátran forgassa:
- Benkő Tiborné – Popper András – Benkő László: Bevezetés a Borland C++ programozásába*. Computer Books, Budapest, 1993.
- Lamár Krisztián: A világ leggyorsabb mikrovezérlője*. (Az SX mikrokontroller és alkalmazástechnikája). ChipCAD Kft. Kiadványa, 1999.
- Lamár K. – Veszprémi K.: A mikroszámitógépek tényerése a villamos hajtások szabályozásában*. Proceedings of the International Conference Kandó 2002, Budapest, Hungary, p. 7. 2002. ISBN 963 7158 03 0
- Lamár, K.: Microcomputer Control of Brushless DC Motors*. Proceedings of the 20th Joint Scientific Conference Science for Practice, Osijek, Croatia, pp. 59–72. 2003. ISBN 953 6032 46 5
- Lucio Di Jasio: Programming 16-bit Microcontrollers in C*. USA, Elsevier, 2007.
- Lucio Di Jasio: Programming 32-bit Microcontrollers in C*. USA, Elsevier, 2008.
- Martin P. Bates: Programming 8-bit PIC Microcontrollers in C*. USA, Elsevier, 2008.
- Dr. Schuszter György – dr. Simán István: C programozás Borland C++ 3.11 környezetben*. Budapest, BMF-KKVK, 1997.

19.4. A BINÁRIS PREFIXUM

A számítástechnikában a bináris prefixumokat (bináris előtétszókat) használják nagy számok jelölésére az International Electrotechnical Commission (IEC) szabványa alapján. Az SI prefixumokkal ellentétben (melyek a 10 hatványai) a bináris prefixumok a 2 hatványai. 1998 előtt a számítástechnikában is az SI szabvány decimális prefixumait használták a bináris prefixumok jelölésére: k = kilo, M = mega, G = giga stb.

A problémát az okozza, hogy kettő tizedik hatványa éppen ezerhuszonötöngy, azaz alig valamivel több a kilo-val jelzett ezernél. A bináris előtétszó elnevezését úgy kapjuk, hogy az eredeti előtétszó első két betűjéhez illesztjük a bi (binary = bináris) szócskát. A jelét pedig úgy, hogy az első betű naggyal írjuk, és utána írunk egy kis i betűt. Például kilo = ezerszeres, kibi = $2^{10} = 1024$ -szeres. Tehát 1 kB = 1000 B, 1 KiB (ejtsd kibibájt) = 1024 B; valamint 1 Kib = 1024 bit = 128 bajt. A használt bináris prefixumok neveit és értékeit a következő táblázat tartalmazza.

| Szimbólum | Név | Jelentés | Értéke |
|-----------|-------|---------------|---|
| Ki | kibi- | bináris kilo | $2^{10} = 1024^1 = 1\ 024$ |
| Mi | mebi- | bináris mega | $2^{20} = 1024^2 = 1\ 048\ 576$ |
| Gi | gibi- | bináris giga | $2^{30} = 1024^3 = 1\ 073\ 741\ 824$ |
| Ti | tеби- | bináris tera | $2^{40} = 1024^4 = 1\ 099\ 511\ 627\ 776$ |
| Pi | pebi- | bináris peta | $2^{50} = 1024^5 = 1\ 125\ 899\ 906\ 842\ 624$ |
| Ei | exbi- | bináris exa | $2^{60} = 1024^6 = 1\ 152\ 921\ 504\ 606\ 846\ 976$ |
| Zi | zebi- | bináris zetta | $2^{70} = 1024^7 = 1\ 180\ 591\ 620\ 717\ 411\ 303\ 424$ |
| Yi | yobi- | bináris jotta | $2^{80} = 1024^8 = 1\ 208\ 925\ 819\ 614\ 629\ 174\ 706\ 176$ |

Példa:

300 GB = 279,5 GiB (= 0x117,6592E GiB = 0x45D96,4B8 MiB = 0x1176592E KiB = 0x45D964B800 bajt).

Ez a jelölés jelenleg kezd széles körben terjedni.

A bájtot az irodalom általában **B**-vel jelöli. Javasolt a **b** használata a bit jelölésére, de talán a legjobb megoldás a **bit** és a **B** használata, hogy minél könnyebb legyen a kettőt megkülönböztetni, mint például a *kbit* esetén.

Bizonyos mértékegységeket a számítástechnikában is mindig decimálisan használnak. Például, a hertz (Hz), ami számítástechnikai eszközök órajelének mérésére szolgál, vagy a bit/s, ami a sávszélességére. Tehát egy 2 GHz-es processzor 2 000 000 000 elemi műveletet végezhet másodpercenként, egy 128 kbit/s-os MP3 adatfolyam 128 000 bitet (15,625 KiB) másodpercenként, és egy 1 Mbit/s-os internetkapcsolat 1 000 000 bitet (kb. 122 KiB) visz át másodpercenként (8 bites bájtot feltételezve, a kommunikációs vízfej – overhead – nélkül).

Az elektronikus **memória** legtöbb fajtájánál, úgymint a RAM, ROM és a flash memória (a flash-alapú diszkek kivételek lehetnek) általában bináris mértéket adnak meg, mert a gyártástechnológia miatt a kettő hatványai fordulnak elő méretként, hogy az összes címvonal érvényes címhez tartozzon a memóriában.

