



ÓBUDAI EGYETEM
KANDÓ KÁLMÁN
VILLAMOSMÉRNÖKI KAR

Digitális Technika

laboratóriumi gyakorlatok
a 8051-es mikrokontroller családdal

Szerző:

Kiss Attila

Automatika szakirányon,

Programozható Irányítások (PIR) szakterületen

végzett BSc villamosmérnök,

MSc egyetemi hallgató, demonstrátor

Konzulens:

Lamár Krisztián

egyetemi adjunktus

Automatika Intézet

2021. március

Köszönetnyilvánítás

Köszönöm konzulensemnek, **Lamár Krisztián** egyetemi adjunktusnak, hogy a Digitális Technika tantárggyal való első megismerkedésemtől kezdve támogatott, és hogy a Programozható Irányítások (PIR) modul keretében felkínálta számomra szakdolgozat témának egy FPGA-alapú mikrokontroller architektúra megtervezésének és elkészítésének lehetőségét.

Köszönöm neki továbbá a jelen dokumentum gondos átnézését, javítását és kiegészítését.

Az útmutató elkészítésénél felhasználtam az Óbudai Egyetem Automatika Intézetének oktatói (Lamár Krisztián és Zalotay Péter) által korábban készített Digitális Technika mérési útmutató feladatait.

Tartalomjegyzék

A mérési útmutatóról röviden	5
1. Első mérés: Ismerkedés a 8051 programozásával	6
1.1. A szükséges szoftverek telepítése	7
1.2. A fejlesztői környezet konfigurálása	8
1.3. Új projekt létrehozása	11
1.4. Az Assembly program elemei	18
1.4.1. Számok ábrázolása	18
1.4.2. Cím vagy érték?	18
1.4.3. Kommentek	19
1.4.4. Címkék	19
1.4.5. Hogyan írjunk szép kódot?	20
1.5. Assembly Hello World!	22
1.5.1. Fordítsuk le a kódot!	23
1.5.2. Breakpointok elhelyezése	23
1.5.3. Indítsuk el a szimulációt!	25
1.5.4. Lépkedjük a programban!	26
1.6. Adjunk össze: a regiszter ablak használata	28
1.6.1. A gépi ciklusok száma és az eltelt idő kapcsolata	30
1.7. Memóriák elérése	31
1.7.1. DATA memória elérése	31
1.7.2. IDATA memória elérése	34
1.7.3. DATA-IDATA közti kapcsolat	36
1.7.4. Mi van a 7FH címen túl?	38
1.7.5. Te kinél bankolsz? Ismerjük meg a regiszter bankok használatát!	40
1.7.6. A veremkezelés alapjai	42
1.7.7. Az XDATA memória kezelése	47
1.8. Pillantsunk rá a kódmemóriára!	51
1.8.1. A gépi kódokról nagyon dióhéjban	52
1.9. Önálló feladatok:	53
2. Második mérés: 41 20 6B 75 6C 63 73 3A 20 33	54
2.1. A mérés célja	54

2.2.	A párhuzamos portok használata	54
2.3.	Maszkoljuk a biteket!	59
2.4.	A bitcímezhető terület és a bitszintű műveletek	61
2.5.	Logikai függvények megvalósítása mikrokontrollerrel	64
2.6.	Logikai függvények II: olvassunk a kódmemóriából!	66
2.7.	Szubrutinok használata	69
2.7.1.	Már megint a stack overflow...	73
2.8.	A soros port kezelése	78
2.8.1.	Kommunikációs csatornák típusai	78
2.8.2.	Aszinkron soros kommunikációs protokoll: történelem	78
2.8.3.	Az UART interfész	84
2.8.4.	ASCII? Az ki?	96
2.9.	Önálló feladatok	101
3.	Harmadik mérés: Meg-sza-kí-tá-sok	102
3.1.	Miért kell nekünk a megszakítás?	102
3.2.	A 8051-es megszakításai	104
3.3.	A 8051-es megszakítási vektortáblája	110
3.4.	Időzítés megszakításokkal	112
3.4.1.	5ms-os Időalap generálása	116
3.4.2.	Generáljunk más időalapot!	122
3.4.3.	(Majdnem) 5ms-os időzítés 1-es üzemmódban	123
3.5.	Oldjuk meg a második labor lehetetlen feladatát!	127
3.5.1.	Mi történik, ha túl hosszú a megszakítási rutin?	130
3.6.	Konkurens megszakítások kezelése	133
3.7.	Megszakítás megszakítása	133
3.8.	Alacsony fogyasztási üzemmód	135
3.9.	Önálló feladatok:	136
4.	Ami a laborokból kimaradt: Hogyan legyünk igazi fekete öves ASM programozók?	138
4.1.	Ultimate-8051-ASM-template-project	138

A mérési útmutatóról röviden

Kedves Digitális Technika II-t hallgató hallgató! Eme mérési útmutató teljesen új szerzemény, mely a 2020/21 tanév tavaszi félévében került elsőnek a laborokba, hogy a home-office idején ne legyen olyan misztikum a 8051 és az assembly programozás. Elsősorban olvasmányos jellegűre próbáltam írni az útmutatót, hogy emészthetőbb legyen mindenki számára a labor. Igyekeztem a legtöbbször feltett kérdésekre választ adni (főleg azokra, amik nekem is homályosak voltak pár éve) a példák során, de biztosan akad olyan is, amivel nem számoltam.

A laborok során próbáld feladatról-feladatra követni az útmutatót, mert vannak részek, melyek feltételezik, hogy az előzőeket már megcsináltad!

Szívesen fogadom az **építő jellegű kritikát** a mérési útmutatóval kapcsolatban: mit lehetne jobban leírni, milyen ábra/szöveg/kód nem érthető, illetve bármilyen gépelési hibát is javítok a következő évfolyamok számára, ha jelzitek nekem. Ezt a kissattila@stud.uni-obuda.hu e-mail címen tehetitek meg. A levél tárgyába kérlek írjátok bele, hogy **DT2-labor-findings**, mert külön fogom őket gyűjteni ezen kulcsszó alapján!

```
1 BECSI_UT_96_B EQU 2021H
2 CSEG AT BECSI_UT_96_B
3 LJMP MASODIK_FELEV
4 MASODIK_FELEV :
5 LCALL DIGIT_2_LABOR
6 LCALL MATEK_2
7 LCALL ANYAGISMERET
8 LCALL AMI_2
9 LCALL VILLANY_2
10 LCALL MERESTECH
11 LCALL PROG_1
12 SJMP $
13 DIGIT_2_LABOR :
14 LCALL ELSO_MERES
15 LCALL MASODIK_MERES
16 LCALL HARMADIK_MERES
17 RET
18 END
```

1. Első mérés: Ismerkedés a 8051 programozásával

A mérés célja

A Digitális Technika tantárgy előadásai során tárgyalt 8051-es mikrokontroller gyakorlatban való megismerése. A szimulációs lépések alapjainak elsajátítása az [ARM](#) Keil C51 (μ Vision) fejlesztői környezetben, valamint az Assembly nyelvű programozás gyakorlása.

Pro tipp

Mivel nagyon király kis fejlesztői környezet a μ Vision, ezért a mérés után nyugodtan írd meg 1-2 saját programot is (a kötelezőkön túl), vagy csak módosítsd a példaprogramokat, és nézd meg mi hogyan változik.

Az Assembly elsőre nehéz, de ha minden jól alakul, akkor a mérés után rájössz, hogy nem egy ördögtől való nyelv, a 8051-es mikrokontroller felépítése pedig tök egyszerű! Ez az első mérés kicsit hosszú lesz, de próbáld elvégezni becsületesen. Ha ezeket az alapokat ~~végigszenveded~~ végigcsinálod, akkor onnantól gyerekjáték lesz ASM-ben programozni.

Mindenképpen tanulmányozd át az utasításkészletet. Ezt a mérési útmutató végén is megtalálod, ha még nem töltötted volna le külön. Nem kell mindent fejből vágni, de legalább tudd, mit hol keress benne. Az előadásjegyzetet és a tankönyvet is pörgesd nyugodtan a mérés során, ha szükséged lenne rá. Ha nem ijeszt meg az indiai akcentussal beszélt angol, akkor keress videókat YouTube-on! Azért vagy itt, hogy megtanuld az alapokat Assemblyből: **ha nem világos, nem érthető valami, kérdezz!** Hülye kérdés nincs.

Elsőre soknak tűnhet egy ilyen hosszú leírás, de ne aggódj. Csak azért ennyi, mert rengeteg magyarázó szöveg is van a példakódok mellé, hogy egyszerűbb legyen az élet home-office idején. A cél az, hogy profi módon nyomasd az Assemblyt a laborok végére, aztán a vizsgára :)

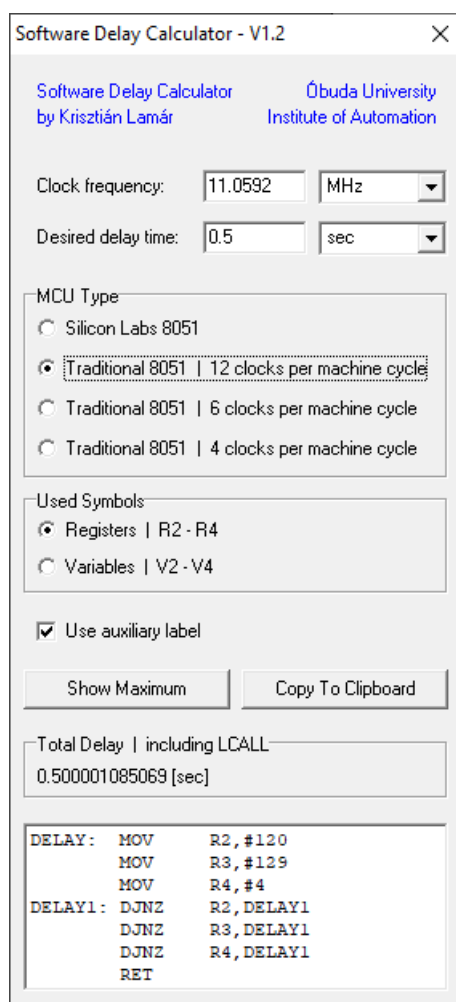
Biztonsági öveket becsatolni, indul az Assembly!

Első feladatként hozzunk létre egy 8051 projektet a μ Vision fejlesztői környezetben! A következő pár lépést minden mérés elején meg kell majd csinálnod! A telepítést természetesen elég csak egyszer.

1.1. A szükséges szoftverek telepítése

A Moodle e-learning rendszerben keresd meg az ARM Keil C51 (μ Vision) fejlesztői környezet telepítőjét, és rakd fel! A telepítés vége felé rákérdez, hogy "Add sample projects to the recently used project list", vagyis hogy hozzáadjon-e néhány minta projektet a projekt listánkhoz. Jelöld be! Sokat lehet belőlük tanulni.

Ezután ugyaninnen szedd le, és telepítsd a Kandó C51 Tools nevű kiegészítést. Ennek segítségével tudod majd – többek között – a késleltető szubrutinokhoz automatikusan legyártatni a kódot. Nulla szenvedés, nulla fejfájás :)



Software Delay Calculator - V1.2

Software Delay Calculator by Krisztián Lamár Óbuda University Institute of Automation

Clock frequency: 11.0592 MHz

Desired delay time: 0.5 sec

MCU Type

- ☐ Silicon Labs 8051
- ☒ Traditional 8051 | 12 clocks per machine cycle
- ☐ Traditional 8051 | 6 clocks per machine cycle
- ☐ Traditional 8051 | 4 clocks per machine cycle

Used Symbols

- ☒ Registers | R2 - R4
- ☐ Variables | V2 - V4

☒ Use auxiliary label

Show Maximum Copy To Clipboard

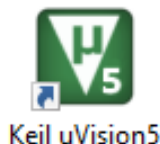
Total Delay | including LCALL
0.500001085069 [sec]

```
DELAY:  MOV    R2, #120
        MOV    R3, #129
        MOV    R4, #4
DELAY1: DJNZ   R2, DELAY1
        DJNZ   R3, DELAY1
        DJNZ   R4, DELAY1
        RET
```

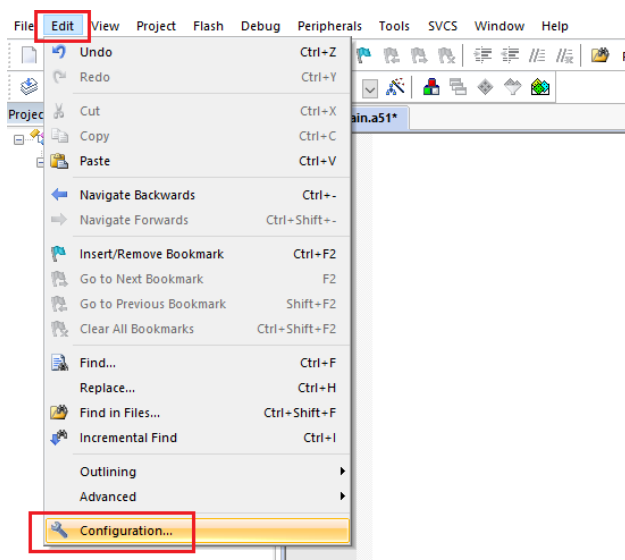
1. ábra. Ez (is) a Kandó C51 Tools tartalma

1.2. A fejlesztői környezet konfigurálása

Indítsd el a fejlesztői környezetet, amit az alábbi ikonnal tehetsz meg.



Ha esetleg nem találod az asztalon, akkor keress rá a Start Menüben: uVision

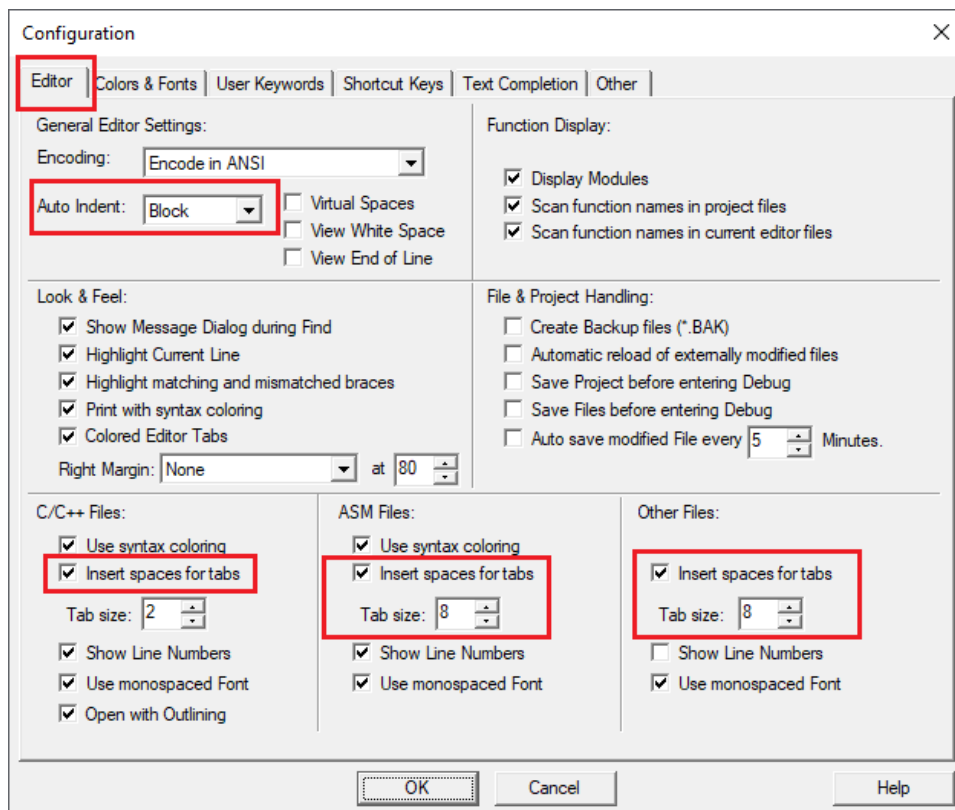


2. ábra. A szövegszerkesztő konfigurálása

Az előadáson biztos volt szó arról, hogyan is kell egy Assembly kódnak kinéznie ahhoz, hogy igényes legyen. **Navigálj a felső menüsorban az Edit → Configuration opcióra!** Itt tudjuk beállítani, hogyan viselkedjen a szövegszerkesztő ablak. A színektől kezdve a tabulátorokon át a margóig mindent a saját szájízünk szerint állíthatunk. Quot capita, tot sententiae¹, de mi most a kádós beállításokat fogjuk használni!

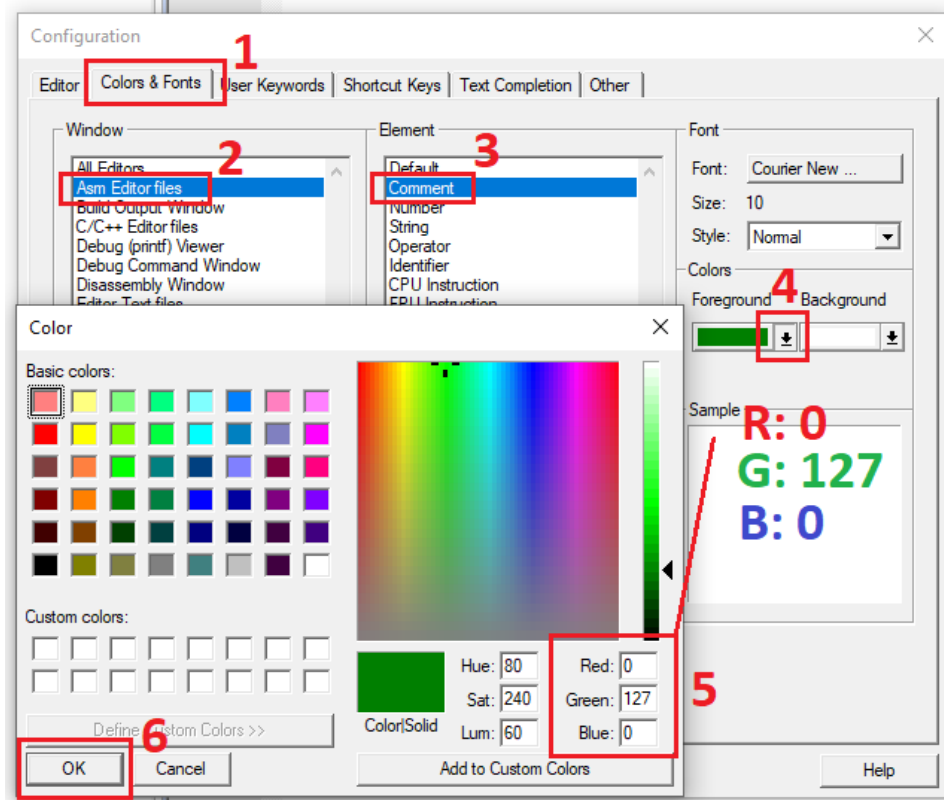
¹Ahány ember, annyi vélemény (latinul)

Ha megnyitottad a konfigurációs ablakot, akkor állítsd át az editor fülön a beállításokat úgy, ahogy a 3. ábrán látod! Ezekkel a beállításokkal nagyon egyszerűen lehet majd formázni a forráskódot, hogy ne csak jó, de szép is legyen!



3. ábra. Tabulátorok beállítása

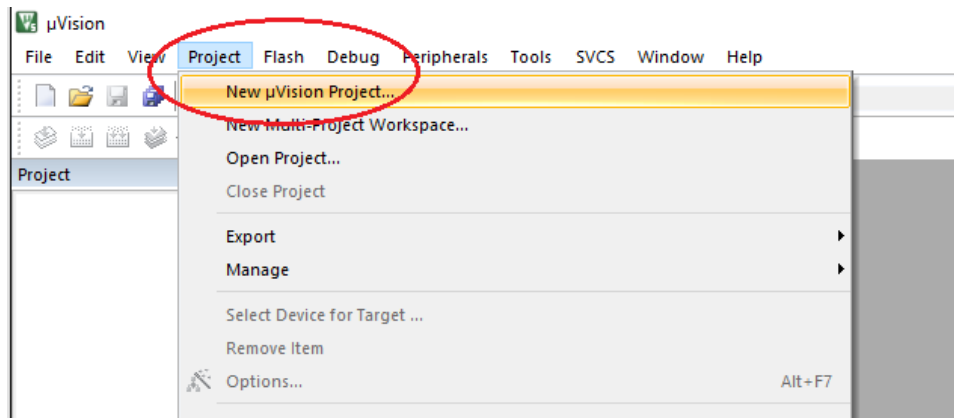
Ha ezzel készen vagy, akkor navigálj át a Colors & Fonts fülre, majd végezd el a beállításokat a 4. ábra szerint! Ezzel a beállítással a kód mellé írt megjegyzések színe jól kivehető lesz.



4. ábra. Kommentek (megjegyzések) színének átállítása

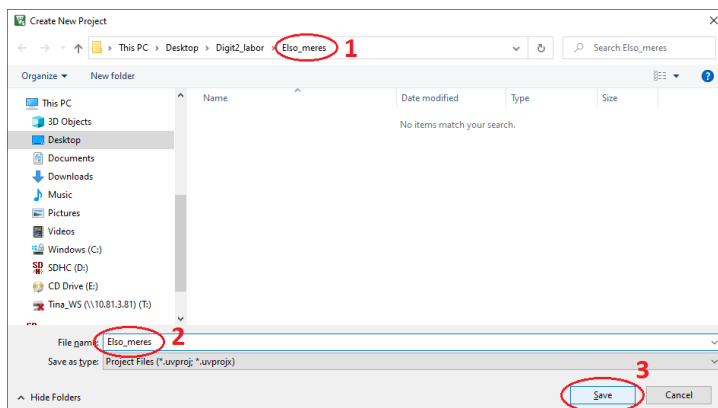
1.3. Új projekt létrehozása

Ha végeztél a konfigurációval, és visszajutottál az üres fejlesztői környezetbe, akkor navigálj a felső menüben a **Project** → **New uVision Project** opcióra, és klikkelj rá!

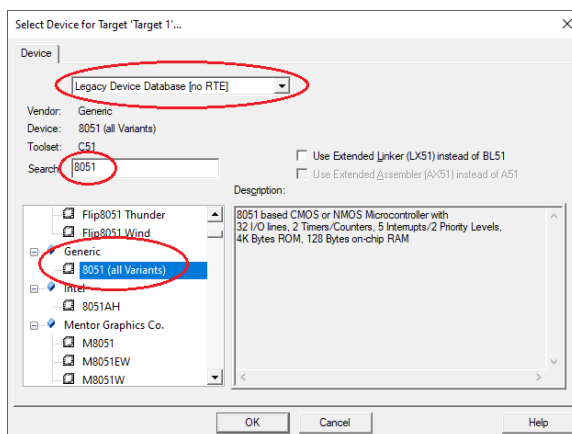


5. ábra. Új projekt létrehozása

Felugrik egy ablak, ahol megadhatod, **hol** és **milyen névvel** szeretnéd létrehozni a projektet. Válassz valami jó mappát ahová dolgozni szeretnél, és **hozz létre egy almappát, amiben szerepel a neved** is pl: GipszJakab_Elso_meres! Ebbe az almappába kattints bele, és itt add meg a nevét a projektnek! Kész? Klikkelj a Save gombra!

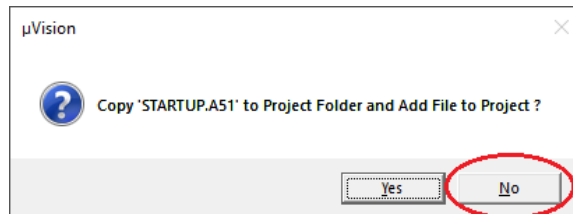


Ha ezzel végeztél, akkor felugrik egy új ablak, ahol ki kell választani milyen mikrokontrollerre szeretnénk programot írni. A software packs helyett válaszd a **Legacy Device Database [no RTE]** opciót, majd **írd be a keresőablakba, hogy 8051!** Ha ez kész, akkor keresd meg a kidobott mikrokontrollerek közül a **Generic 8051** típust. Amint látod, elég sokféle 8051-es létezik, de ez az ami maximálisan megegyezik az előadáson tanult 8051-essel. **Klikkelj az OK-ra!**



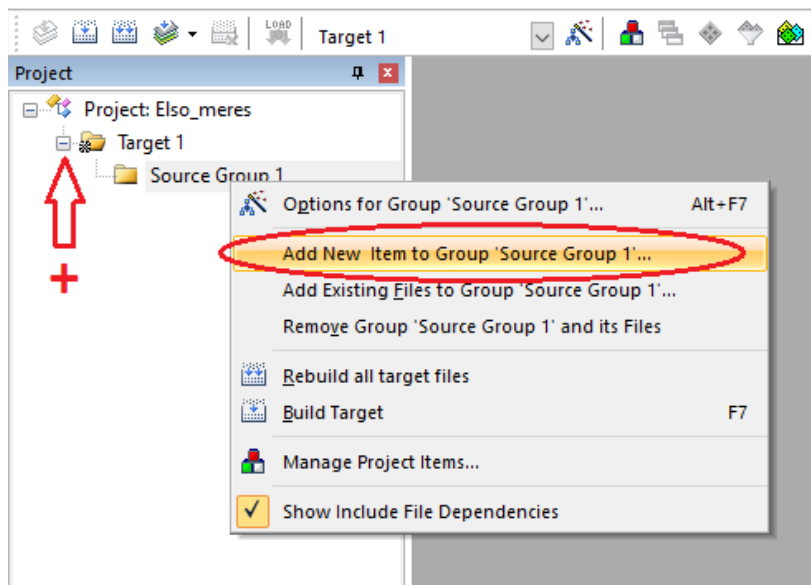
6. ábra. Generic 8051

Ha siker, akkor ismét felugrik egy ablak, ahol lehetőségünk van a projekt-hez hozzáadni egy STARTUP fájlt. **Ezt most nem tesszük meg, mert csak a fekete öveseknek van rá szüksége!** Kattints a NO gombra!



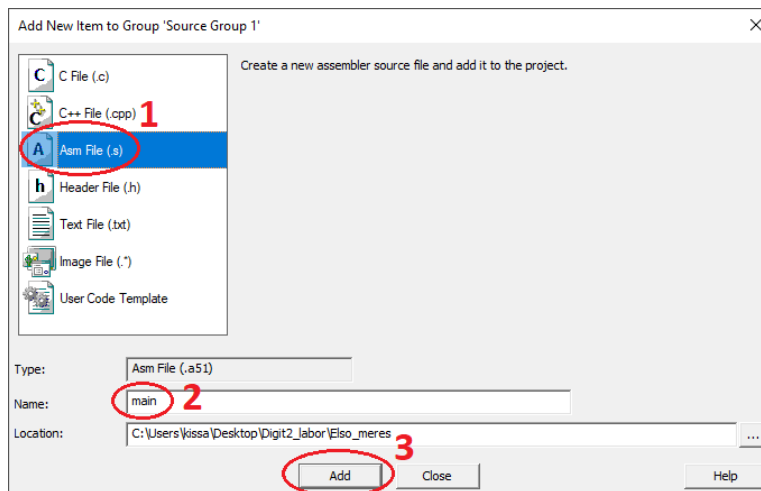
7. ábra. NOPE

Ha kész, akkor egy üres fejlesztői környezetet kell láss. A projekt már létezik, a mikrokontrollert kiválasztottuk, most pedig hozzunk létre egy forrásfájlt, amibe a programot írhatjuk! Navigálj bal szélre, és keresd meg a **Target 1** mappát. Nyisd ki, és **kattints jobb egérrel a Source Group 1** mappára, majd válaszd ki az **Add New Item to Group 'Source Group 1'** opciót!



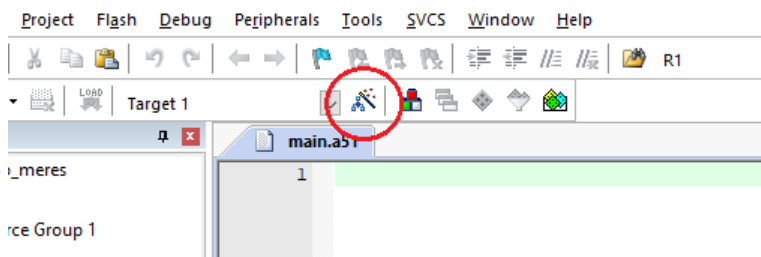
8. ábra. Fájl hozzáadása a projekthez

Ismét felugrik egy ablak, ahol ki kell választanunk milyen fájlt szeretnénk létrehozni. **Válaszd az Asm File (.s) opciót!** Adj neki nevet, majd klikkelj az Add gombra! **Soha ne kezdődjön egy Assembly fájl neve számmal, és ne legyen benne ékezetes betű se!** Érdemes megjegyezni, hogy bár a fejlesztői környezet .s kiterjesztésű fájlt jelöl a menüben, de a tényleges fájloknál emellett még használatban van a **.asm** és a **.a51** kiterjesztés is. Mi a **.a51** kiterjesztést fogjuk használni.



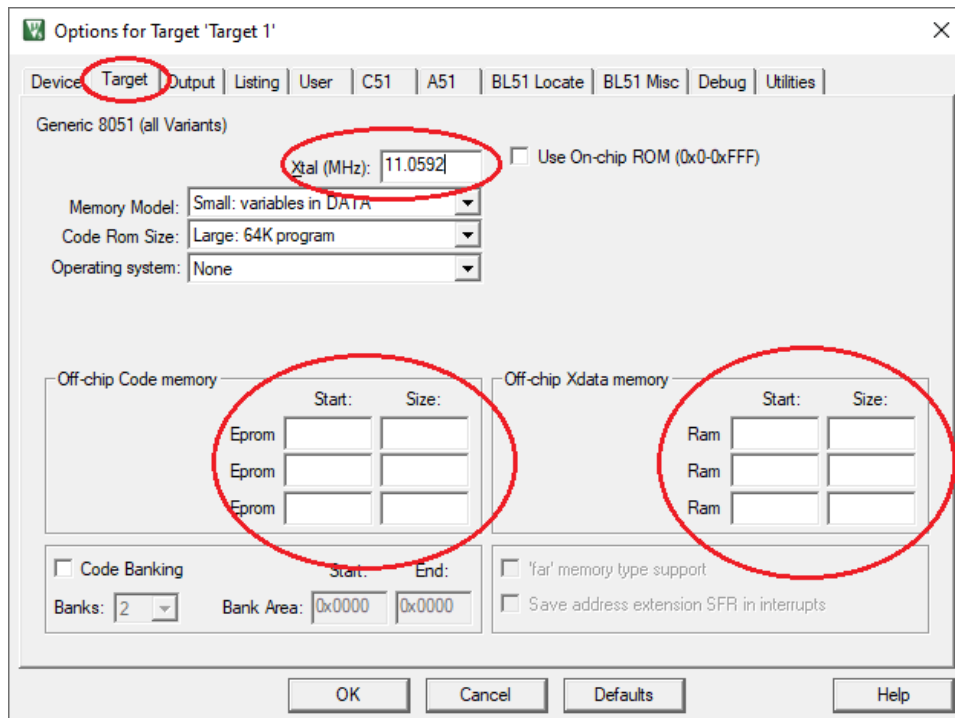
9. ábra. Assembly fájl hozzáadása

Egy teljesen üres szövegszerkesztőt (editort) kell látnod. Már akár most megírhatnánk az első ASM programot, de még el kellene végezni pár beállítást, hogy minden sínen legyen! **Keresd meg a Target 1 (a mikrokontroller) beállításait!**



10. ábra. A mikrokontroller beállításai

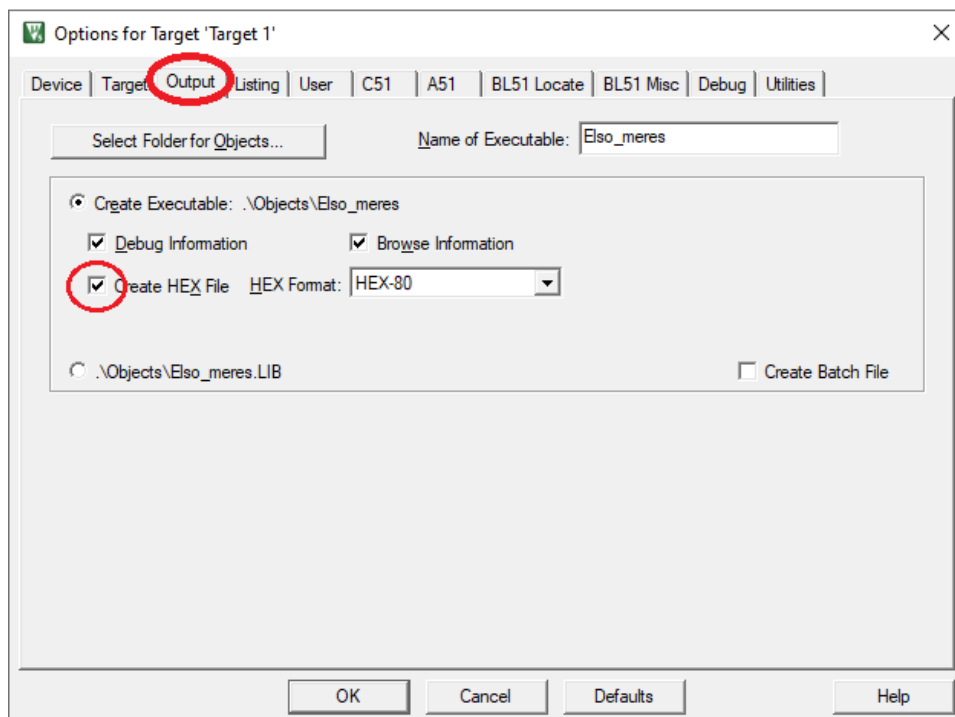
Jön az újabb felugró ablak. Ha minden igaz, akkor a **Target** fülön kell lenni. Ha nem így lenne, akkor kattints oda, és írd át az **Xtal²** értékét **12 MHz-ről 11.0592 MHz-re!** Hogy miért pont ez a megjegyezhetetlen érték az órajel frekvenciája, majd kiderül a második mérésből! **A többi ablakba NE írd semmit!**



11. ábra. Target fül

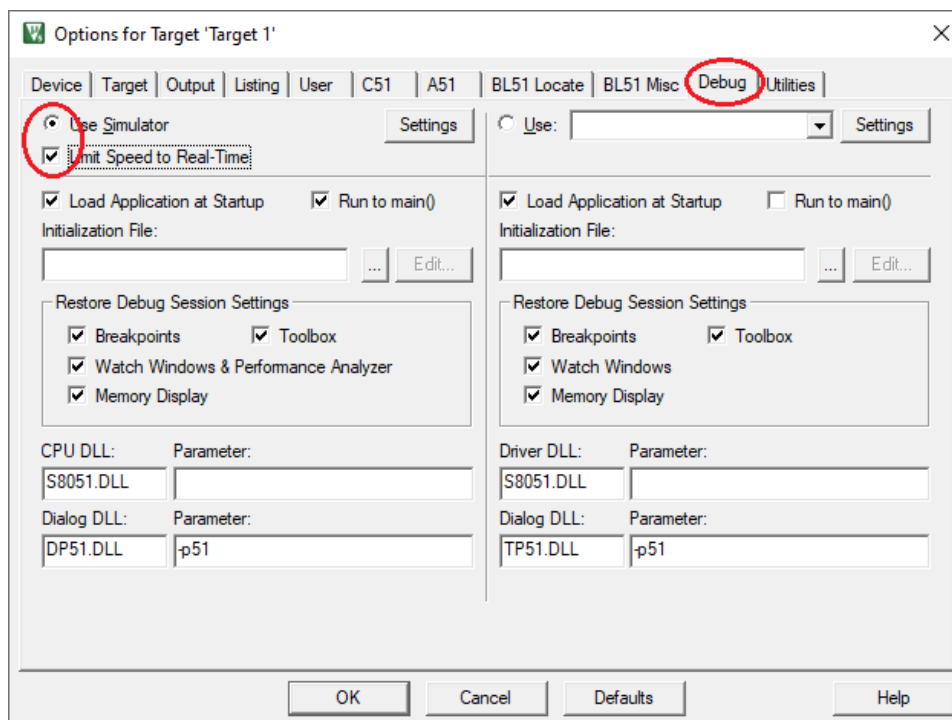
²Xtal → Crystal → Kvarc oszcillátor: a mikrokontroller órajelét szolgáltatja

Az **Output** fülön pipáld be a **Create HEX File** opciót! Ez a fájl tartalmazza a gépi kódot Intel HEX formátumban, ezt érti a mikrokontroller. A szimulációhoz nem kell, de érdemes párszor belenézni, mit is tartalmaz! Meglepően sokat lehet belőle tanulni. Ha viszont van előttünk mikrokontroller, amire rá tudjuk tölteni a programot, akkor ez a fájl elengedhetetlen.



12. ábra. Hex fájl generálás

A μ Vision konfigurálásának utolsó lépéseként navigálj a **Debug** fülre, és válaszd ki a **Szimulátor** opciót, valamint ellenőrizd le, hogy be van-e pipálva a **Limit Speed to Real-Time** rubrika. Erre azért van szükség, mert ha nem pipálnánk be, akkor picit túl gyorsan futna a szimulátor, ha folyamatos futtatást használunk. Ez abból a szempontból persze jó, hogy hamarabb megkapjuk az eredményeket a program végén, de a tanulást nem segíti elő. Azzal, hogy bepipáltuk a valós idejű szimulációt, megoldottuk, hogy a 64 magos, 5GHz-es erőmű is vegyen vissza a számítási teljesítményéből, és csak olyan gyorsan végezze el a 8051 szimulációját, mintha a mikrokontrolleren futna a program 11.0592 MHz-es órajellel.



13. ábra. Limit speed

Kész minden? **Klikkelj az OK gombra!** Visszajutottál a szövegszerkesztőhöz. Mielőtt megírnánk az első programunkat, frissítsük fel a memóriánkat Assemblyből!

1.4. Az Assembly program elemei

Gyorsan foglaljuk össze, milyen elemeket, jelöléseket tartalmazhat egy ASM program az utasításokon kívül.

1.4.1. Számok ábrázolása

Nem mindegy, hogy decimális, bináris vagy esetleg hexadecimális szám-ábrázolást használunk. Amikor egy számot leírunk, akkor jelezni kell az Assembler számára, hogy milyen számrendszerben is gondoljuk mi azt a számot. Ha nem használunk semmilyen jelölést, akkor automatikusan decimálisan értendők a számok. Ha viszont más számrendszerben szeretnénk dolgozni, akkor:

RADIX	Jelölés	Példa 1	Példa 2	Példa 3
2	B	01010011B	11110000B	0b10101010
10	D	255D	80	40
16	H	45H	0DCH	0xFF

Figyelem: ha hexadecimális számot szeretnénk beírni, aminek az első számjegye már betű lenne, akkor egy extra '0' szükséges a szám elé, amennyiben a 'H' betűs írást választjuk!

1.4.2. Cím vagy érték?

A kettő nem ugyanaz, tehát valahogyan meg kell különböztetni őket. Ha értéket szeretnénk írni, akkor azt az Instagramon megszokottak szerint jelölni kell, amit a # szimbólummal tehetünk meg! Hashtag Digit Labor.

Cím	35H
Érték	#35H

```
1 MOV A,35H
2 ;Az A regiszterbe belerakjuk, ami a 35H cimen van.
3 MOV A,#35H
4 ;Az A regiszterbe 35H erteket pakolunk bele!
```

1.4.3. Kommentek

A kommentek segítségével dokumentálni tudjuk a programunkat. Ezeket egy ; (pontosvessző) után írhatjuk.

```
1 NOP
2 ;Ha pontosvesszot teszek, akkor kommenteket irhatok utana.
3 ;Hasznaljuk rendszeresen, mert sokat segit a megertesben!
4 ;Irhatom kulon sorba,
5 NOP ; de irhatom egy utasitas moge is!
```

1.4.4. Címkék

Ha egy adott kódrészletre (pontosabban: a kódmemóriában egy adott címre) vagy egy adatra ember által is értelmezhető módon szeretnénk hivatkozni, akkor címkéket kell használnunk. Sokkal többet mond egy címke, mint egy hexadecimális szám. A címkét leírjuk, majd utána biggyesztünk egy ':' kettőspontot. A példa most CIMKE névvel illetett, de azért a programozás során célszerű értelmes és beszédes neveket adni. Persze az assembler nem érdekli, hogy LOLXD vagy END_OF_PROGRAM a címke neve, de minket annál inkább. Én például a méteres, nagyon deskriptív címke neveket szeretem.

```
1 CIMKE:      ;ez itt egy cimke
2 LJMP CIMKE  ;ugorjunk vissza a cimkere.
3 ;Sokkal erthetobb, mintha pl. LJMP 45FDH lenne, ugye?
```

1.4.5. Hogyan írjunk szép kódot?

Az assembler egy olyan fordító, amely mindent megeszik. Kisbetű vagy NAGYBETŰ, tabulátor vagy space, az ASM-et nemigen érdekli.

```
1 cseg at 0000h ;asddsaasddsa
2 MuTaToM_HoGyAn_Ne_KoDoLj:
3 MoV r6,#69h ;pff, ez így nagyon bona
4 div AB ;kerlek, ne így csinald
5 INC a ;mondjuk attól működik ez így is
6 mov R2,#34 ;de undorító
7 ;random komment, aminek nincs köze a kódhöz
8 dJnZ R6,mUtAtOm_HOGYAN_NE_kodolj
9 end
```

Az utasításokat, címkéket, operandusokat mindig **ALL CAPS** módon írjuk, mert sokkal olvashatóbb és szebb lesz tőle a kód! De ez magában még nem elég, hiszen a behúzásokra (indentation) is oda kellene figyelni.

```
1          CSEG      AT      0000H
2 EZ_MAR_SZEP:                                ;Igy kellene kodolni
3 CIMKE:  MOV      R6 ,#69H                    ;Tabulatorokat hasznalni
4          DIV      AB                        ;A format megtartani
5          INC      A                          ;Minden szepen nez ki
6          MOV      R2 ,#34                    ;Ettol nem sirod el magad,
7          DJNZ     R6 ,EZ_MAR_SZEP           ;ha ranezel
8 ;Kulon komment kezdodhet itt is, ha csak az van a sorban!
9          END
```

Azaz, ha négy oszlopra osztjuk a forráskódot, melyeket egy-egy tabulátor választ el, akkor:

- 1. oszlop: címke
- 2. oszlop: utasítás mnemonikja vagy direktíva helye
- 3. oszlop: operandusok
- 4+ oszlop: kommentek

Ha egy címke után már nem férne ki az utasítás, akkor a címke kerüljön külön sorba. Pontosan úgy, ahogy az *EZ_MAR_SZEP* címkénél látható!

Lesznek olyan példakódok az útmutatóban, melyeknél nem tudtam követni a formázást. Ezt nézd el, kérlek. A papír sajnos nem 24+ hüvelykes felület, mint a monitor. De attól Te még kövesd a szép formázást!

1.5. Assembly Hello World!

Írjuk meg a legegyszerűbb 8051 Assembly programok egyikét! **Gépeld be a programot a fejlesztői környezet editorjába!**

```
1      CSEG      AT      0000H
2      ;Az első utasítás címe: 0000H!
3      LJMP      0030H      ;Ugorjunk el a 0030H címre!
4 ;Az első 48 bajtot atugorjuk,
5 ;mert ide jön majd valami a harmadik merésen
6 ;(30H = 48)
7 CSEG      AT      0030H      ;A következő utasítás címe: 0030H!
8      NOP                      ;Innen indul a program lenyegi része.
9      NOP
10     NOP
11     NOP
12     NOP
13 END_OF_PROGRAM:              ;Lezáró végtelen ciklus
14     LJMP      END_OF_PROGRAM
15     END                      ;Assembly fájl vége
```

Kezdsnek ennyi pont elég, így is van mit emészteni rajta. Ha nem PC-n (papír-ceruzán) írunk Assembly programot, akkor nem elég csak a sima utasításokat bepötyögni a billentyűzeten. Szükség van néhány extra kötelező elemre, hogy minden jól működjön!