A **merevlemezgyártók** decimális egységekben jelölik a tárkapacitást. Mivel a legtöbb számítógépes operációs rendszer a lemezhasználatot és -kapacitást binárisan mutatja, a különbség egy látszólagos csökkenést okoz a reklámozott és a formátált, használható méret között. Ez a metódus régi mérnöki tradícióból származik, régebből, mint a vásárlók panaszai a méretkülönbségről, amik az 1990-es évek közepén kezdtek jelentkezni. A merevlemezknél használt decimális méretmegadás egyszerűen csak követi a sorosan elérhető tárolóeszközökönél (például szalagos egység) használt módszert.

A lemezeket azonban szektoronként olvassuk be, nem bájtonként. Mivel a szektorok a RAM-ba olvasódnak be, aminek mérete viszont a kettő hatvanya, így a szektorméret maga is szinte minden kettő hatvanya. Tipikus szektorméretek: 512 bájt (mágneslemez), 2048 bájt (DVD). Emiatt néha egy nagyon zavaró hibrid rendszer használatos, amiben egy „megabájt” ezer darab 1024 bájtos „kilobájtot” jelent.

Tehát a gyártók 1,44 MB-os floppy lemezként árulnak egy olyan terméket, aminek mérete sem $1,44 \times 2^{20}$ bájt, sem $1,44 \times 10^6$ bájt, hanem $1,44 \times 1000 \times 1024$ bájt (körülbelül 1,406 MiB, vagy 1,475 MB).

A CD-lemezek kapacitását minden bináris egységekben adják meg. Egy „700 MB-os” (vagy „80 perces”) CD névleges kapacitása így valójában 700 MiB. Ellenben a DVD-lemezek méretét decimális egységben adják meg. Egy „4,7 GB-os” DVD névleges kapacitása körülbelül 4,38 GiB.

Fontos megjegyeznünk, hogy a könyv írásakor az itt leírtakat figyelembe vettük.

19.5. KÖNYVAJÁNLÓ

Ebben a fejezetben az Elsevier kiadó angol nyelvű szakkönyveiből ajánlunk olvasóink figyelmébe olyan köteteket, amelyek a PIC mikrokontrollerek további alkalmazási lehetőségeit és C nyelvű programozását ismertetik, továbbá FPGA áramköröket, kapcsoló üzemű és rádiófrekvenciás alkalmazásokat mutatnak be.

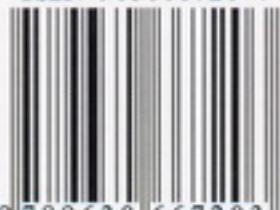
A könyvek megvásárolhatók a ChipCAD Kft. székhelyén.

2000-ben jelentettük meg a **PIC mikrovezérlők alkalmazástechnikája** című könyvet, amelyet 2003-ban átdolgozva adtunk ki másodszor. A második kiadás 2008 elejére elfogyott, így időszerűvé vált, hogy a PIC mikrokontrollerek fejlődésével kapcsolatos változásokat újabb kiadásban foglaljuk össze. Időközben a Microchip a PIC mikrokontrollereivel teljesen új területekre lépett: 2004-ben indította el a 16 bites dsPIC, majd 2008 tavaszán a 32 bites PIC32 mikrokontrollerek gyártását. A harmadik kiadásban dr. Kónya László összefoglalja a hatalmassá bővült PIC mikrovezérlő-palettán bekövetkezett változásokat, és bemutatja az alkalmazásaik kifejlesztéséhez nélkülözhetetlen hardver- és szoftverfejlesztési eszközökét és technikákat.

Az egyre összetettebb PIC mikrokontrollerek programozásához nélkülözhetetlenné vált a magas szintű programozási nyelvek használata. Kopják József a BMF Kandó Kálmán Villamosmérnöki Karon tanít C nyelvű programozást, emellett az elmúlt években a PIC mikrokontrollerek C nyelvű programozását több száz ügyfelünkkel ismertette meg egynapos tervezőtanfolyamainkon. Könyvünk második, a **PIC programozás C nyelven** című részében ezt a területet mutatja be az olvasóknak, akik 78 példaprogramon keresztül ismerhetik meg a PIC24 mikrokontrollerek C nyelvű programozásának alapjait és összetett lehetőségeit. A C programozási fejezetek végén összehasonlító elemzést adunk különböző PIC architektúrák C nyelvű programozásáról és a fordítóprogramjaik – az MPLAB C30, az MPLAB C18, a PIC C és az MPLAB C32 – legfontosabb sajátosságairól.

Új kötetünkben kerültük a korábbi kiadásokban megfogalmazottak átemelését, helyette a PIC architektúrák és perifériakészletek összehasonlító ismertetését tűztük ki célul. A második kiadás a mai napig érvényes és fontos ismereteket tartalmaz, ezért ezt elektronikus formában elhelyeztük a könyv CD-mellékletén, amelyen a könyv mintaprogramjainak és a Microchip szoftvereinek a gyűjteménye is megtalálható.

ISBN 963066720-7



9 789630 667203