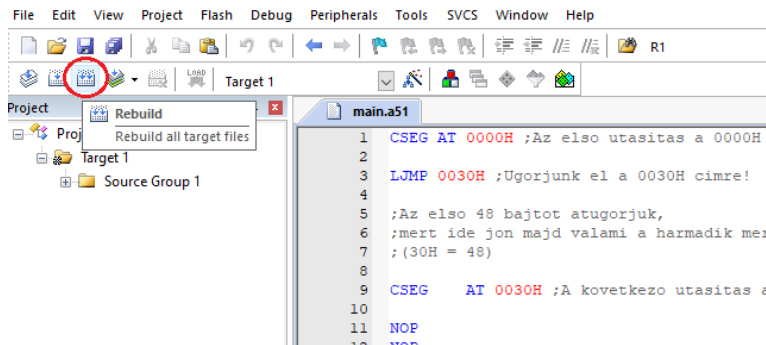
A **CSEG AT** direktíva a kódszegmenst jelenti. Segítségével megadhatjuk, hogy az utána lévő sorban milyen címen legyen az utasítás a kódmemóriában. Az első sorban azt mondtuk, hogy a 0000H címen legyen egy LJMP utasítás, amivel elugrunk a 0030H címre. A CSEG AT 0030H-val pedig megadjuk, hogy a következő utasítás (NOP) az valóban a 0030H címen legyen! Majd később ránézünk, hogyan is néz ki a kódmemória!

Minden ASM program végére **KÖTELEZŐ** a lezáró végtelen ciklus beírása! Ezzel a ciklussal tudjuk megfogni a programot, ha lefutott. Később lesz rá feladat, hogy mi történne ha ez a ciklus nem lenne ott a program végén (nyugi, lesz hozzá segítség is). Az **END** direktívával pedig jelezzük az ASM fájl végét. **Ez is minden program végére kötelező!**

A CSEG AT és az END direktívák, **NEM UTASÍTÁSOK!** Ezek csak az Assembler számára jelentenek bármit is, a mikrokontroller semmit nem tud róluk!

1.5.1. Fordítsuk le a kódot!

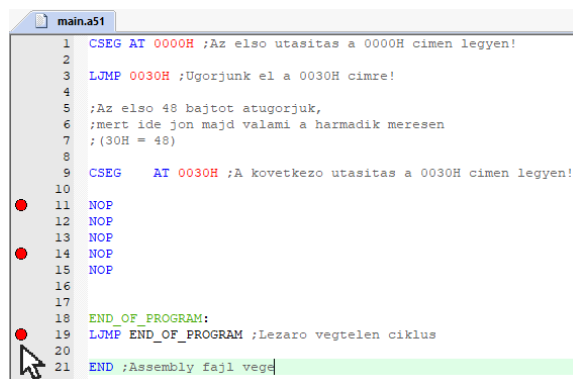
A program megírása után kattintsunk a *Rebuild* ikonra. A gomb megnyomásával elindul a preprocessor, az assembler és a linker. A későbbiekben a Rebuild ikon megnyomása helyett a **fordítás** szó lesz használva. A fordítás után szimulációra kész a program!



14. ábra. A program "fordítása"

1.5.2. Breakpointok elhelyezése

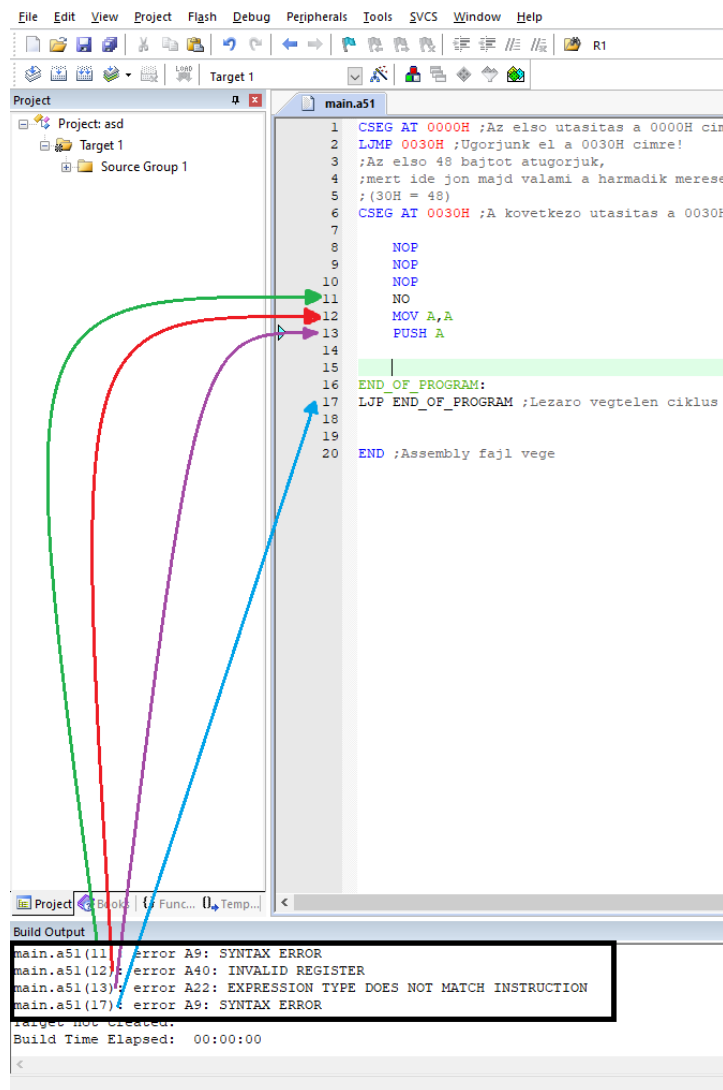
Ha lefordítottuk a kódot, akkor helyezzünk el breakpointokat (piros bogyókat) a kódban. Ezt úgy tehetjük meg, hogy a kívánt utasításnál látható sorszám mellett kattintunk egyet bal egérrel. A breakpointok segítségével szimuláció és debug közben megállítható a program futása a kijelölt soroknál. Tegyük breakpointokat az első, illetve az utolsó előtti NOP utasításhoz, valamint a lezáró ciklushoz!



15. ábra. Breakpoint elhelyezése

Helyesírás: sajnos assemblyben is van ilyen

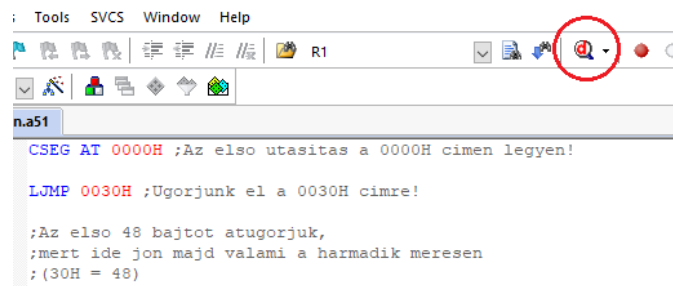
Megeshet, hogy ha véletlenségből elgépettél egy utasítást vagy a kommenteket/címkéket nem helyesen írtad be, akkor nem fog lefordulni a programod. **A fejlesztői környezet jelzi majd neked, mi a hiba, és hol található.** A példaprogramok 1:1-ben lettek a mérési útmutatóba bemásolva, így biztosan hibamentesek. Ha hibát vétettél volna, akkor ezt majd így keresd:



16. ábra. Itt láthatod, milyen hibákat vétettél

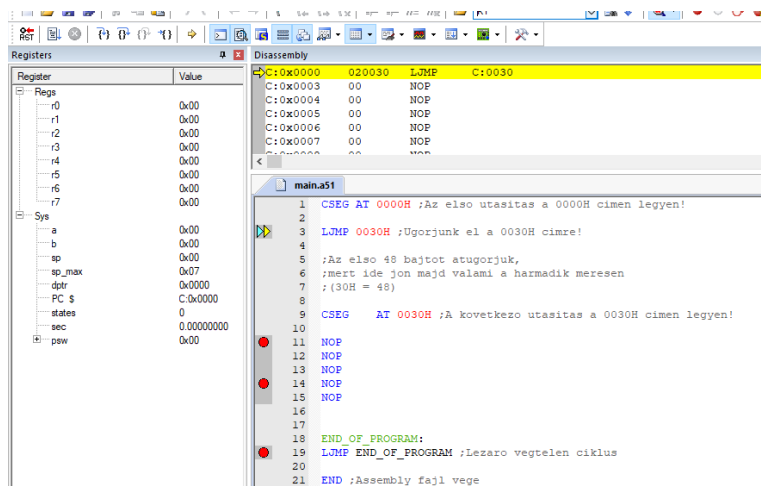
1.5.3. Indítsuk el a szimulációt!

A breakpointok beállítása után keressük meg a felső menüsorból a piros 'd' betűt kiemelő nagyító ikonját. **E gomb segítségével indítható el a szimulátor, vagy tölthető fel a mikrokontrollerre a program, ha hardveres mérést végzünk.** A 'd' betű a debug (magyarul hiba-keresés) szóra utal. A gomb megnyomása után egy figyelmeztető ablak ugrik fel, ami szól nekünk, hogy a program ingyenes változatával maximum 2 kilobyte méretű programot futtathatunk. Ezt a határt mi a Digitális Technika tantárgy során nem fogjuk átlépni, így kattintsunk nyugodt szívvel az OK gombra!



17. ábra. Program indítása

Ha idáig mindent jól követve dolgoztunk, akkor a szimuláció elindult, és az alábbi képernyő fogad minket:

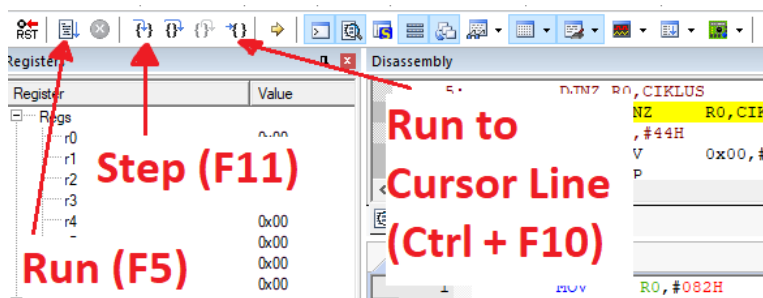


18. ábra. Elindult a szimuláció

1.5.4. Lépkedjük a programban!

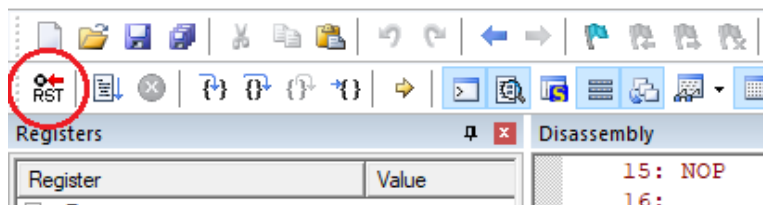
A program csak akkor kezd el futni, ha arra utasítást adunk a szimulátor-nak. A szimulátor ablakban két színes háromszög (nyíl) látható induláskor a LJMP 0030H utasítás mellett. A szövegszerkesztőnél keresd a nyilakat! A disassembly ablakban (felül) is látható egy sárga nyíl, de ezzel az ablakkal nem fogunk foglalkozni a Digitális Technika tantárgy során.

A **sárga nyíl** jelzi, hogy éppen ennél a sornál tart a program végrehajtása, ami nem meglepő, hiszen ez az első utasításunk. **Sorról-sorra léptetni a toolbar Step gombjával vagy az F11 billentyűvel lehet.** Ekkor a sárga nyíllal jelölt utasítást elvégzi a szimulátor, és átugrik a következő utasításra. **Lépkedj végig a programon!** Amikor egy utasítást végrehajtottunk, akkor annál a sornál kizöldül a szimulátor (ott, ahol a sárga nyíl korábban volt.)



19. ábra. A toolbar

Ha a program futását újra szeretnénk indítani, akkor a bal felső menüsorban meg kell nyomni a **RESET** gombot. Ekkor a mikrokontroller újraindul, és visszaugrunk a 0000H címre. Figyelem: a reset hatására nem áll vissza minden memória tartalma 0-ra! **Reseteld le a mikrokontrollert!**



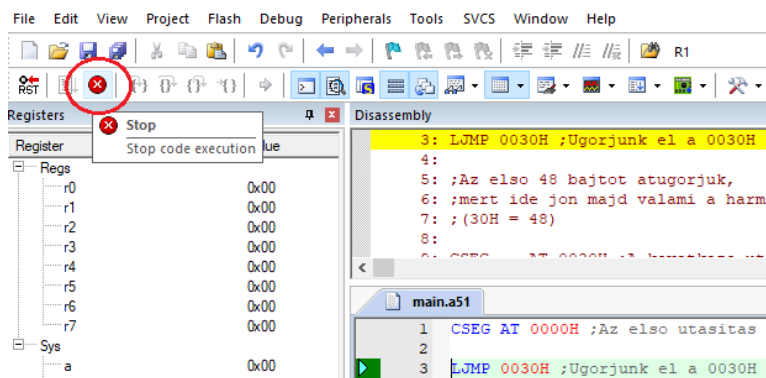
20. ábra. Reset gomb

A **kék nyíl** jelzi, hogy melyik sort jelöltük ki az egerünkkel. Megeshet, hogy egy szimuláció során egyszerre több sort is le szeretnénk futtatni, de lusták vagyunk többször is soronként léptetni. Ekkor ha kijelölünk egy utasítást, majd megnyomjuk a **Run to Cursor Line (fuss a kurzor pozíciójáig)** parancsot vagy a Ctrl+F10 billentyű kombinációt, akkor a kék nyíllal jelölt sorig fut a program, majd ott megáll. Innen pedig tovább tudjuk léptetni a programot soronként, ha szükséges. **Kattints rá az utolsó NOP utasításra, és nyomd meg a Run to cursor line parancsot!**

Hoppá! Nem fut le a program az utolsó NOP utasításig, hanem megáll az első piros bogyónál. A breakpointok segítségével megállítottuk a program futását. **Jelöld ki újra az utolsó NOP utasítást, majd futtasd ismét a kurzor pozíciójáig a programot!** Most az utolsó előtti NOP-ig futott a program, hiszen oda is tettünk egy breakpointot.

Mi van, ha mi megállás nélkül szeretnénk futtatni a programot? Természetesen erre is van lehetőség! **Reseteld le a mikrokontrollert, majd vedd ki az összes breakpointot (kattints az egérrel a piros bogyókra, hogy eltűnjenek!)**

Megint újraindult a program. Most sorról-sorra léptetés és a kurzor pozícióig futtatás helyett **nyomd meg Run gombot vagy az F5 billentyűt!** A program most megállás nélkül, folytonosan fut, amit onnan is látunk, hogy eltűnt a sárga nyíl. Ha meg szeretnénk állítani a programot, akkor meg kell nyomnunk a nagy piros X gombot a toolbaron. **Keresd meg a felső menüsorból a piros X gombot, majd nyomd meg!**



21. ábra. A szimuláció megállítása

Ha a Run gombbal (vagy az F5-tel) futtatjuk a programot, de szeretnénk megállítani azt egy adott sornál, akkor breakpointot kell tennünk a kódba. **Reseteld le a mikrokontrollert, tegyél az utolsó NOP utasításhoz egy breakpointot, majd nyomd meg a Run gombot (F5)!**

Ennyire egyszerű a program futtatása! Nézzük meg a következő lépésekben, hogyan lehet nyomon követni, mennyi ideig fut a program, illetve hogyan lehet monitorozni a regiszterek és a memóriák tartalmát!

1.6. Adjunk össze: a regiszter ablak használata

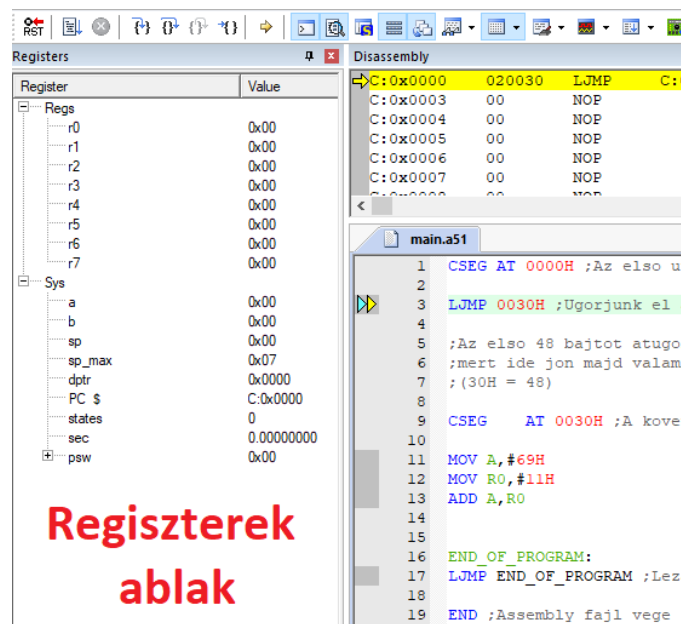
A szimulátorban a bal oldalon található a regiszter ablak. Itt tudjuk nyomon követni a legfontosabb regiszterek értékeit. Látható az R0-R7 regiszterek tartalma, az A, B, SP és a DPTR értéke. Itt találjuk a program-számláló (PC) aktuális értékét, illetve azt is, hogy éppen hány gépi ciklus, valamint mennyi idő telt el az első utasítástól kezdve.

Írjuk meg a következő egyszerű összeadó programot! **Mielőtt nekiállsz beírni, lépj ki a szimulátorból! Ezt úgy teheted meg, hogy a 'd' betűt mutató nagyítóra rákattintasz!**

```
1      CSEG      AT      0000H
2      LJMP      0030H
3      ;Az első 48 bajtot atugorjuk,
4      ;mert ide jön majd valami a harmadik meresen
5      CSEG      AT      0030H
6      MOV       A, #69H
7      MOV       R0, #11H
8      ADD       A, R0
9      END_OF_PROGRAM:
10     LJMP      END_OF_PROGRAM ;Lezaro vegtelen ciklus
11     END              ;Assembly fajl vege
```

Ha megírtad a programot, akkor **fordítsd le a Rebuild gombbal, majd indítsd el a szimulációt ('d' betűs nagyító!)** Mielőtt lépésenként ellenőriznénk a programot, nézzük meg, mit kellene kapnunk az összeadás (ADD A,R0) elvégzése után! **Számold ki papíron, mennyi lesz az A regiszter értéke az ADD utasítás után! Figyelem: HEXADECI-MÁLIS számokkal dolgozunk!**

Ha kiszámoltad, mennyi az annyi, akkor vessünk egy pillantást a Regiszterek ablakra! Felül láthatjuk az éppen kiválasztott regiszter bankban található R0-R7 regisztereket, alul pedig még az ezeknél is érdekesebb SFR-ek értékét találjuk. Ezeken felül még itt látható az eltelt idő, és a végrehajtott gépi ciklusok száma is.



22. ábra. Registers ablak

Nézzük mi mit jelent. **Figyelem, az összes regiszternél ahol 0x előtag van a szám előtt, az hexadecimális értéket jelent!**

Regs	az R0-R7 regiszterek értéke
A	az akkumulátor értéke
B	a B regiszter értéke
SP	a stack pointer (veremmutató) értéke
SP_max	a veremmutató eddigi legnagyobb értéke
DPTR	a datapointer értéke (16 bites érték)
PC \$	a programszámláló aktuális értéke (16 bites érték)
States	az elvégzett gépi ciklusok száma
Sec	az eltelt idő másodpercben
PSW	a Program Status Word regiszter értéke

Léptesd a programot soronként, majd figyeld meg, hogyan változnak az értékek a Registers ablakban! Annyit kaptál eredményül az A regiszterben, amennyit korábban kiszámoltál papíron?

FELADAT: Írd le a jegyzőkönyvbe a számolt és a kapott értéket összeadás után. Mennyi lesz a programszámláló értéke az ADD A,R0 utasítás után? Hány gépi ciklus telt el eddig?

1.6.1. A gépi ciklusok száma és az eltelt idő kapcsolata

Ahogy előadáson tanultuk, egy gépi ciklus 12 órajel periódusból áll. Minden egyes órajel periódusban történik valami a 8051-en belül, de egy utasítás elvégzéséhez sok-sok részfeladatot kell megcsinálnia a mikrokontrollernek. Be kell olvasnia az utasítás opkódját a kódmemóriából, be kell olvasnia az utasítás operandusait, például össze kell adnia a két operandust, el kell tárolnia az eredményt a memóriában és még a programszámlálót is növelnie kell. A sok kis részfeladat elvégzéséhez minimum egy, maximum négy gépi ciklus szükséges.

Ha tudjuk az órajel frekvenciáját és azt is, hogy hány gépi ciklus alatt futott le a program, akkor könnyen ki tudjuk számolni, ez másodpercben mennyi időt jelent. Az órajel frekvenciáját már megadtuk a projekt beállításainál az Xtal ablakban, aminek az értéke a megjegyezhetetlen 11.0592 MHz volt. Számold ki, mennyi ideig tart a program végrehajtása, ha az ADD A,R0 utasítást még lefuttatod, de a lezáró végtelen cikluson már **nem** léptetsz!

$$t_{futas} = N_{gepiciklus} \cdot 12 \cdot \frac{1}{f_{clk}}$$

FELADAT: Mennyi ideig tartott a program futása a szimulátor szerint? Vesd össze azzal, amit kiszámoltál! Hány utasítást végeztük el? Ezek az utasítások egyenként hány gépi ciklust vesznek igénybe? Használd az utasításkészletet tartalmazó lapot (megtalálható a mérési útmutató végén is!) Mennyi idő alatt futna le a program akkor, ha az órajel frekvenciája 12 MHz lenne? Miért kényelmesebb számolni a 12 MHz-es órajellel? *Hint: hány órajel alatt történik meg egy gépi ciklus?*

1.7. Memóriák elérése

Programozás során a kiszámított eredményeket a szabadon felhasználható memóriában tároljuk el, hiszen az SFR regisztereknek speciális szerepük van. Ismerjük meg a DATA, az IDATA és az XDATA memóriák kezelését!

1.7.1. DATA memória elérése

A **DATA** memóriát **direkt** módon tudjuk elérni. Ez azt jelenti, hogy ha adatot szeretnénk mozgatni, akkor a forrás vagy a cél mindig egy konkrét cím, például 50H. **Lépj ki a szimulációból, gépeld be, majd fordítsd le az alábbi programot:**

```
1      CSEG      AT      0000H
2      ;Az elso utasitas cime: 0000H
3      LJMP      0030H      ;Ugorjunk el a 0030H cimre!
4
5      ;Az elso 48 bajtot atugorjuk,
6      ;mert ide jon majd valami a harmadik meresen
7      ;(30H = 48)
8      CSEG      AT      0030H      ;A kovetkezo utasitas cime: 0030H
9
10
11      MOV      30H,#40
12      MOV      50H,#32
13      MOV      30H,50H
14
15
16      END_OF_PROGRAM:      ;Lezaro vegtelen ciklus
17      LJMP      END_OF_PROGRAM
18
19      END      ;Assembly fajl vege
```

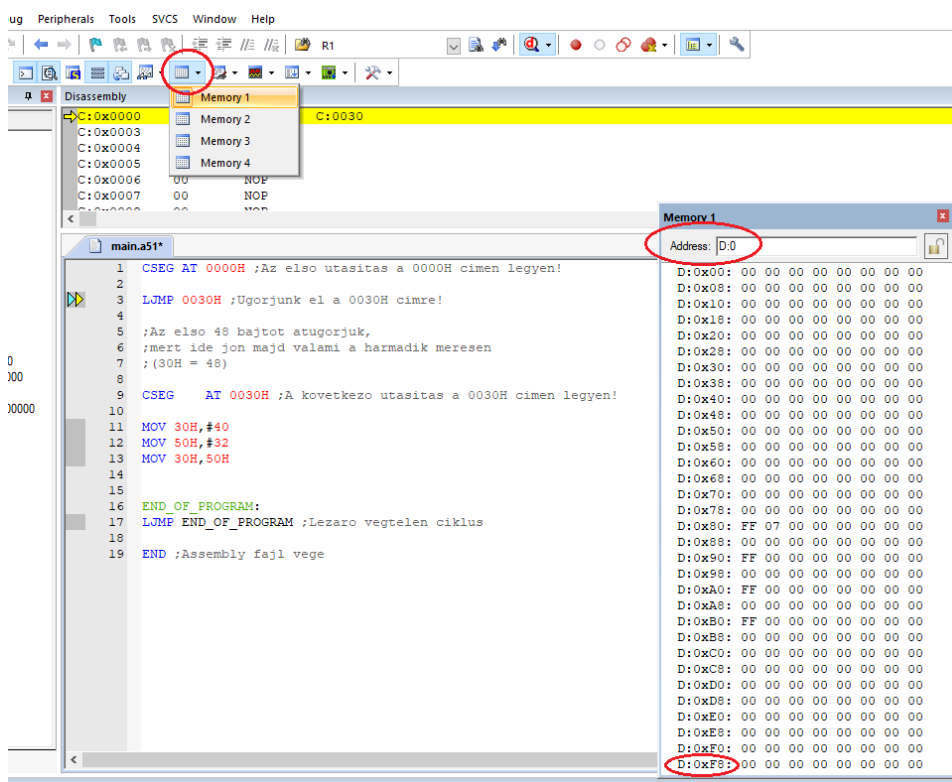
FELADAT: Mit gondolsz, mennyi lesz a 30H című memóriarekesz tartalma a program futása után?

- a) 28H b) 20H c) 50H d) 40H

A memory ablak megismerése

Indítsd el a szimulációt, majd keresd meg a felső menüsorban a Memory gombot. Nyisd meg a Memory 1 ablakot, utána pedig helyezd valahova a forráskód mellé. Az ablak tetején megadható, hogy milyen címtől kezdődően szeretnénk látni egy adott memóriaterületet, illetve hogy melyik memóriaterület érdekel. Nekünk most a DATA memória a fontos, méghozzá az egész területet látni szeretnénk.

Írd be az ablakba, hogy D:0! Ha sikerrel jártál, akkor méretezd át az ablakot úgy, hogy 1 sorba 8 bájtt férjen ki, illetve az utolsó sor kezdőcíme *0xF8* legyen! Valahogy így:

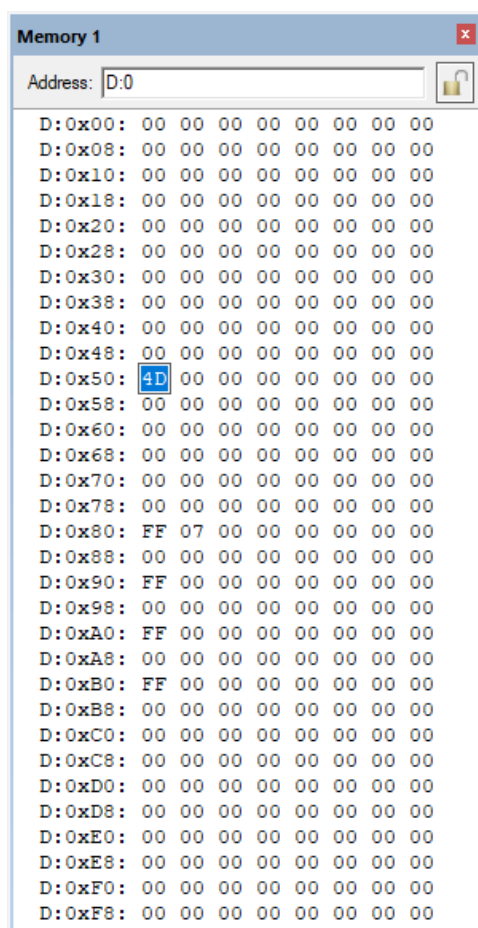


23. ábra. DATA memory ablak

Ebben az ablakban láthatod az egész DATA memóriát!

Ha a memória ablak bevetésre kész, akkor léptesd végig a programot, majd figyeld meg, hogyan változik a DATA memória tartalma. A sorok elején látod a kezdőcímeket, balról jobbra pedig elemenként növekszenek a címek. **Azt az eredményt kaptad, amire számíottál?**

Reseteld le a mikrokontrollert, majd kezd el léptetni ismét a programot! Ha elértél a MOV 30H,50H sorhoz, akkor állj meg, azt még ne futtasd le. Menj a memória ablakhoz, majd kattints rá duplán bal egérrel az 50H című rekeszre, és írd bele egy tetszőleges 8 bites HEXADECIMÁLIS számot (például 4D) majd nyomj Entert! Ha kész, akkor léptess egyet a programon! **Milyen érték került bele a 30H című rekeszbe?** Így lehet futás közben manuálisan beleírni a memóriába. Ezt nem a program végzi, hanem mi külsőleg "rondítunk" bele a memória tartalmába.



Address	Value
D:0x00:	00 00 00 00 00 00 00 00
D:0x08:	00 00 00 00 00 00 00 00
D:0x10:	00 00 00 00 00 00 00 00
D:0x18:	00 00 00 00 00 00 00 00
D:0x20:	00 00 00 00 00 00 00 00
D:0x28:	00 00 00 00 00 00 00 00
D:0x30:	00 00 00 00 00 00 00 00
D:0x38:	00 00 00 00 00 00 00 00
D:0x40:	00 00 00 00 00 00 00 00
D:0x48:	00 00 00 00 00 00 00 00
D:0x50:	4D 00 00 00 00 00 00 00
D:0x58:	00 00 00 00 00 00 00 00
D:0x60:	00 00 00 00 00 00 00 00
D:0x68:	00 00 00 00 00 00 00 00
D:0x70:	00 00 00 00 00 00 00 00
D:0x78:	00 00 00 00 00 00 00 00
D:0x80:	FF 07 00 00 00 00 00 00
D:0x88:	00 00 00 00 00 00 00 00
D:0x90:	FF 00 00 00 00 00 00 00
D:0x98:	00 00 00 00 00 00 00 00
D:0xA0:	FF 00 00 00 00 00 00 00
D:0xA8:	00 00 00 00 00 00 00 00
D:0xB0:	FF 00 00 00 00 00 00 00
D:0xB8:	00 00 00 00 00 00 00 00
D:0xC0:	00 00 00 00 00 00 00 00
D:0xC8:	00 00 00 00 00 00 00 00
D:0xD0:	00 00 00 00 00 00 00 00
D:0xD8:	00 00 00 00 00 00 00 00
D:0xE0:	00 00 00 00 00 00 00 00
D:0xE8:	00 00 00 00 00 00 00 00
D:0xF0:	00 00 00 00 00 00 00 00
D:0xF8:	00 00 00 00 00 00 00 00

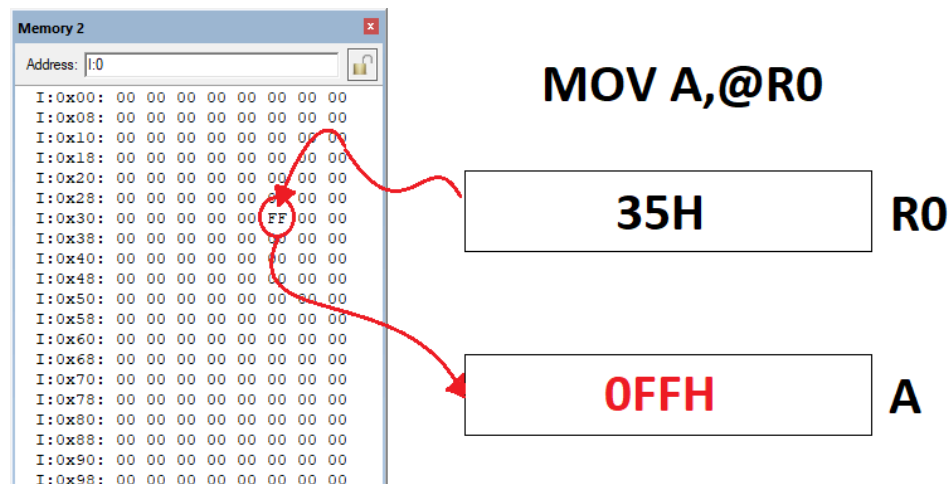
24. ábra. A memória átírása

1.7.2. IDATA memória elérése

Az **IDATA** memóriát **indirekt** módon érhetjük el az **R0** és az **R1** (röviden **Ri** regiszterek segítségével!

Mit is jelent az indirekt elérés? Arra a rekeszre vagyunk kíváncsiak, amit az Ri regiszter "mutat". A mutató pedig azt jelenti, hogy abból a memóriarekeszből kell kiolvasnunk egy értéket, aminek a címe az Ri értéke. Ha belegondolunk, akkor ezt a @ jel (:at → -nál/nél) nagyon jól szemlélteti. Nézzünk rá egy példát, aztán programot is írunk rá a 1.7.3. alfejezetben. Most csak a program egy részletét látjuk. Képzeljük ide a CSEG AT, END és a többi megismert elemet is!

```
1 ;Olvassuk ki a 35H cimu memoria rekesz tartalmat indirekten!  
2     MOV     R0 ,#35H  
3     MOV     A ,@R0
```



25. ábra. Indirekt olvasás

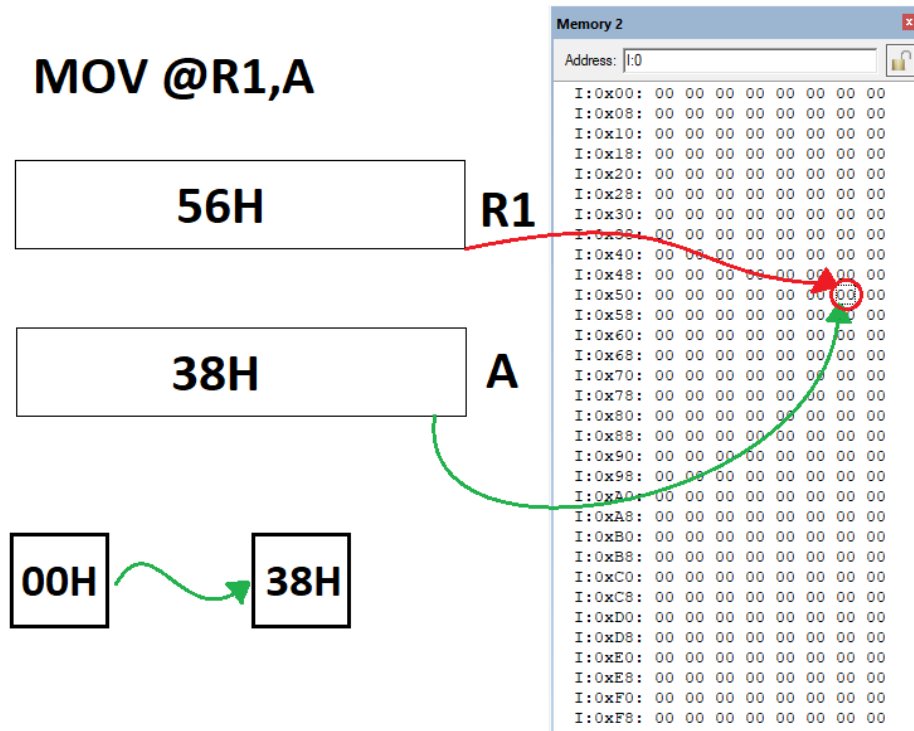
A hihetetlenül jó Paint skillekkel szerkesztett képen azt láthatjuk, hogy az R0 regiszter értéke 35H. A girbe-gurba piros nyíl mutatja, hogy a 35H címen lévő memóriarekesz érdekes a számunkra és bármi is legyen benne, azt rakjuk bele az A regiszterbe. Mivel 0xFF (0FFH) érték található ebben a rekeszben, ezért ez kerül bele az akkumulátorba! Azaz az R0 regiszter segítségével (indirekt módon) mozgattunk adatot az A-ba! Easy-peasy, ugye?

Az indirekt elérés írásra is érvényes. Abba a memóriarekeszbe írunk egy értéket, amelynek a címe az Ri regiszter tartalma. Nézzünk erre is egy példát!

```

1 ;Irjunk bele 38H erteket az 56H cimu memoriaba indirekt modon!
2     MOV     R1 ,#56H
3     MOV     A ,#38H
4     MOV     @R1 , A

```



26. ábra. Indirekt írás

Mit látunk a képen? Azt, hogy bármi is volt az R1 által címzett helyen, oda a **MOV @R1,A** utasítás elvégzése során az akkumulátor értéke (**38H**) kerül!

1.7.3. DATA-IDATA közti kapcsolat

Ha szemfüles voltál, akkor észrevehetted, hogy míg a direkt módon címzett DATA memória kezelésénél a memória ablakba a **D:0** cím volt írva, az indirekt elérésű IDATA példánál viszont már **I:0** volt a kezdőcím. Miben más a kettő, vagy teljesen ugyanazok lennének? Nézzük meg egy próba programmal!

Innentől kezdve a példákban már nem látjuk a teljes forráskódot, csak a lényegi részüket. A CSEG AT 0030H direktíva és az END_OF_PROGRAM: címke közé kell mindent beírni, ahogy eddig is tettük!

Mielőtt kilépsz a szimulátorból, nyisd meg a Memory 2 ablakot, rakd ki a D:0 ablak mellé, és írd bele, hogy I:0! Utána méretezd át ezt az ablakot is úgy, ahogy a DATA memóriánál is csináltad! Ha kész, akkor lépj ki a szimulátorból, és írd be az alábbi programot!

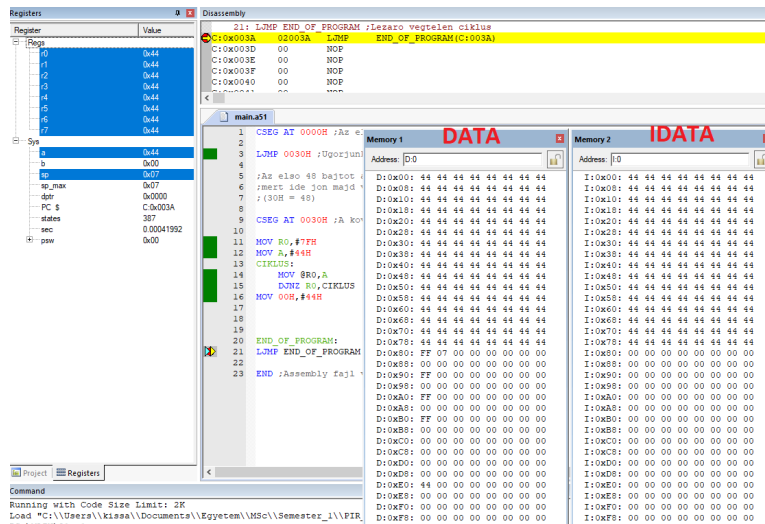
```
1      MOV      R0 ,#7FH      ;Hanyszor fusson a ciklus?
2      MOV      A ,#44H      ;Mit akarok lepakolaszni?
3  CIKLUS:
4      MOV      @R0 ,A        ;Hova akarok pakolni?
5      DJNZ     R0 ,CIKLUS    ;Csökkentem R0-t, aztan R0>0?
6      MOV      00H ,#44H    ;Az utolso cimet direkten irom!
```

Megjelent egy új vezérlési elem: a **ciklus**, amit a DJNZ (decrement and jump if not zero → csökkentsd eggyel és ugorj a címkére, ha nem nulla) ciklus-szervező utasítással valósítottunk meg!

A már megismert indirekt címzési módszerrel 7FH címtől kezdődően feltöltöttük a memóriát 44H értékkel csökkenő sorrendben. Mivel az R0 értéke minden iterációnál csökken a DJNZ miatt, ezért a címzett memória is szépen halad 7FH-től a 00H felé. Na de a 00H címet már a cikluson kívül töltjük fel (ráadásul most direkt módon), hiszen a DJNZ csak akkor ugrik vissza a CIKLUS címkére, ha még nem nulla az R0 értéke a dekrementálás után. Amikor az R0 értéke 01H és érkezik a DJNZ utasítás, akkor az R0 értéke pont 00H lesz, azaz ilyenkor már NEM ugrik az utasítás!

Inkább nézd meg a szimulátorban, úgy minden világos lesz! Lépkedj végig párszor a cikluson, nézd meg mi történik a memória ablakokkal! Ha meguntad, hogy ismétlődik a folyamat, akkor tegyél egy breakpointot a **LJMP END_OF_PROGRAM** utasításhoz, és futtasd a programot a Run gombbal!

Ha jól sikerült minden, akkor ezt kell látnod: dugig lett rakva a memória 44H értékkel!

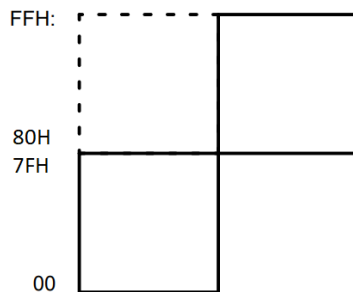


27. ábra. 44H 44H 44H 44H 44H 44H 44H 44H 44H 44H

FELADAT: Hogy lehetséges, hogy az **indirekt** módon címzett **IDATA** memória rekeszei és a **direkt** módon címezhető **DATA** memória rekeszei teljesen megegyeznek a 7FH címig? És utána miért különbözik a kettő? Emlékeztet valamire a 28. ábra? Írd le a jegyzőkönyvbe a választ!

Hány alkalommal futott le a **ciklus**?

- a) 7FH b) 80H c) 44H d) Nem futott le



28. ábra. Előadásról ismerősnek kell lennie!

1.7.4. Mi van a 7FH címen túl?

Az előadáson tanultak szerint (illetve a 28. ábra szerint is) a belső memória 7FH cím felett érdekesen viselkedik, mert itt élnek a **speciális funkciójú regiszterek**, az SFR-ek! Őket csak **direkt** címezéssel érhetjük el. Az akkumulátor (A), amit eddig használtunk már párszor, ezen a területen található. Aztán nem csak az akkumulátor, hanem a B, az SP, a DPTR (DPH és DPL felei), a PSW és egyéb más SFR is itt lapul egy adott címen. Nézzük meg, hogy viselkedik a belső memória 7FH felett, ha direkt módon címezzük! Nézzük meg azt is, hogy milyen címen rejtőznek az SFR-ek! **Gépeldd be, fordítsd és léptesd az alábbi programot!**

```
1 CIKLUS :
2     INC     A
3     INC     B
4     INC     SP
5     INC     PSW
6     ;16 bites (2 bajtos) értékkel toljuk fel a DPTR-t!
7     MOV     DPTR ,#0EFFFH
8     INC     DPTR ;Az egész DPTR-t inkrementáljuk, egyben!
9     INC     DPH ;A DPTR felső bajtját inkrementáljuk
10    INC     DPL ;A DPTR alsó bajtját inkrementáljuk
11    LJMP    CIKLUS
```

FELADAT: Figyeld meg, hogyan változik a memória ablakok (D:0, I:0) tartalma! Miért nem változik mindkét ablak úgy, mint az előző programnál? Miben különböznek az *INC DPTR* és az *INC DPH*, *INC DPL* utasítások? Milyen címen találhatóak az SFR regiszterek? Töltsd ki a táblázatot HEXADECIMÁLIS címekkel!

SFR	A	B	SP	PSW	DPH	DPL
Cím						

Mindenképpen csalni szeretnénk, és bepróbálkozunk a 7FH címen túli memóriák **indirekt** elérésével. Nézzük meg mi történik! Módosítsuk az 1.7.3. alfejezetben lévő programot úgy, hogy ne 7FH-tól induljon a ciklus, hanem 82H-tól! Fordítsd le (Build), majd futtasd (Run)!

```

1      MOV      R0 ,#082H
2      MOV      A ,#44H
3  CIKLUS :
4      MOV      @R0 , A
5      DJNZ     R0 , CIKLUS
6      MOV      00H ,#44H

```

Hoppá, hoppá! Azt vártuk volna, hogy végigfut a ciklus, és feltölti a belső memóriát 82H címtől lefelé 44H értékkel. Nem éppen ez történt. A μ Vision megállította a program futását már egyből az első indirekt címzést használó utasításnál, arra hivatkozva, hogy: *error 65: access violation at: I:0x82 : no 'write' permission*. Azaz egy olyan memóriaterülethez szerettünk volna indirekt módon hozzáférni, amihez nem lehet! Ha tovább léptetjük a programot, akkor egészen a 7FH cím eléréséig hibát fog dobni a fejlesztői környezet, és csak utána kezdi el megállás nélkül futtatni a programot.

```

Command
Running with Code Size Limit: 2K
Load "C:\\Users\\kissa\\Documents\\Egyetem\\MSc\\Semester_1\\PIR_retro'
BS \\MAIN\\6
*** error 65: access violation at   I:0x82 : no 'write' permission
*** error 65: access violation at   I:0x81 : no 'write' permission
*** error 65: access violation at   I:0x80 : no 'write' permission

```

29. ábra. Indirekt címzéssel 8051-nél csak 7FH-ig!

FELADAT: Miért dobálta a hibát a fejlesztői környezet? Mi a magyarázat rá? Rajzolj ábrát!

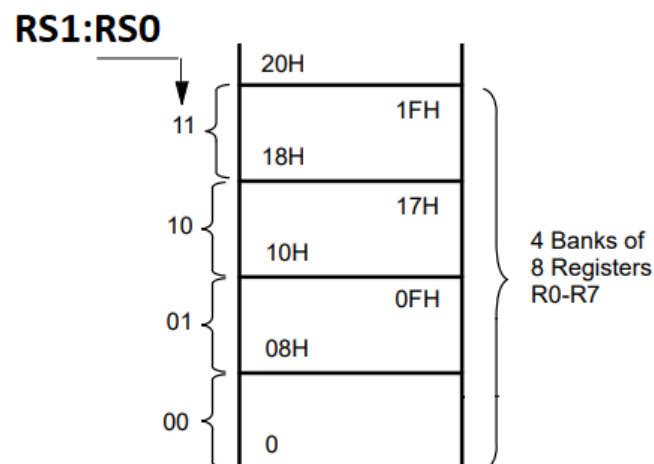
1.7.5. Te kinél bankolsz? Ismerjük meg a regiszter bankok használatát!

Az előadáson tanultak alapján a 8051-es mikrokontroller 4 regiszter bankkal rendelkezik, amik között tetszőlegesen váltogathatunk. Emiatt előáll az az érdekes helyzet, hogy 4 db létezik az R0-R7 nyolcas csoportból. Honnan tudjuk, hogy például a *MOV R6,#34H* utasítás melyik bankban szereplő R6-ot címzi? Ehhez ismerjük meg a *PSW* (*Program Status Word*) regisztert kicsit közelebbről! Az SFR-ek egy része attól speciális, hogy a bitjeikkel vezérelhetünk (vagy megfigyelhetünk) valamilyen működést a mikrokontrolleren belül, például a regiszter bankok kiválasztását.

7	6	5	4	3	2	1	0
CY	AC	F0	RS1	RS0	OV	-	P

PSW regiszter bitjei

Minket most csak két bit érdekel: **RS1:RS0**! Az RS a **Register Select** rövidítése. Kétféle mennyiség, amelyen pont négy lehetséges értéket tárolhatunk (00,01,10,11). Ezen bitek segítségével választhatjuk, ki melyik regiszter bankkal szeretnénk dolgozni. Azaz ha átírjuk a PSW értékét úgy, hogy változzanak az RS1:RS0 bitek, akkor bankot váltunk.



30. ábra. Regiszter bankok

Gépeldd be a programot, fordítsd le, indítsd el a szimulációt majd soronként léptesd. Figyeld meg, mi történik a registers ablakban az R0, R1, R7 és a PSW tartalmával! Na és mit látsz a Memory ablakokban? Pörgesd a ciklust is párszor!

```

1      MOV      PSW ,#0
2      ;0-as bankbol indulunk!
3      MOV      R0 ,#69H
4      MOV      R1 ,#33H
5      MOV      R7 ,#40H
6      ;1-es bankba váltunk at
7      SETB     RS0          ;vagy SETB PSW.3
8      CLR      RS1          ;Ismeri a uVision az RS1 nevet
9      MOV      R0 ,#13H
10     MOV      R1 ,#23H
11     MOV      R7 ,#33H
12     ;2-es bank, here we go!
13     MOV      PSW ,#10H
14     MOV      R0 ,#20H
15     MOV      R1 ,#0DH
16     MOV      R7 ,#73H
17     BANK_VALTO_CIKLUS:
18     MOV      PSW ,#0
19     ORL       PSW ,#08H
20     MOV      PSW ,#10H
21     ORL       PSW ,#08H
22     LJMP     BANK_VALTO_CIKLUS

```

FELADAT: Töltsd ki a táblázatokat!

BANK0	Érték	Cím	BANK1	Érték	Cím
R0			R0		
R1			R1		
R7			R7		
PSW		0D0H	PSW		0D0H

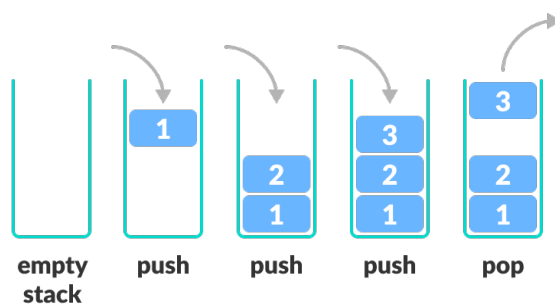
BANK2	Érték	Cím	BANK3	Érték	Cím
R0			R0		
R1			R1		
R7			R7		
PSW		0D0H	PSW		0D0H

1.7.6. A veremkezelés alapjai

Ismerjük meg a stack (verem) működését és kezelését! Mi is az a stack? Egy tárolási modell, ami LIFO (Last in first out) elven működik. Ami adat legutoljára került bele, azt tudjuk leghamarabb kivenni. A FIFO (First in first out) ennek a párja, ahol amelyik adatot elsőnek raktuk be, azt fogjuk elsőnek kivenni. Nagyon jól szemlélteti a 31. ábra mi is az a LIFO és FIFO, a 32. ábrán pedig a stack (LIFO) működését láthatod.



31. ábra. FIFO vs LIFO



32. ábra. A stack működése

A 8051-es mikrokontrollerben a stack kezelését az SP (Stack Pointer → veremmutató) SFR regiszterrel és a PUSH valamint a POP utasításokkal valósíthatjuk meg. Az SP-vel beállítjuk, hol is legyen a verem (hova akarunk pakolgatni a memóriában), a PUSH utasítással be, a POP utasítással pedig kipakolunk a veremből egy elemet (arra a címre írunk / arról a címről olvasunk a memóriából, amilyen címet az SP értéke "mutat".)

Hasonlóan működik a stack mint az *Ri* regiszteres indirekt címzés, de van egy kis extra! **PUSH** utasításnál az SP értéke automatikusan inkrementálva (+1) lesz az utasítás végrehajtása ELŐTT, míg **POP** során dekrementálódik (-1) a végrehajtás UTÁN³.

Mire kell nagyon figyelni? **A PUSH és POP utasítások egy DIREKT címet várnak, a stack pedig az IDATA területen létezik!** Nézzünk példát! Bepakolunk 16 elemet a verembe, aztán kidobjuk őket a 40H címtől kezdődően! **Gépelj, fordíts, futtass, lépkedj!** Figyeld a **memóriát és az SP értékét a registers ablakban!**

```

1  MOV     SP, #4FH           ;Induljon 50H-tol a stack!
2  MOV     A, #00H
3  MOV     R5, #0
4  BEPAKOL:
5  PUSH    ACC               ;A PUSH mellett ACC operandus kell
6  ADD     A, #11H           ;Szebben latszik, mi kerül a verembe
7  INC     R5                ;Szamoljuk, hanyszor futott a ciklus
8  CJNE    R5, #16, BEPAKOL ;Lefutott 16 alkalommal?
9
10 MOV     R0, #40H          ;Hova akarok lepakolni?
11 MOV     R5, #0            ;Szamoljuk a kipakolast is
12 KIPAKOL:
13 POP     B                 ;Pakoljunk ki a B-regiszterbe
14 MOV     A, B              ;Mozgassuk at A-ba
15 MOV     @R0, A            ;Lerakjuk 40H-tol kezdodoen
16 INC     R0                ;Kovetkezo cim
17 INC     R5                ;Szamoljuk, hanyszor futott a ciklus
18 CJNE    R5, #16, KIPAKOL ;Lefutott 16 alkalommal?

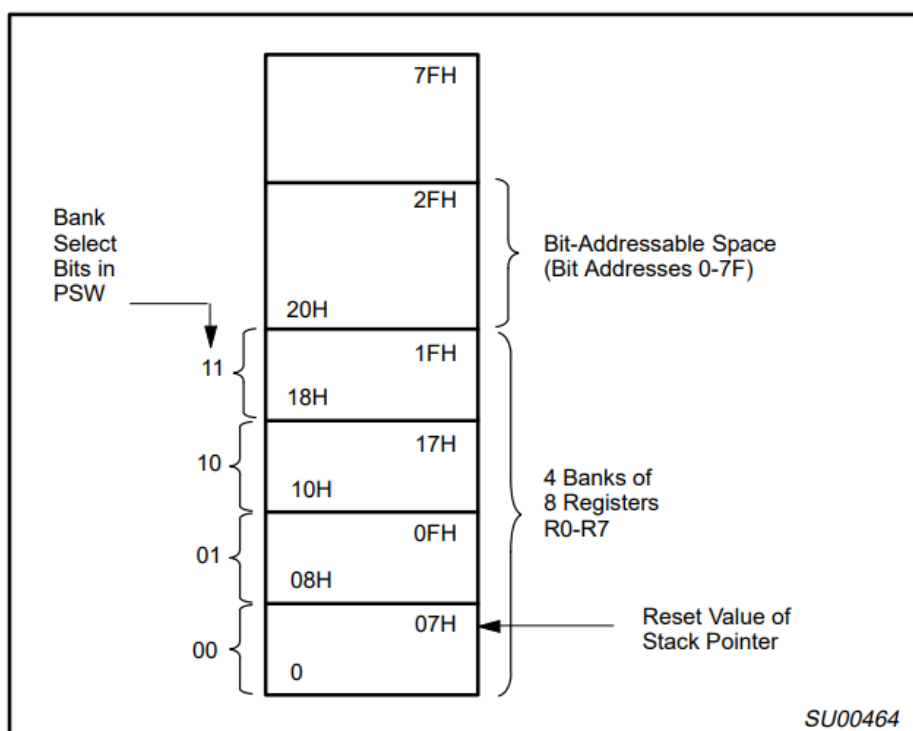
```

³Érdemes megjegyezni, hogy ez azért egy kisebb lódítás, mivel az inkrementálás/dekrementálás a végrehajtás része. A PUSH során előbb megnöveljük az SP értékét, majd írunk a memóriába, míg a POP során előbb olvasunk a memóriából, és utána csökkentjük az SP értékét. Ha csak utasításkészlet szinten ismerjük a mikrokontroller működését, akkor jogosan mondhatjuk, hogy ezek a végrehajtás előtt/után történnek, de a mikroarchitektúra szintjén ez már nem igaz.

FELADAT: Hogyan fordult a számok sorrendje a memóriában? Szerinted mi rá a magyarázat? Rajzolj is mellé!

A stack pointer értéke alapesetben 07H, azaz a 08H címtől kezdene el pakolgatni a PUSH hatására. Miért mozgattuk el 4FH-ra, hogy 50H-tól pakolásson? Mikor nem probléma, ha 07H címtől indul a verem?

Hint: Mik is vannak a belső memória elején, amikbe nem kellene beleírni random értékeket? Milyen címtől kezdve pakoltuk ki az értékeket a példában?



33. ábra. Rémlik valami?

EXTRA: Ennek a programnak van egy nagy-nagy limitációja. Mi történne, ha nem 16 elemet pakolnánk bele a verembe, hanem többet? Hogyan lehetne megjavítani a programot, hogy helyesen működjön ebben az esetben? *Hint: Bele szabad írni a stack tartalmába egy nem stack kezelő utasítással? Persze, hogy NEM! Hova kellene átmozgatni az SP-t?*

STACK OVERFLOW: Na, ilyenkor van gáz. Nagyon nagy gáz. Főleg a szubrutinoknál jön elő a probléma, de azokról majd a második mérésen. **Mi történik akkor, ha tele a stack?** Pontosabban, mi történik akkor, ha a Stack Pointer elérte a 7FH, ne adj isten a 0FFH címet?

Azt tudjuk, hogy 7FH felett nem működik a PUSH és a POP, hiszen a stack az IDATA (indirekt) területen létezik, ami csak 7FH-ig lett implementálva a 8051-es mikrokontrollernél. Szerencsére a μ Vision szól ilyen esetben, és dobálja az *error 65: access violation at I:0x80 : no 'write'/'read' permission* hibákat. Ilyenkor adatvesztés történik. Ha kitartóak vagyunk a PUSH-olgatással, akkor garantáltan el tudjuk rontani a stacket, csak akarni kell. **Módosítsuk kicsit a programot, induljon a stack pointer a 0F7H címről! Csak fordítsd, még ne futtasd!**

```

1      MOV     SP, #0F7H           ;PLS NE. CSAK EZT NE
2      MOV     A, #00H
3      MOV     R5, #0
4  BEPAKOL:
5      PUSH    ACC                ;A PUSH-hoz ACC kell
6      ADD     A, #11H
7      INC     R5
8      CJNE    R5, #16, BEPAKOL   ;Ciklus, most CJNE-vel!
9
10     MOV     R0, #40H
11     MOV     R5, #0
12  KIPAKOL:
13     POP      B                  ;Pakoljunk ki a B-regiszterbe
14     MOV      A, B               ;Mozgassuk át A-ba
15     MOV      @R0, A             ;Lerakjuk 40H-tol kezdődően
16     INC      R0
17     INC      R5
18     CJNE     R5, #16, KIPAKOL

```

Mit várunk? Az biztos, hogy a 0xF8, 0xF9, 0xFA ... 0xFE és 0xFF címekre nem fog kerülni semmi, mert az már nem implementált IDATA terület a 8051-ben (de a 8052-ben már igen). A stack pointert ez nem nagyon érdekli attól még növekszik szépen, ahogy azt kell neki minden egyes **PUSH** utasítás során, és el fog veszni az első nyolc adat. Oké, elértünk a 0xFF címre és jön a PUSH. Még 8 adatot kellene lepakolni. Na de, **mivel 8 bites az SP regiszter, ezért $0xFF + 1 = 0x00$!** Ajajajajajj..

Mi is van a belső memória 00H–07H címein? **HÁT A 0-ÁS REGISZTERBANK!** Azaz szépen elkezdjük PUSH-olgatni az adatokat a regiszterek területére. Mivel még nyolc adatot kell lepakolnunk, ezért szépen át is írjuk vele a ciklust számláló R5 (05H címen van) értékét is. Hogyan fog ezután működni a program? Hányszor fog futni a ciklus akkor? Na ez az amiről gőzünk sincs. Valamit csinálni fog a program, csak nem tudjuk mit. Ezért olyan nagy baj a stack overflow.

Lépkedj végig a kódon (kapni fogsz pár errort a fejlesztői környezettől, de hagyd figyelmen kívül az access violation-öket) és nézd meg, milyen szuper módon elrontottuk az egész programot egy stack overflow miatt. Ha csak ránézünk a forráskódra, az alapján észre sem vennénk a hibát, de ha futtatjuk a programot, akkor nagyon nem azt csinálja, amit szerettünk volna. Tanulság? **A Stack mindig olyan címen kezdődjön, hogy tudjunk bele pakolni eleget!** Ha túlsordul a stack, akkor a program működése kiszámíthatatlan lesz.

Ez még nem minden. Akkor is probléma van a verem túlsordulásával, ha például 2FH-tól indítjuk, de már a 35H címtől kezdve a stacktől független adatokat tárolunk. Ilyenkor a stack mérete öt bájt, hiszen csak a 30H–34H címekre tudunk PUSH-olni és onnan POP-pal elvenni. Ha hat adatot szeretnénk belerakni a verembe, akkor már beleírnánk a 35H című rekeszbe is, ahol már tárolunk egy másik adatot. Ha ezen a címen egy fontos adatot (például egy léptetőmotoros vonóhorog pozícióját) tárolunk, amit csak egy adott programrésznek szabad módosítania, akkor annak beláthatatlan következményei lennének⁴.

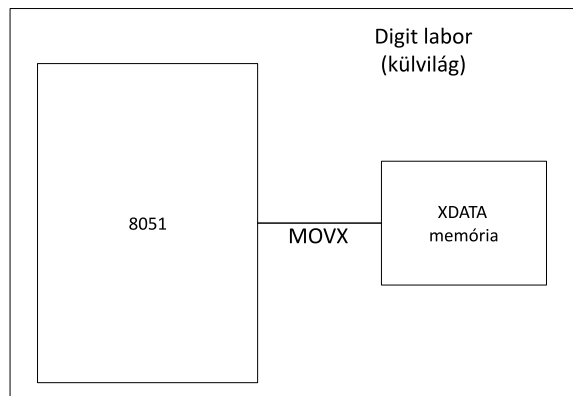
A Stack Pointert mindig olyan címről indítsd, ami felett már nem tárolsz adatokat! Aztán ha expert Assemblys leszel, akkor majd a szegmens kezeléssel dinamikusan lehet elhelyezni a stack kezdőcímét fordítási időben. Ha pedig az adataidhoz nem lenne elég a belső, általános memória, akkor pedig használd az **XDATA** területet!

FELADAT: Mi is az a stack overflow? Rajzolj ábrát!

⁴Jó, persze: minek tettük oda a Stack Pointert, vagy minek kell ott tárolni a fontos adatokat..

1.7.7. Az XDATA memória kezelése

Eddig a mikrokontroller **belső, 8 bites** címzésű memóriáját ismertük meg a gyakorlatban. Ennek a memóriának a mérete erősen limitált, hiszen összesen 256 bájt területet foglal magába, ráadásul csak az alsó 128 bájtot érhetjük el megkötések (direkt-indirekt) nélkül. Ha jelentős mennyiségű adatot szeretnénk eltárolni, akkor az **XDATA** területet kell használnunk. Az XDATA memória a mikrokontrolleren kívül helyezkedik el, azaz egy külön integrált áramkört tartalmaz. Szerencsére ezt a μ Vision tudja, így most nem kell foglalkozni a részletekkel. Móricka ábrán így kell elképzelni:



34. ábra. XDATA memória vázlatosan

Az XDATA memóriát **csak indirekten** tudjuk elérni, de már kapásból két módon. Az egyik megoldás, hogy a **DPTR-en** keresztül **16 bites címmel** érjük el, a másik pedig hogy az **Ri regisztereken** keresztül **8 bites címmel**. Az indirekt elérés ugyanúgy működik, mint ahogy azt az IDATA területnél láttuk (a piros-zöld girbe-gurba nyilak), csak most lehetőség van 16 bites címmel is dolgozni. **16 biten hány bájtot is lehet címezni? Pontosan 65536 bájtot. Jóval több mint 8 biten a 256, ugye?**

```
1 ;Olvasas
2 MOVX A,@DPTR ;Mind a 65536 bajtot elerem (A DPTR 16 bites)
3 MOVX A,@R0 ;Csak az elso 256 bajtot erem el (Az Ri 8 bites)
4 ;Iras
5 MOVX @DPTR,A ;Mind a 65536 bajtot elerem (A DPTR 16 bites)
6 MOVX @R1,A ;Csak az elso 256 bajtot erem el (Az Ri 8 bites)
```

Írd be a következő programot!

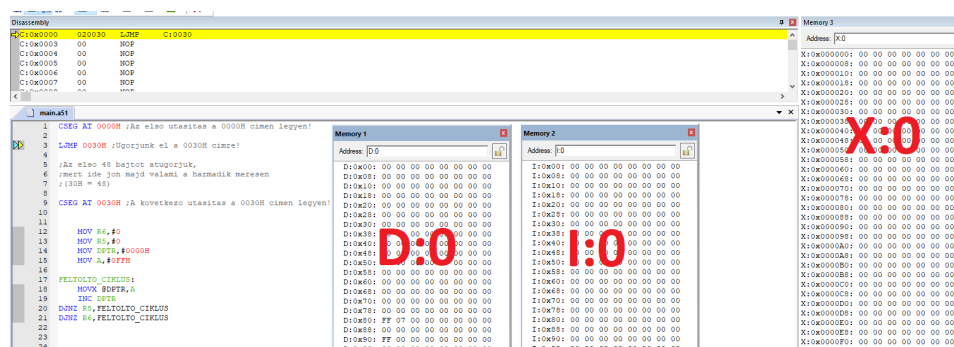
```

1      MOV      R6 ,#0
2      MOV      R5 ,#0
3      MOV      DPTR ,#0000H
4      MOV      A ,#0FFH
5
6  FELTOLTO_CIKLUS :
7      MOVX     @DPTR ,A
8      INC      DPTR
9      DJNZ     R5 ,FELTOLTO_CIKLUS
10     DJNZ     R6 ,FELTOLTO_CIKLUS
11 ;65536 alkalommal fut le a ciklustorzs.
12 ;256*256 = 65536

```

Csak a szokásos jön: fordítsd le a kódot, tegyél egy breakpointot a lezáró végtelen ciklushoz, majd indítsd el a szimulációt. **A Memory 3 ablakot nyisd meg, és írd bele hogy X:0!** Húzd ki az ablakot valahova jobb szélre, és méretezd át, hogy egy sorban nyolc bájt adat férjen ki. Ne aggódj, az egész XDATA memória tartalmát biztosan nem fog sikerülni megjeleníteni, hiszen 64 kB-ról van szó.

FELADAT: Mennyi is az a 64 kB, ha bájtokban számoljuk? Hány sora lesz az XDATA (X:0) ablaknak, ha egy sorba nyolc bájt fér ki?

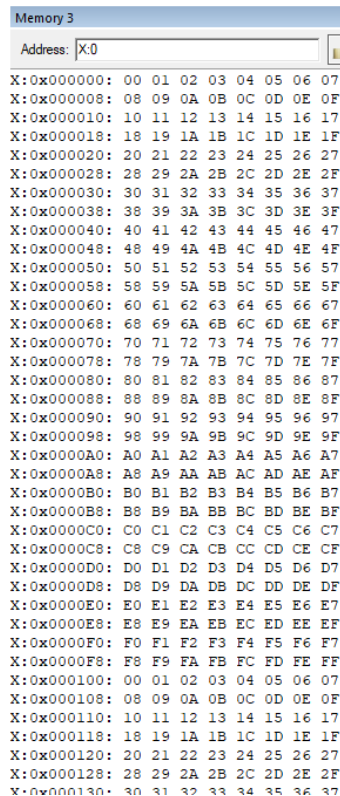


35. ábra. Az XDATA ablak is bevetésre kész!

Léptesd a programot végig a cikluson párszor, és figyeld, mi történik az X:0 ablak értékeivel! Ha meguntad a léptetést, mehet a Run (F5) a lezáró ciklusig! Teletöltöttük az XDATA memóriát 0FFH-val.

Módosítsd úgy a programot, hogy ne 0xFF értékkel legyen feltöltve a memória, hanem 0-tól kezdődően növekvő értékekkel! Írj, fordíts, futtass!

Hint: Melyik SFR értékét tudjuk beleírni az XDATA memóriába a MOVX utasítással? Azt kellene növelni minden írás után.



36. ábra. Ezt kell kapnod

FELADAT: A 0xFF után nem 0x100-nak kellene jönnie, ha hozzáadunk egyet? Miért jön mégis 0x00? A választ és a módosított programot írd le a jegyzőkönyvbe!

Próbáld meg az XDATA feltöltését a MOVX @Ri,A és INC Ri utasításokkal elvégezni! Használj a cilus előtt MOV P2,#0 utasítást! A P2 port fogja a felső 8 bitet indexelni. A második laboron beszélünk majd erről bővebben.

Egészítsük ki az előző, **módosított, növelgetős** programot egy összetett programmá! Számoljuk ki, mennyi az XDATA memória 0010H címétől kezdődően az első 7 rekesz átlaga! *Hogyan is számolok átlagot? Összeadom a számokat, majd elosztom annyival, ahány számot összeadtam* → **DIV AB**

```

1      MOV      R6 ,#0
2      MOV      R5 ,#0
3      MOV      DPTR ,#0000H
4      MOV      A ,#0
5  FELTOLTO_CIKLUS:
6      MOVX     @DPTR ,A
7      INC      A                ;az elozo feladat megoldasa :)
8      INC      DPTR
9      DJNZ     R5 ,FELTOLTO_CIKLUS
10     DJNZ     R6 ,FELTOLTO_CIKLUS
11 ;INNEN JON A KIEGESZITES
12 ;Atlag szamolasa 0010H-tol kezdve 7 szamra
13     MOV      DPTR ,#0010H ;Honnan akarok kezdeni?
14     MOV      R6 ,#7        ;Hanszor fusson a ciklus?
15     MOV      B ,R6         ;Mennyivel osztok?
16     MOV      R7 ,#0        ;Ideiglenes tarolo
17     CLR      A             ;Kezdetben 0 az osszeg
18  OSSZEADO_CIKLUS:
19     MOV      R7 ,A          ;Elmentem az A tartalmat
20                                ;mert a MOVX atirja
21     MOVX     A ,@DPTR       ;Kiolvassuk a szamot
22     ADD      A ,R7          ;Adjuk oket össze darabonkent!
23     INC      DPTR           ;Lepjunk a kovetkezo szamra!
24     DJNZ     R6 ,OSSZEADO_CIKLUS ;R6>0?
25     ;Szamoljunk atlagot!
26     DIV      AB             ;A-ban egesz , B-ben maradek

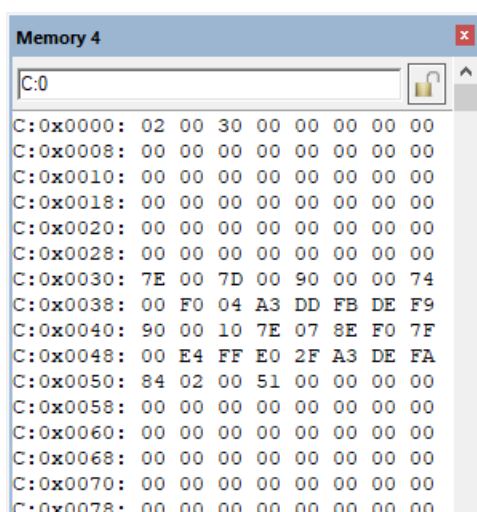
```

Ez már egy elég komplex program. Ciklus, adatmozgatás, XDATA, összeadás és osztás is van benne. **Léptesd végig a programot, és figyeld meg a regiszterek értékét a léptetés során.**

FELADAT : Írd át úgy a programot, hogy most **olvasás során** ne a DPTR-en keresztül érjük el az XDATA memóriát, hanem valamelyik Ri regiszteren keresztül! Mehet a módosított teljes forráskód a jegyzőkönyvbe! Kommentekkel együtt, természetesen :)

1.8. Pillantsunk rá a kódmemóriára!

Eddig minden memória, amit megnéztünk az RAM volt, ami tetszőleges címen **írható, olvasható**. A kódmemória a 8051-es mikrokontrolleren egy külön ROM terület, azaz **csak olvasható** memória⁵. Az összetett XDATA átlagot számoló programra nézzünk rá, hogyan is néz ki gépi kód szinten! **Nyisd meg a Memory 4 ablakot, helyezd el valahova, írd bele hogy C:0, majd méretezd át a szokásos módon!** Valami ilyesmit kellene látnod:



37. ábra. A kódmemória: ez az amit ért a mikrokontroller

Ebből a memóriából hajtja végre a mikrokontroller az utasításokat. A programszámláló (PC) címzi meg kódmemóriát, ahonnan a 8051-es beolvassa a soron következő utasítást. Amikor elindul a mikrokontroller, akkor a PC mindig a 0000H címről indul. **Mi van a 0000H címen?** Az ábra szerint 02H, ami akár hiszed, akár nem, pont a LJMP utasítás gépi kódja :)

⁵Oké, oké, valahogy írható is a kódmemória, hiszen a programot, amit futtatni akarunk, azt bele kellene tudni írni. A lényeg az, hogy ezt az assembly programból sajnos nem tudjuk megtenni, mert nincs is rá utasítás. Ha hallottál olyanról, hogy JTAG vagy ICSP, akkor pont ezek segítségével lehet beleírni a gépi kódot a kódmemóriába. Ilyenkor a memóriát nem a mikrokontroller utasításaival írjuk, hanem direktben egy programozó eszköz segítségével.

1.8.1. A gépi kódokról nagyon dióhéjban

Programozás során szerencsére nem kell tudnunk a gépi kódokat, hiszen az assembler elrejti ezeket előlünk a mnemonikok és a címkék segítségével. De úgy kerek az első mérés, ha megnézzük őket is. Például hogyan is néz ki a LJMP 0030H utasítás gépi kódban? Nyisd meg az alábbi weblapot:

https://www.keil.com/support/man/docs/is51/is51_opcodes.htm

Itt láthatod az összes létező utasításhoz tartozó opkódot. 256 darab van belőlük, amiből egy darab (0xA5) nem használt. **Keress meg a LJMP utasítást**, majd klikkelj rá és nézd meg, mit ír az oldal. Három bájtos az utasítás. és két gépi ciklust vesz igénybe. Eddig minden világos, az utasításkészletes lapon is látható. Ami érdekesebb ennél, az az "encoding" sor. Itt látható, hogyan néz ki az utasítás gépi kód szinten. Az első rubrikában az LJMP mnemonikra hallgató utasítás opkódja látható, a másik kettőben pedig a 16 bites cím, ahova ugrani szeretnénk vele. Az $A_{15} - A_8$ az Address (cím) felső bájtját (00H), míg az $A_7 - A_0$ a cím alsó bájtját (30H) jelenti. Azaz a LJMP 0030H hexadecimális gépi kódban úgy néz ki, hogy 02 00 30.

LJMP

Home » Instructions » LJMP

The **LJMP** instruction transfers program execution to the specified 16-bit address. The PC is loaded with the high-order and low-order bytes of the address from the second and third bytes of this instruction respectively. No flags are affected by this instruction.

See Also: **AJMP**, **SJMP**

LJMP addr16

C	AC	FO	RS1	RS0	OV	P
---	----	----	-----	-----	----	---

Bytes 3

Cycles 2

Encoding 00000010 00 30H

Operation LJMP PC = addr16

Example
LJMP LABEL

Memory 4	
CSEG AT 0000H	
C:0x0000	02 00 30 00 00 00 00 00
C:0x0008	00 00 00 00 00 00 00 00
C:0x0010	00 00 00 00 00 00 00 00
C:0x0018	00 00 00 00 00 00 00 00
C:0x0020	00 00 00 00 00 00 00 00
C:0x0028	00 00 00 00 00 00 00 00
C:0x0030	7E 00 7D 00 90 00 00 74
C:0x0038	00 F0 04 A3 DD FB DE F9
C:0x0040	90 00 10 7E 07 8E F0 7F
C:0x0048	00 E4 FF E0 2F A3 DE FA
C:0x0050	84 02 00 51 00 00 00 00
C:0x0058	00 00 00 00 00 00 00 00
C:0x0060	00 00 00 00 00 00 00 00

38. ábra. Utasítás → gépi kód és a kódmemória kapcsolata

Ha kellően elborult az agyad, mert Villamosságtanból sok volt már a Bode meg a Nyquist, akkor próbálj írni egy kis programot, kizárólag gépi kódokkal. **NÆZ az igazi izmozás az Assemblyvel!**

1.9. Önálló feladatok:

Zárásnak oldd meg az alábbi pici feladatokat önállóan! Puskázz nyugodtan az eddig megírt programokból, és használd a szimulátort is bátran!

- Töltsd fel az akkumulátort 82 decimális értékkel, majd másold a tartalmát 55H-ba direkt, 56H-ba pedig indirekt címezéssel!
- Cseréld meg az A és B regiszterek tartalmát Stack kezelő utasításokkal! SP jó helyen legyen! Hogyan máshogyan tudnád megcserélni még őket?
- Ciklus segítségével töltsd fel a 20H címtől kezdődő 60 bájtot 0FFH értékkel! *Hint: indirekten címezz!* Stack használatával menne?
- Másolj át az XDATA 4000H címétől kezdődően 5 bájtot a belső memória 60H címén kezdődő területre!
- **Ez új lesz:** Másold át a **Kódmemória** első 5 bájtaját a belső memória 70H címmel kezdődő területére! *Hint: Indirekten tudunk olvasni a kódmemóriából az (A+DPTR) vagy az (A+PC) címről. Használd a MOVC A,@A+DPTR, CLR A, INC DPTR utasításokat!*
- **Ez is új lesz:** Írj olyan programot, ami kiszámolja a P1 és P2 portokon lévő számok összegét, különbségét, szorzatát, hányadosát, AND, OR, XOR és NAND kapcsolatát. Pakold le az eredményeket a belső memória 40H címétől kezdődően. A P1 legyen az első operandus! *Hint: A P1 és P2 SFR regiszterek, a nevük is ez. A portokkal lehet kapcsolatot teremteni a külvilággal. A következő mérésen lesz majd szó róluk.*
- Vedd ki az egyik programból a lezáró végtelen ciklust és futtasd le a programot a Run (F5) gombbal! Mit ír a μ Vision? Mi történne a programmal, ha nem dobna hibát a fejlesztői környezet? Mi történik a 16 bites programszámlálóval ilyen esetben? *Hint: Nézz rá a kódmemóriára ott, ahol dugig van 00H-val! Milyen utasítás gépi kódja a 00H? Ezt is végrehajtja a mikrokontroller!*
Hint2: Mennyi 0xFFFF+1 16 biten? Az lesz a következő utasítás címe a 0xFFFF cím után! Mi van ezen a címen?

2. Második mérés: 41 20 6B 75 6C 63 73 3A 20 33

2.1. A mérés célja

Az előző mérésen megismerkedhettünk a fejlesztői környezettel, illetve elkezdtük az Assembly programozás gyakorlását. Ezen a mérésen végre kiderül, miért az a megjegyezhetetlen 11.0592 MHz az órajel frekvenciája, valamint a járványügyi előírásokat betartva: megtudjuk, mi is az a bit maszkolás. Ezeken felül megismerjük a párhuzamos és a soros port használatát is. Hogyan tudunk kommunikálni a számítógéppel, mi is az az ASCII kód? Ma ezekre keressük a választ.

A projekt létrehozása ugyanúgy fog zajlani, mint az első mérés során, illetve a már megismert szimulátor ablakokat mind-mind használni fogjuk. Frissítsd fel a belső memóriád, ha azóta adatvesztés történt volna :)

2.2. A párhuzamos portok használata

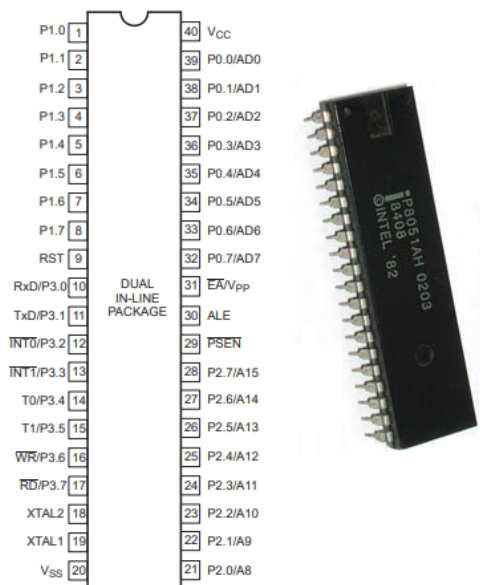
A párhuzamos portokon a külvilággal léphet interakcióba a mikrokontroller. Segítségükkel például ledet villogtathatunk, vagy éppen kapcsolók és nyomógombok állapotát olvashatjuk be, de ugyanitt tudunk UART-on (soros porton) kommunikálni egy számítógéppel vagy akár egy másik mikrokontrollerrel. Az első mérésen ha jobban szemügyre vetted a DATA memória SFR részét, akkor láthattad, hogy 4 bájt (0x80, 0x90, 0xA0, 0xB0 címeken) értéke 0FFH volt akkor is, ha nem írtuk őket direkt címezéssel. Ők lennének a portokhoz tartozó SFR-ek.

```
D:0x78: 00 00 00 00 00 00 00 00
D:0x80: FF P0 00 00 00 00 00 00
D:0x88: 00 00 00 00 00 00 00 00
D:0x90: FF P1 00 00 00 00 00 00
D:0x98: 00 00 00 00 00 00 00 00
D:0xA0: FF P2 00 00 00 00 00 00
D:0xA8: 00 00 00 00 00 00 00 00
D:0xB0: FF P3 00 00 00 00 00 00
D:0xB8: 00 00 00 00 00 00 00 00
```

39. ábra. Portok SFR regiszterei

Elméletileg a P0, P1, P2 és P3 névre hallgató portok mindegyike szabadon felhasználható, de azért ez nem teljesen igaz. **A P0, P2 portokra van kivezetve a mikrokontroller adat és címbusza. Adat? Cím? Ki-**

vezetve? *Külvilág?* Eléggé úgy hangzik mintha az XDATA memóriával állna kapcsolatban ez a két port. Ez pontosan így is van. Azaz, ha használunk XDATA memóriát (és/vagy **külső kódmemóriát**), akkor ezt a két portot nem szabadna általános célra felhasználni. Szerencsére a szimulátorban ez nem lényeges, így egy kicsi szabályszegéssel fogunk élni, és használni fogjuk őket is. **A P3-as port sem használható teljesen szabadon**, mert ide vannak kivezelve a soros port adó (TxD) és vevő (RxD) lábai, a külső memória eléréséhez szükséges Write (\overline{WR}) és Read (\overline{RD}) jelek, valamint az időzítő 1-2 funkciója is itt kapott helyet, de ezekről majd később lesz szó úgy is. A P1-es port viszont teljesen szabad, azt csinálhatunk vele, amit csak akarunk.



40. ábra. A mikrokontroller lábkiosztása és maga az integrált áramkör

Pár láb már ismerős lehet (XTAL). A **P0** portnál az AD jelzi, hogy ez a port **adat és címbusz** is egyben, míg a **P2** portnál csak A látható. Ez a **címbusz** felső bájtja, hiszen 16 bites a PC és a DPTR is! Amikor külső memóriát kezelünk, akkor erre a portra kerül ki a DPTR, PC vagy az Ri értéke. Ri esetén a felső 8 bitet a P2 port SFR-je adja. Emiatt kellett az első laboron a `MOVX @Ri, A` utasítás elé beírni a P2 port nullázását. A P3-nál is látni 1-2 másodlagos funkciót, például a P3-as port 0-ás bitje (P3.0) az RxD láb. Itt fogjuk fogadni a beérkező adatot a soros kommunikáció során,

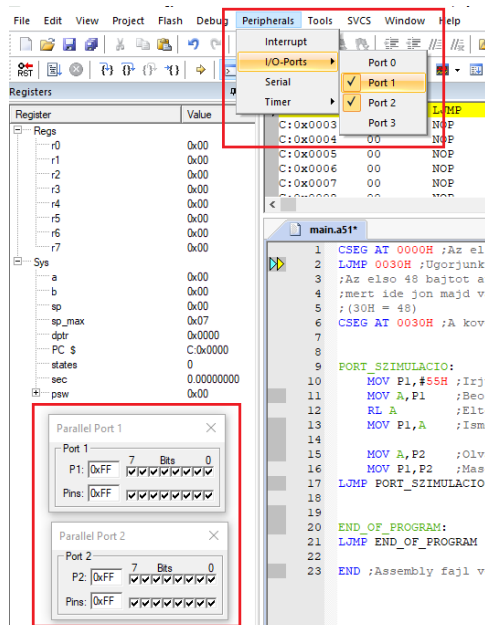
a P3.1 (TxD) lábon pedig küldeni tudunk majd.

Szimuláljuk a portokat! **Írd meg az alábbi programot, fordítsd, de még NE induljon a szimuláció!** Ne felejtse el, ismét kell az a pár extra elem a programba, amit az első mérésen láttunk! *CSEG AT, LJMP END_OF_PROGRAM!*

Ha nem ugrik még be fejből, akkor **LJMP ELSO_MERES**, és másold be a kellő részeket!

```
1 PORT_SZIMULACIO:
2     MOV    P1,#55H ;Irjuk a P1 port SFR-t
3     MOV    A,P1    ;Beolvassuk a P1 port SFR-t
4     RL     A       ;Eltoljuk balra egy bittel
5                     ;az egeszet (mert miért ne?)
6     MOV    P1,A     ;Ismet irjuk a P1 port SFR-t
7     MOV    A,P2     ;Olvasas P2-bol
8     MOV    P1,P2    ;Masolas P1-be
9     LJMP   PORT_SZIMULACIO
```

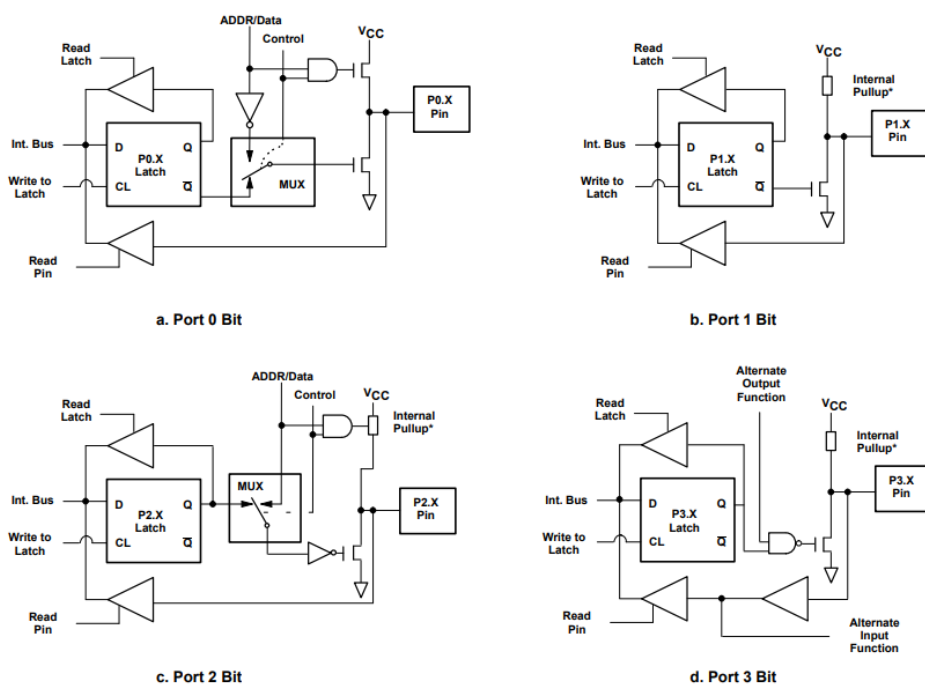
A portokat szerencsére nem csak a DATA memóriát nézegetve tudjuk monitorozni, ugyanis van erre külön ablak a szimulátorban! **Keresd meg a felső menüsorból a Peripherals → I/O Ports → Port1, Port2 opciókat. Tedd ki mindkét portot valahova!**



41. ábra. Port szimulátor ablakok

Hogyan is működnek a portok, és hogyan az ablakok? Érdekes módon nem csak a Port SFR értékét, hanem a Pineket (lábakat) is láthatjuk. Ez amiatt van, mert a kettő nem ugyanaz. Sőt, a 8051-es kétirányú (bi-direkcionális) portokkal rendelkeznek. Digit 1-ből megszokhattad, hogy a kapuknak/latch-eknek/flip-flopoknak külön van bemenete, és külön kimenete. Olyan nem volt, hogy valamelyik lábuk mindkettő lehetett volna. Na itt meg pont ez a helyzet. A portok kimenetek és bemenetek is lehetnek. Az már tőlünk függ, hogyan szeretnénk őket épp használni.

A következő ábra picit mély víz érzés lesz, mert szükséges hozzá egy kis elektronika tudás (ami a 3. félévben majd meg is lesz) de addig is, érdekességnek itt van, hogyan működnek a portok. Mindegyikből (P0,P1,P2 és P3) egy-egy port bitet látsz.



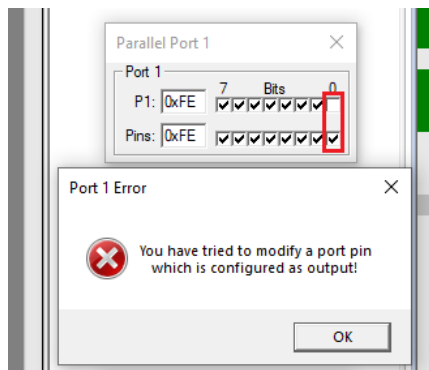
42. ábra. A portok felépítése

Innen látszik is, hogy a **Port PIN**-ek, nem ugyanazok, mint a **Port SFR**-ek! A **Pin** az, ami fizikailag ki van vezetve a mikrokontrollerből egy meghajtó fokozaton keresztül (tranzistorok), míg az **SFR** a **D** tároló.

Mivel még nem biztos hogy tanultál elektronikát, így legyen annyi elég, hogy ha egy **Port SFR** egyik bitje 1-es, akkor a port bemenet (vagy kimenet, és a belső felhúzó ellenállás miatt 1-et ad), ha viszont 0, akkor pedig CSAK kimenet lehet. A tranzisztor lehúzza 0-ra a **Pint**. Ha 0-át írunk például a P1 port SFR 0-ás bitjébe, és a P1.0 lábra mégis rákötünk egy jelet ami H szintű és nincs korlátozva az árama, akkor hello-szia mikrokontroller, szépen el is égetjük a meghajtó tranzisztort.

Ezt a fejlesztői környezet szerencsére tudja, jönni is fognak az errorok, ha hülyeséget csinálunk véletlenül. **Indítsd el a szimulációt, és lépkedj végig a programon! Figyeld meg, mi történik az akkumulátorral és a portokkal!** Ha készen vagy, akkor állítsd a **P1 és P2 SFR bitjeit az ablakban, majd nézd meg, mi történik!** Állítgatás után vissza kell majd kattintanod az egyik utasításra, hogy működjön a léptetés (Step, F11).

Töltsük ki minden mérgünket a 8051-en! Füstöljük el az egyik portot, úgy is csak virtuálisan tesszük tönkre a mikrokontrollert. **Állítsd át mondjuk a P1 SFR 0-ás bitjét 0-ba (kimenet), majd a P1 Pin 0-ás bitjét állítsd 1-be (rákötünk egy H szintű, nem áramkorlátozott jelt)! Jön az error, gratula, meg is ölted az egyik port pint :) A szimulátorban mindig a port SFR bitjeit állítsd meg, mert akkor a PIN is változik! Fordítva ez már nem igaz.**



43. ábra. P1.0 → 0xDEAD

Érdemes megjegyezni, hogy vannak olyan utasítások, melyek a port SFR, míg mások a PIN értékét olvassák. Azok az utasítások, melyek a **READ-MODIFY-WRITE** táblázatban vannak az utasításkészletes lapon, azok az SFR értékét olvassák, minden más pedig a PIN értékét olvassa!

2.3. Maszkoljuk a biteket!

Jöjjön egy kicsit egyszerűbb téma, a bit maszkolás. Eddig, amikor adatot mozgattunk egy regiszterbe, akkor az egész regiszter tartalma (mind a nyolc bit) megváltozott. Mi van, ha csak bizonyos bitek értékét szeretnénk megváltoztatni? Sajnos ezt a MOV utasítással nem tudjuk megtenni, szóval valamilyen más megoldási módot kell keresnünk.

A logikai műveleteket megvalósító utasításokkal (ANL, ORL) lehet elvégezni a bitek maszkolását. Nézzünk is rá gyorsan egy példát: **A TMOD névre hallgató SFR értékét úgy szeretnénk megváltoztatni 2EH-ről 21H-ra, hogy csak az alsó 4 bitjét változtatjuk meg és nem használhatunk MOV utasítást.** Ezt még nem kell beírnod!

```
1  MOV TMOD, #2EH ;ebből kellene 21H-t csinálni MOV nélkül
2  ANL TMOD, #20H ;lemaszkoljuk az alsó 4 bitet, hogy 0 legyen
3  ORL TMOD, #01H ;az alsó 4 bitet beallítjuk a kívánt értékre
```

Miért működik ez így? Nézzük meg bitenként miről is van szó:

$$\begin{array}{r} 0010|1110 (2EH) \\ \text{AND } 0010|0000 (20H) \\ \hline = 0010|0000 (20H) \end{array}$$
$$\begin{array}{r} 0010|0000 (20H) \\ \text{OR } 0000|0001 (01H) \\ \hline = 0010|0001 (21H) \end{array}$$

A bitenkénti ÉS művelettel lenullázzuk az alsó 4 bitet, hiszen

$$\begin{aligned} \text{valami AND } 0 &= 0 \\ \text{valami AND } 1 &= \text{valami} \end{aligned}$$

A maszkolás után pedig a bitenkénti VAGY művelettel tudunk új értéket adni anélkül, hogy megváltoztattuk volna a felső 4 bitet, hiszen

$$\begin{aligned} \text{valami OR } 0 &= \text{valami} \\ \text{valami OR } 1 &= 1 \end{aligned}$$

A bit maszkolás nem csak egy regiszter értékadásánál hasznos, hanem akkor is, ha csak bizonyos bitek értékét szeretnénk tudni. Ha a nyolcbites akkumulátorban csak az alsó három bitre vagyunk kíváncsiak, akkor egy egyszerű AND típusú maszkolással le tudjuk nullázni a nem kellő biteket. Például: a P1 porton be szeretnénk olvasni egy hárombites számot, amihez hozzá akarunk adni hetet, az eredményt pedig ki akarjuk írni a P0 portra.

```

1  MOV P1,A      ;beolvassuk a szamot
2  ANL A,#03H    ;a maszk miatt olyan, mintha 3 bites lenne a szam
3  ADD A,#7      ;hozzaadunk hetet
4  MOV P0,A      ;kiirjuk a helyes eredmenyt

```

Teszteld le a kódot! Mi történne, ha nem maszkolnánk le az akkumulátorba beolvasott értéket? Akkor is egy hárombites számhoz adnánk hozzá hetet?

FELADAT:

- Írj programot, amivel megváltoztatod az akkumulátor értékét 0FFH-ról 5AH-ra bit maszkolás segítségével!
- Írj egy másik programot, ami a P1-es és a P2-es porton bevitt számok alsó 4 bitjét összeadja, és az eredményt eltárolja a P3 porton! Teszteld is a programot! *Hint: az ADD utasítás első operandusa mindig az akkumulátor! Tárold el a részeredményeket valahova!*
- Hogyan lehet kinullázni, vagy FFH-ba állítani egy regisztert csak az AND és OR utasításokkal?
- Mit csinál az alábbi program? Mire használható az XRL utasítás a logikai műveleten kívül?

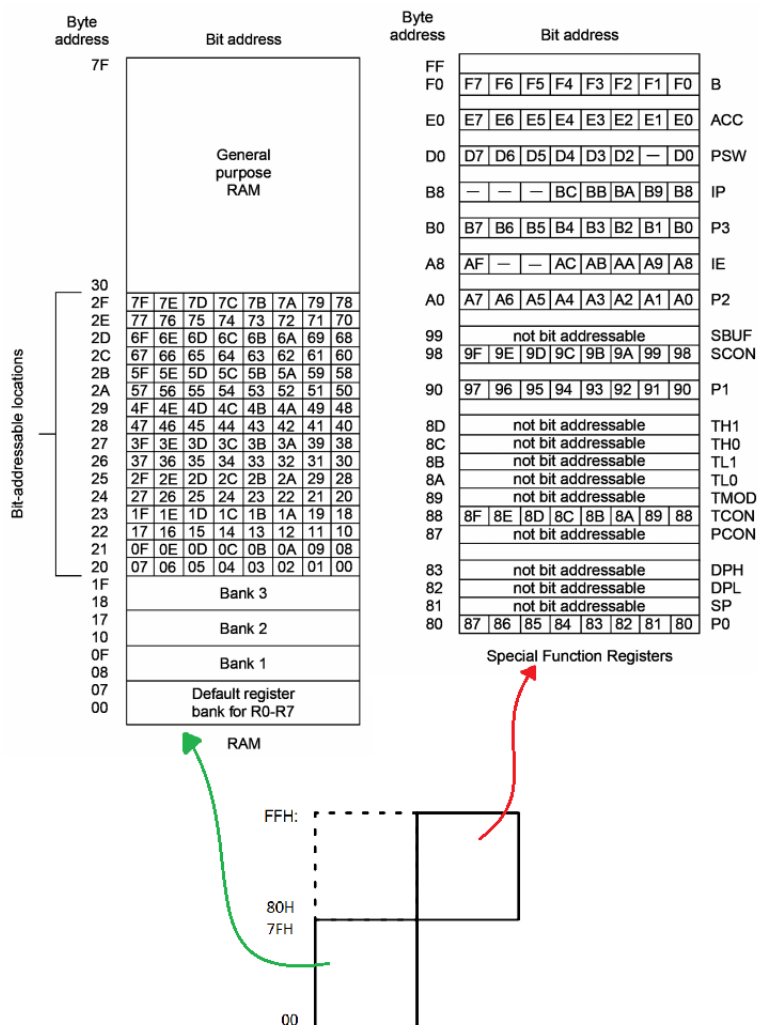
```

1      MOV      P1,#0FH
2  CIKLUS:
3      XRL      P1,#0AFH
4      ;Mire jo meg az XRL a logikai muveleten kivul?
5      LJMP     CIKLUS

```

2.4. A bitcímezhető terület és a bitszintű műveletek

Az első mérésen nem esett róla külön szó, de a **belső memória** 20H—2FH című területe továbbá bizonyos SFR-ek bitenként is elérhető. Ez azzal jár, hogy további nyolc címe van ezeknek a bitenként is elérhető rekeszeknek.
Figyelem: bitenként csak direkt módon lehet címezni!



44. ábra. A bitenként is címezhető területek

A 44. ábrán bal oldalakon láthatod a memóriát a bájtos címekkel, míg belül, minden egyes kis rekesz egy-egy bitet jelöl, amiknek külön bitcímük is van. Azaz, ha a 20H címen lévő bájtban az 5-ös bitjére vagyok kíváncsi, akkor ennek a bitnek a címe 05H!

Jó analógia a bitenként is címezhető területre a panelház. Az emeleteket a bájtos címükkel azonosíthatjuk, de az ablakokra már egy-egy saját bitcímmel is hivatkozhatunk.



45. ábra. Analógia a bitcímezhető területre

Hogyan tudunk hivatkozni az Automatika Intézet logójára a panelházas képen? **Abszolút címzéssel: 1BH** , **bájt-relatív címzéssel: 23H.3!** Azaz a bájt-relatív címzés esetében azt mondjuk meg, melyik bájtban melyik bitje érdekel minket.

Visszatérve a 8051-re: Bitcímekkel hivatkozhatunk a 20H–2FH bájtosan címezhető területen mind a 128 bitre, (16 darab 8 bites rekesz), az SFR területen belül pedig minden olyan SFR regiszterre melynek a címe 0-ra vagy 8-ra végződik. A 44. ábrán láthatod, hogy például az akkumulátor (**ACC**), a **B**, a **Port SFR-ek** és a **PSW** is **bitenként címezhető!** Ezen felül még a TCON, SCON, IE és IP névre hallgató speciális funkciójú regisztereket is lehet bitenként állítgatni, de róluk majd később esik szó bővebben.

Nézzünk példát a bitcímezhető memóriaterületre! **Gépelj be a programot, fordítsd, és léptesd a szimulátorban! Figyeld meg, hogyan változik a 20H című rekesz, az A, B és PSW tartalma!**

```

1  MOV    20H,#0A2H ;ez bajtos mozgatas
2  SETB   00H       ;a 00H BIT cimu rekeszt 1-be allitom
3  SETB   20H.3     ;a 20H cimu bajt 3-as bitjet 1-be allitom
4
5  MOV    A,#0FFH   ;az akkumulatort bajtosan feltoltom
6  CLR    ACC.4     ;a bitjeit pedig kulon is tudom allitani
7  CLR    ACC.2
8  CLR    ACC.0
9  CLR    0E0H.3    ;Ez is az akkumulatort allitja!
10
11 MOV    B,#20H
12 MOV    C,B.5     ;A carrybe mozgatom B.5-öt

```

Felmerülhet a kérdés: miért kell a bit műveleteknél **ACC** névvel hivatkozni az akkumulátorra, amikor más utasításoknál elég az **A** is? Ez az utasításkészlet miatt van így. Amikor azt írjuk hogy **A**, akkor azzal egy olyan utasítást adunk a mikrokontrollernek, amiben az egyik operandus **FIXEN** az akkumulátor, például: `MOV A,#data` vagy `MOV direct,A`. Ha viszont azt írjuk hogy **ACC**, akkor az akkumulátor SFR címét (0E0H) adjuk meg, mint operandust. Emiatt tudtuk az első mérésen csak `PUSH ACC` módon bepakolni az akkumulátort a verembe, hiszen a `PUSH` egy **direkt** címet vár, külön `PUSH A` utasítás pedig nem létezik!

Érdekeség: a `MOV A,#69` és a `MOV ACC,#69` teljesen ugyanazt fogja csinálni, viszont az utóbbi utasítás nem két, hanem három bájtos⁶. Sőt, a `MOV A,ACC`, `MOV ACC,A` és a `MOV ACC,ACC` is futtatható utasítások, még ha sok értelmük nincs is.

⁶`MOV A,#69` esetén a `MOV` opkódja és a `#69` egy-egy bájtot foglalnak, de az `A` nem foglal semmit, hiszen ennél az utasításnál mindig ez a cél regiszter. A `MOV ACC,#69` esetén viszont mind az opkód, mind a direkt című célregiszter és a `#69` is egy-egy bájtot foglal!

2.5. Logikai függvények megvalósítása mikrokontrollerrel

Evezzünk ismerős vizekre! A kombinációs hálózatokat nem csak kapukkal lehet realizálni: a mikrokontroller is képes ezeket megvalósítani, még hozzá bit szinten. Persze a hálózat sokkal lassabb lesz, hiszen egy negálás (CPL) is 12 órajelbe (1 gépi ciklus) kerül, hát még egy összetett függvény. A nagy előny viszont, hogy 4, 5 vagy akár több, pl. 24 változós logikai függvény esetén megmentjük a kapukat az IC temetőtől. Sőt, ha bármikor módosítani szeretnénk a kombinációs hálózat működését, akkor csak át kell írni a programot, és már kész is! Olcsó és gyors átervezés: a projekt menedzserek kedvence!

Az előadások során biztosan hallottál a bitműveletekről. Ezek segítségével konkrét biteken lehet műveletet végezni, megkerülve a maszkolást. **Amit mindenképp jegyezz meg: A bitműveletek akkumulátora a CY!** A CY bit a PSW regiszterben lapul a 7-es biten. Míg a bájtos műveleteknél az akkumulátorban, addig a bites műveleteknél a CY-ben kapjuk meg az eredményt. Nézzünk példát bitműveletekre! **Oldjuk meg ezt a szépséget:**

$$F = \sum^4 1, 3, 5, 7, 11, 12, 13 + X : (2, 6, 8, 15)$$

$$D \div 2^3$$

$$C \div 2^2$$

$$B \div 2^1$$

$$A \div 2^0$$

Az interneten találtam egy tök jó⁷ megoldó programot, ami ezt dobta ki eredménynek:

$$F = \overline{D}A + BA + DC\overline{B}$$

Lássuk, hogyan is kell megoldani ezt a logikai függvényt Assemblyben! **Gépeld be, fordítsd és szimuláld! A bemeneteket a P1 port szolgálja! Emlékezz: a P1 SFR bitjeit állítgasd a szimulátor ablakban!**

⁷Direkt olyan függvényt adtam meg, ahol felléphet statikus házárd, azonban ezt nem jelzi a megoldó program a minimalizálás során :P


```

1      CSEG      AT      0000H
2      LJMP      0030H
3
4
5      CSEG      AT      0030H
6
7      ;F = not(D)*A + B*A + D*C*not(B)
8      ;D --> ACC.3 (akkumulator 3-as bitje)
9      ;C --> ACC.2
10     ;B --> ACC.1
11     ;A --> ACC.0
12     ;ideiglenes tarolo 0 --> ACC.4
13     ;ideiglenes tarolo 1 --> ACC.5
14     ;ideiglenes tarolo 2 --> ACC.6
15     KOMBINACIOS_HALOZAT:
16     MOV        A,P1      ;Innen jönnek a bemenetek
17     ANL        A,#0FH    ;Az also 4 bit érdekelt csak (maszkolok)
18     ; /D*A
19     MOV        C,ACC.0    ;a 'C' A 'CY' bit másik neve
20     ANL        C,/ACC.3   ;A and not(D) (a '/' jel a tagadás)
21     MOV        ACC.4,C    ;elmentem a reszeredmenyt
22     ; B*A
23     MOV        C,ACC.1
24     ANL        C,ACC.0
25     MOV        ACC.5,C    ;ezt is elmentem
26     ; D*C*/B
27     MOV        C,ACC.3
28     ANL        C,ACC.2    ;D and C
29     ANL        C,/ACC.1   ;D and C and not(B)
30     ;DC/B + BA
31     ORL        C,ACC.5
32     ;DC/B+BA+/DA
33     ORL        C,ACC.4
34
35     CLR        A          ;Az eredmény már megvan CY-ben
36     ;kinullazhatjuk az A-t nyugodtan
37     RLC        A          ;beforgatom az eredményt A-ba!
38     MOV        P2,A       ;kipakolom az eredményt P2-re
39     LJMP       KOMBINACIOS_HALOZAT
40
41     END_OF_PROGRAM:
42     LJMP       END_OF_PROGRAM
43     END

```

FELADAT: Ellenőrizd le a logikai függvényt, hogy biztosan mentes-e a statikus házárdoktól! *Ha nem így lenne*, akkor elimináld a problémát (hurkold le), és **egészítsd ki a programot, hogy valóban helyesen működjön!** A módosított függvényt és a kiegészített programot vedd fel a jegyzőkönyvbe! Várjunk csak: felléphet egyáltalán statikus házár egy programban? Ha igen, ha nem, miért?

FELADAT: Oldd meg az alábbi 3 változós logikai függvényt önállóan, a bitműveletek használatával! Szimulálj is! A megoldást írd le a jegyzőkönyvbe!

$$F = \sum_{i=0}^3 (0, 1, 3, 5, 6)$$

$$C \div 2^2 \quad B \div 2^1 \quad A \div 2^0$$

2.6. Logikai függvények II: olvassunk a kódmemóriából!

Ha minden igaz, akkor az első mérés végén már megoldottál egy feladatot, ahol a kódmemória első öt bájtját kellett átmásolnod a belső memóriába. Mi másra lehet még használni a kódmemóriát az utasításokon kívül? **Például táblázatok, szövegek és bármilyen egyéb konstans adatok eltárolására!**

A logikai függvények megoldásánál nagyon hasznosak a bitműveletek, de kicsit sokáig tart a program futtatása. Mi lenne ha egyből az igazságtáblából tudnánk megadni az eredményt? Nézzük rá példát: oldjuk meg gyorsabban az előző feladatban megadott függvényt!

$$F = \sum_{i=0}^3 (0, 1, 3, 5, 6)$$

$$C \div 2^2 \quad B \div 2^1 \quad A \div 2^0$$

Hol 1 a függvény értéke? Csak a 0, 1, 3, 5 és 6-os bemeneti kombinációknál. Ennyit elég is tudnunk, hiszen ebből azonnal felírható az igazságtábla! Hogyan is néz ki egy igazságtábla Assemblyben?

```
1 ;CBA erteke:      0 1 2 3 4 5 6 7
2 IGAZSAGTABLA: DB 1,1,0,1,0,1,1,0 ;Mi legyen a kimenet?
```

Hmmmm. Úgy látszik a címkeket nem csak ugrásokhoz és a szubrutinokhoz használhatjuk. Jelölhetnek a kódmemóriában egy konstans táblázatot is. A **DB** kulcsszó a Direkt Bájtot jelenti, aminek segítségével megadhatjuk, hogy a kódmemóriában a címke által jelölt területen milyen konstans adatok legyenek.

A **MOVC A,@A+DPTR** utasítás úgy működik, hogy a táblázat 0. elemét megcímzi a DPTR, az akkumulátorral pedig eltoljuk a kezdőcímet például hárommal. Mi van a 3-as elemnél? 1! Ezt fogjuk bemozgatni az A-ba.

Jöjjön a megoldás! **Gépeld be, fordítsd (Build), és léptesd soronként (Step, F11) a programot! Utána mehet a Run (F5), majd állítgasd a P1 port SFR értékét, és figyeld P2-t!**

```

1      CSEG      AT      0000H
2      LJMP      0030H
3
4      CSEG      AT      0030H
5 LOGIKAI_FUGGVENY:
6      MOV       A,P1           ;P1 a bemenet
7      ANL       A,#00000111B ;AND maszkolas also 3 bitre
8      MOV       DPTR,#IGAZSAGTABLA
9      MOVC      A,@A+DPTR
10     MOV       P2,A           ;rakjuk ki az eredmenyt P2 portra!
11     LJMP      LOGIKAI_FUGGVENY
12
13 END_OF_PROGRAM:
14     LJMP      END_OF_PROGRAM
15
16 ;A tablazatok mindig a lezaro ciklus utan jonnek!
17 ;CBA erteke      0 1 2 3 4 5 6 7
18 IGAZSAGTABLA:   DB 1,1,0,1,0,1,1,0 ;Mi legyen a kimenet?
19
20     END

```

FELADAT: Gyorsabban fut ez a program a bitműveletes megoldásnál? Hány gépi ciklus alatt fut le a logikai függvény megoldása? Vesd össze a bitműveletes megoldás gyorsaságával! Mi a hátránya az igazságtáblás megoldásnak? *Hint: Arra gondolj, hogy 3 változónál nem túl nagy a táblázat, de mi van akkor, ha 4, 5 vagy 8 változós a függvény? Hány változós igazság-*

tábla lenne a maximum elméleti határ, ha 64kB-os a kódmemória?

Mi történne, ha nem maszkolnánk le az akkumulátor értékét, hogy csak az alsó három bitet figyeljük? *Hint: Hány elemű a táblázat? Ha az akkumulátor értéke például 01011001B akkor honnan mozgatunk adatot az A-ba?*

2.7. Szubrutinok használata

De jó is lenne az életünk, ha nem kellene folyamatosan spagetti kódot írni, és bizonyos elemeket újra felhasználhatnánk. Vagy éppen át szeretnénk menteni pár programrészletet a következő mérésre, mert lusták vagyunk újra kiszervezni magunkból azt, ami egyszer már működött. A szubrutinok pontosan erre jók. Megírunk egy pici programot, amit sokszor szeretnénk lefuttatni különböző helyeken a kódban, és onnantól elég csak azt mondani: *Oké, most akkor csináld meg ezt a pici programot.*

Ha a laborban lennénk, akkor a ledek és a gombokat szubrutinnal írnánk/olvasnánk, mert nem ám annyira egyszerű őket kezelni, mintha csak a portokat manipulálnánk. Ezek a perifériák ugyanis az XDATA memóriába lettek beágyazva. *Hogy mi van?* A beágyazás azt jelenti, hogy 1-2 logikai kapu és D tároló segítségével a memóriát és a perifériákat (az XDATA címtérrel) feldaraboljuk: ha pl. 0C001H cím érkezik, akkor nem a memóriába, hanem a ledekre írunk ki a `MOVX @DPTR,A` utasítással. Ha meg pl. 0C000H címről szeretnénk olvasni a `MOVX A,@DPTR` utasítással, akkor pedig nem a memóriából, hanem a nyomógombokról kapjuk meg a beolvasott értéket. **Sőt**, ráadásul mindkét periféria **negatív logikás**. Azaz akkor világít a led, ha 0 a kimenet, a gomb pedig akkor ad 1-et, ha nem nyomom le. Gondolom érted már, miért hasznosak a szubrutinok. Csak egyszer kell megírni a beolvasást/kiírást, utána elég csak meghívni a szubrutint, és a több lépéses folyamatot meg is csinálja a mikrokontroller.

Na nézzük hogyan is olvasok be a gombokról az akkumulátorba:

```
1  MOV    DPTR,#0C000H ;itt vannak a gombok
2  MOVX   A,@DPTR      ;beolvasom oket
3  CPL    A             ;invertalok, mert negativ logikasak
```

És hogyan írom ki a ledekre az akkumulátor tartalmát:

```
1  MOV    DPTR,#0C001H ;itt vannak a ledek
2  CPL    A             ;invertalok, mert negativ logikasak
3  MOVX   @DPTR,A      ;kipakolom amit ki akarok
```

Most már csak az a kérdés, hogy a szubrutint hogyan hozzuk létre..

Szubrutint egy címke és a záró **RET** utasítás jelez. Nézzük meg hogyan néz ki a ledes-gombos mini program szubrutinnal! Csak annyit szeretnénk csinálni, hogy beolvassuk a gombok értékét, majd kipakoljuk őket a ledekre!

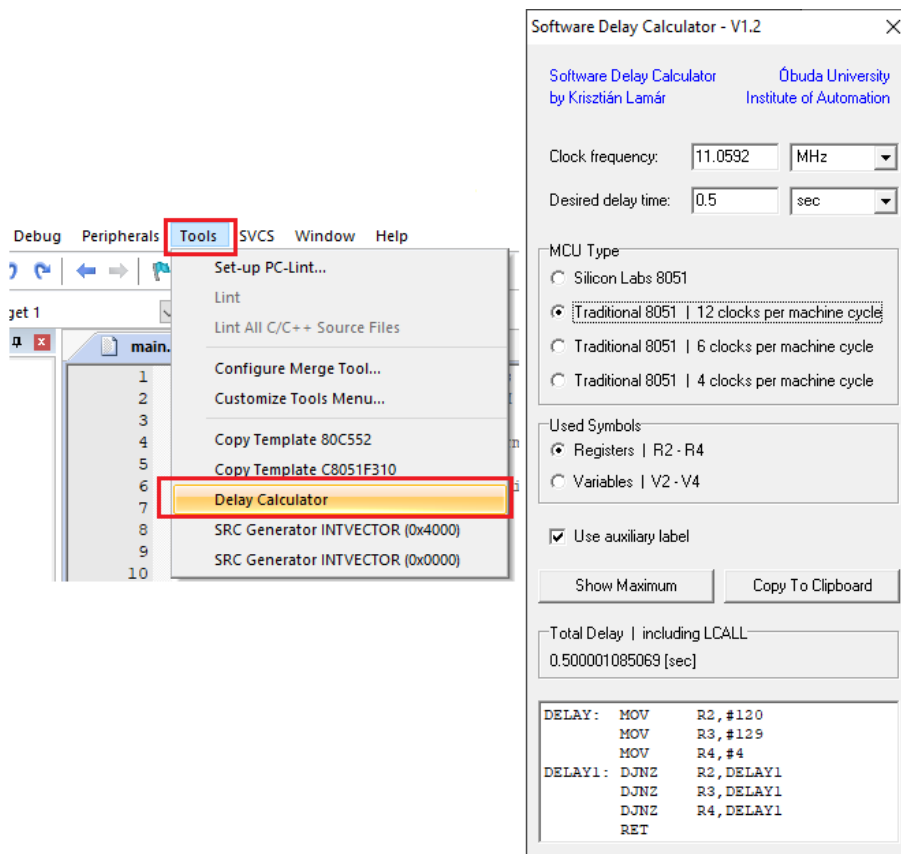
```

1      CSEG      AT      0000H
2      LJMP      0030H
3
4      CSEG      AT      0030H
5
6      MOV       SP,#2FH      ;Kell SP a szubrutinokhoz
7  CIKLUS:
8      LCALL     S_READ_BUTTONS ;olvassunk be a gombokrol
9      LCALL     S_WRITE_LEDS   ;irjunk ki a ledekre
10     LJMP      CIKLUS
11
12  END_OF_PROGRAM:
13     LJMP      END_OF_PROGRAM ;Lezaro vegtelen ciklus
14
15  ;A SZUBRUTINOK MINDIG A LEZARO CIKLUS UTAN JONNEK!
16  ;Life hack: A szubrutin cimkeknek erdemes S_ elotagot adni
17  ;mert így latszik is, hogy az egy szubrutin cimke
18  S_READ_BUTTONS:
19     MOV        DPTR,#0C000H
20     MOVX       A,@DPTR
21     CPL A
22     RET
23     ;Ki ne felejtsuk a RET utasitast a szubrutin vegen!
24  S_WRITE_LEDS:
25     MOV        DPTR,#0C001H
26     CPL A
27     MOVX       @DPTR,A
28     RET
29     ;Ki ne felejtsuk a RET utasitast a szubrutin vegen!
30     END ;Assembly fajl vege

```

FELADAT: Gépeld be a programot, fordítsd és indítsd el a szimulátort! Az XDATA memória ablak most ne X:0-tól, hanem X:0C000H-tól induljon! A **léptetés** közben írd át a gombok értékét, majd nézd meg mi történik az XDATA ablakkal! Milyen úton fut a program? Mikor, hova "ugrik"?

Nézzünk pár kézzelfoghatóbb szubrutinos példát, ami otthon is használható. **Csináljuk egy futófényt a P1 porton!** Ehhez meg kell ismerjünk a delay generátor tool használatát! Ha felraktad a Kandó C51 Toolst, akkor **navigálj a felső menüsorban a Tools → Delay Calculator gombra!** Felugrik egy ablak:



46. ábra. A delay generátor

Ezzel az eszközzel **szoftveres** késleltető szubrutinokat lehet gyártani a megadott paraméterek alapján. **Állíts be 0.5 másodperces késleltetést!** Figyelem: ki kell választani milyen mikrokontrollerre szeretnénk kódot generálni. A mi 8051-es mikrokontrollerünk 12 órajeles gépi ciklussal rendelkezik, így ezt kell kiválasztani! Illetve a tanulást segítő céllal állítsd be, hogy másodlagos címkéket és általános regisztereket használjon a szubrutin! A változós (V1,V2..) megoldással most nem foglalkozunk. Nekik még külön helyt kellene foglalni a belső memóriában.

Hogyan is csinálunk futófényt? Nézzük! Gépeld be a programot, fordítsd, és mehet is a szimuláció! Figyeld a P1 port ablakot! A target 1 beállításoknál legyen bepipálva a limit speed to real-time!

```

1      CSEG      AT      0000H
2      LJMP      0030H
3
4      CSEG      AT      0030H
5
6      MOV       SP,#2FH ;induljon a stack innen
7      MOV       P1,#01H
8 FUTOFENY:
9      MOV       A,P1
10     RL        A        ;most balra fog futni
11     MOV       P1,A
12     LCALL     DELAY     ;kesleltessuk a programot
13 ;a szimulatorban nem lesz egyenletes a kesleltetes
14 ;neha picit gyorsabb, neha lassabb. Ezt le kell nyelni sajnos
15     LJMP      FUTOFENY
16
17 END_OF_PROGRAM:
18     LJMP      END_OF_PROGRAM
19
20 ;A SZUBROUTINOK MINDIG A LEZARO CIKLUS UTAN JONNEK!
21 DELAY:
22     MOV       R2,#120
23     MOV       R3,#129
24     MOV       R4,#4
25 DELAY1:
26     DJNZ      R2,DELAY1
27     DJNZ      R3,DELAY1
28     DJNZ      R4,DELAY1
29     RET
30
31     END

```

FELADAT: Módosíts a programon! Fusson jobbra a futófény! Csinálj belőle számlálót! Lassítsd be 2 másodperces léptetésre! Gyorsítsd fel 0.1 másodperces késleltetésre! Csinálj visszafelé számláló fényt az XDA-TA memóriába ágyazott ledeken! *Hint: használd az S_WRITE_LEDS szubrutint!*

2.7.1. Már megint a stack overflow...

Mi történik akkor, ha nincs hely a stacken? Mi van, ha véletlenül elfelejtettük beírni a RET-et? Mi van, ha használtunk stack kezelő PUSH utasításokat egy szubrutinban, de nem POP-poltuk ki az adatokat a szubrutin vége előtt?

Eddig tök jól működött minden. Meghívtuk a szubrutint, elugrott a program a rutinhoz, lefutott, ami benne volt, majd visszaugrottunk oda, ahonnan hívtuk a szubrutint. Sajnos rosszul is vissza lehet térni, vagy nem visszatérni. Mindegyikre nézünk egy-egy példát **elrettentő** jelleggel.

Előadáson azt tanultad, hogy szubrutin hívásakor a programszámláló (PC) aktuális értéke elmentődik a stackre, és a hívni kívánt szubrutin címére ugrunk el. Aztán, ha lefutott a szubrutin, akkor a RET hatására visszaállítjuk a programszámlálót az LCALL utáni utasítás címére. **Hány bites a PC? 16! Hány bites az IDATA memória? 8! $2 \cdot 8 = 16$. Tehát az LCALL során 2 bájtot (PCH, PCL: a programszámláló felső és alsó bájtjai) rakunk bele a verembe, RET esetén pedig 2 bájtot veszünk ki belőle. Az SP is változik közben! Mit ronthatunk el?**

Ha nincs hely a veremben még 2 bájtot PUSH-olni (LCALL), akkor honnan tudjuk, hova kell visszatérnünk? Hát sehonnan. Ha a szubrutinban használtam pl. PUSH ACC utasítást, de a POP ACC párját már elfelejtettem, akkor hova térek vissza? Gőzöm sincs. Valahova.

Nézzük, miket NE csináljunk, ha assemblyben programozunk. Mind-egyik hibás kódot írd be, és nézd meg mi történik, ha lépteted a programot. **Az error és IDATA memory ablakokat, illetve a PC és SP értékét kell nézni!**

Van PUSH, de nincs POP a RET előtt:

```
1      CSEG      AT      0000H
2      LJMP      0030H
3      CSEG      AT      0030H
4
5      MOV       SP ,#2FH
6      MOV       A ,#0
7      MOV       P1 ,#0
8      LCALL     S_TEST_ROUTINE
9      MOV       P1 ,#1
10     CIKLUS :
11         MOV    A ,P1
12         INC    A
13         MOV    P1 ,A
14         LJMP   CIKLUS
15
16     END_OF_PROGRAM :
17         LJMP   END_OF_PROGRAM
18
19     S_TEST_ROUTINE :
20         PUSH   ACC
21         PUSH   P1
22         ;ACC es P1 megy a verembe
23         NOP
24         NOP
25         NOP
26         ;Hoppa. elfelejtettem a 2 POP utasitast
27         RET
28         END
```

Hova fogunk visszatérni a RET utasítással? Pontosan arra a címre, amit a stackből kivehető első 2 bájt értéke ad. Most ez 00 és 00, azaz a RET hatására a PC nem a jó címre, hanem 0000H-ra fog visszaugrani. Ezért még a μ Vision sem szól, hiszen a 0000H címen van ám utasítás, a kezdő ugrás 0030H-ra.

Írd át a programot, hogy az akkumulátor értéke 00H, a P1 értéke pedig 40H legyen! Így hova tér vissza a szubrutin a RET hatására? Hogyan lehetne megjavítani a programot? Mi történik a veremmel? Marad benne elem, ha nincs POP? Írd le a jegyzőkönyvbe, magyarázattal együtt!

Nincs RET:

```
1      CSEG      AT      0000H
2      LJMP      0030H
3
4      CSEG      AT      0030H
5
6      MOV       SP ,#2FH
7      MOV       A ,#0
8      MOV       P1 ,#0
9      LCALL     S_TEST_ROUTINE
10     MOV       P1 ,#1
11
12     CIKLUS :
13         INC     A
14         LCALL   S_WRITE_LEDS
15         LJMP    CIKLUS
16
17
18     END_OF_PROGRAM :
19         LJMP    END_OF_PROGRAM
20
21     S_TEST_ROUTINE :
22         NOP
23         ;Hol van a RET?????????
24
25
26     S_WRITE_LEDS :
27         MOV     DPTR ,#0C01H
28         CPL     A
29         MOVX    @DPTR ,A
30         RET
31         END
```

Most úgy néz ki, **mintha** jól működne a program. A szubrutinból nem térünk vissza RET-tel, így a POP ACC utasítás után a programszámláló ugrás helyett halad tovább, és **végrehajtjuk a következő szubrutint is!** Szerencsére a második szubrutin végén már ott a RET, így vissza tudunk térni oda, ahonnan indultunk. **Mi történne a programszámlálóval és a programmal, ha az XDATA ledekre kiíró szubrutin sem térne vissza?**

Tele a stack az LCALL előtt:

```
1      CSEG      AT      0000H
2      LJMP      0030H
3
4      CSEG      AT      0030H
5
6      MOV       SP,#7FH ;Hat, ez tele van mar most..
7
8      INC       A
9      LCALL     S_TEST_ROUTINE
10     NOP
11     NOP
12     NOP
13
14 END_OF_PROGRAM:
15     LJMP      END_OF_PROGRAM ;Lezaro vegtelen ciklus
16
17 S_TEST_ROUTINE:
18     NOP
19     RET
20
21     END
```

Amikor tele van a stack, akkor a LCALL hatására is adatvesztés fog történni, ugyanúgy, mintha a PUSH utasítást használtuk volna. A programszámláló elmentésre váró értéke elveszik, mert az IDATA memória csak 7FH címig létezik a 8051-ben, de a Stack Pointer 0FFH-ig vígan növekszik. A RET hatására az SP most visszatér 7FH-ra, de a 81H és 80H címekről nem tud adatot kipakolni a PC-be, hiszen azokat el sem éri. Ilyenkor 00H értékeket fogunk kipakolni automatikusan, ezzel töltődik fel a PC is, azaz elkezdődik előről a program futása a visszatérés után. **Mi történne, ha az LCALL előtt a Stack Pointer értéke valami miatt 0FFH lenne? Akkor működne a program? Ha igen, akkor milyen megkötések mellett? Mi van a belső memória elején? Miket írtunk át az első mérésen a stack overflow miatt? Ha ezeket most nem bántom, akkor működni fog a program?**

Mindig legyen hely a veremben, mert a szubrutinok hívásához is kell a stack! Arra is nagyon figyeljünk, hogy ha a későbbiekben 8052-vel dolgoznánk, akkor ne csorduljon túl a stack pointer a 00H címre!

Egy érdekesség: a túl mélyen hívott szubrutinok problémája

Mi történik akkor, ha egy szubrutinban meghívok egy szubrutint, amiben meghívok egy szubrutint, amiben meghívok egy szubrutint, amiben meghívok egy szubrutint.....? Erre nagyon jó példa, ha rekurzívan szeretnék meghívni egy rutint. A rekurzivitás azt jelenti, hogy a szubrutin önmagát hívja meg, és csak egy adott feltétel esetén indul el a visszatérések lánc az első LCALL utasításhoz. Például: Inkrementáljuk rekurzív szubrutin hívással az akkumulátort, amíg eléri a 14H értéket. **Próbáld ki a programot! Tényleg 14H lesz az akkumulátor eredménye a program végén?**

```
1      CSEG      AT      0000H
2      LJMP      0030H
3      CSEG      AT      0030H
4
5      MOV       SP ,#2FH
6      MOV       A ,#0
7      LCALL     S_RECURSIVE_SUBROUTINE
8      NOP
9      NOP
10     NOP
11     NOP
12 END_OF_PROGRAM:
13     LJMP      END_OF_PROGRAM ;Lezaro vegtelen ciklus
14
15 S_RECURSIVE_SUBROUTINE:
16     CJNE      A ,#14H, INCREMENT_AGAIN
17     LJMP      DONE
18 INCREMENT_AGAIN:
19     INC       A
20     LCALL     S_RECURSIVE_SUBROUTINE
21 DONE:
22     RET
23
24     END ;Assembly fajl vege
```

Mi lenne, ha nem 14H-ig, hanem 0FFH-ig szeretnénk növelni az akkumulátort? Akkor is jól fog működni a program? *Hint: arra gondolj, hogy minden LCALL 2 bájtot foglal a stacken. Egy idő után tele lesz.*

2.8. A soros port kezelése

Fel szeretnénk venni a kommunikációt a számítógéppel, mert mindenképpen meg szeretnénk jeleníteni a képernyőn egy szöveget! Hogyan kell eljárunk? Ismerjük meg a soros port használatát és az UART működését!

2.8.1. Kommunikációs csatornák típusai

Mielőtt megismernénk a soros port használatát, tisztáznunk kell pár dolgot. A kommunikáció mindig két – vagy több – eszköz között zajlik, előre meghatározott paraméterek alapján. Mi csak két eszköz között fogjuk megvalósítani a kommunikációt. A kérdés: ki az adó, és ki a vevő?

- **Szimplex** csatorna esetén az információ csak egy irányban folyhat. Az adó és a vevő szerepe nem cserélődik fel. Erre tipikusan jó példa, amikor tévét nézünk.
- **Fél-duplex (half-duplex, semiduplex)** csatornánál már nincs fix adó és vevő. Az információ mindkét irányban folyhat, de egyszerre csak az egyik irányban. Például a walkie-talkie CB-rádió így működik.
- **Teljes-duplex (full-duplex)** csatorna esetében pedig mindkét fél adó és vevő egyszerre. Ez a típus két ellentétes irányú szimplex csatornaként is felfogható. Milyen jó példa lenne full-duplex kommunikációra? Például a telefon.

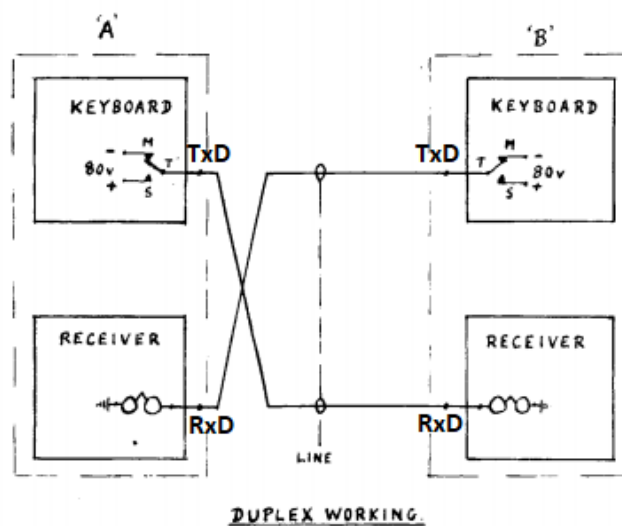
2.8.2. Aszinkron soros kommunikációs protokoll: történelem

Ahhoz, hogy értsük, hogyan működik a soros port és az UART interfész, vissza kell repülnünk az időben. Az aszinkron soros protokollnak a gyökerei nagyon régre nyúlnak vissza, amikor még **elektromechanikus** módon működtek a kommunikációs eszközök, a távírógépek (angolul: teletypewriter, rövidítve: TTY). Ezeknél az automatizált eszközöknél jelent meg az aszinkron protokoll, a karakterkódolás, a Baud ráta, a start és stop bitek száma, és még nagyon sok más terminológia is, amit a mai napig használunk. Korábban a Morse kódolást használták, de ezt úgyis ismeri mindenki.

Hogyan lehetett elküldeni New York városából San Franciscóba egyetlen 'Y' betűt egy távvezetéken, és hogyan lehetett fogadni azt? Nézd a 48. ábrát!



47. ábra. Távirógép (teletypewriter, TTY)



48. ábra. Két távirógép közti közvetlen kommunikáció: null-modem.

Amennyiben a két távírógép között az adatátvitel telefonvonalon keresztül történik, akkor az átvitel neve TELEX (teletypewriter exchange), ahol a távírógép egy MODEM segítségével tud a telefonvonalra kapcsolódni. A távírógép további rejtelseibe nem fogunk fejest ugrani, de az általa használt protokollba már igen. Kisebb módosításokkal, de ugyanezen elven működik a soros port is.

Adásszünetben a csatorna jele magas (mark) szinten volt. *De miért?* Ha a két távírógépet több száz kilométer távolság választotta el egymástól, akkor valahogyan meg kellett győződni arról, hogy a két gép között létezik-e fizikai kapcsolat a távvezetéken keresztül. Amennyiben vezetékszakadás történt volna a vonalon valahol, úgy a vevői oldalon a csatornán alacsony (space) szintet lehetett mérni.

Az információt valamilyen módon kódolni kellett, hiszen a távvezetékre csak impulzusok sorozatát lehetett elküldeni a távírógépekkel. Erre az öt bites Baudot-kód (ejtsd: Bodó-kód), később a CCIT kód szolgált. Minden karakterhez (betű, szám, speciális karakterek, a csengő) egy-egy öt bites kódot rendeltek, és ennek a kódnak megfelelően szaggatták meg a távvezetéken lévő jelt.

C H B ^h N°	S A T	ELEMENTS CODE					L T R S C A S E	F I G S C A S E
		1	2	3	4	5		
1							A	-
2							B	?
3							C	:
4							D	W R U
5							E	3
6							F	OPTIONAL
7							G	OPTIONAL
8							H	OPTIONAL
9							I	8
10							J	BELL
11							K	(
12							L)
13							M	.
14							N	,
15							O	9
16							P	0

C H B ^h N°	S A T	ELEMENTS CODE					L T R S C A S E	F I G S C A S E
		1	2	3	4	5		
17							Q	1
18							R	4
19							S	'
20							T	5
21							U	7
22							V	=
23							W	2
24							X	/
25							Y	6
26							Z	+
27							CARR. RETN	
28							LINE FEED	
29							LETTERS	
30							FIGURES	
31							SPACE	
32							ALL SPACE	

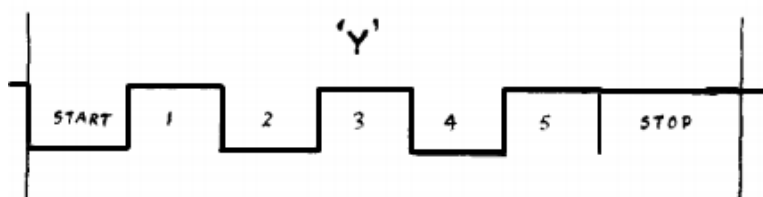
☐ SPACE ELEMENT
☒ MARK ELEMENT

C.C.I.T. N°2
CODE

49. ábra. A CCIT kódtábla. Ezt váltotta fel az ASCII

A 49. ábrán láthatod, hogy például az 'Y' betű kódja az 10101 bitsorozat volt, ahol a magas jel (mark) 1-et, az alacsony jel (space) pedig a 0-át jelenti. *De honnan tudták a gépek, hogy hol kezdődik, és hol végződik egy karakter?*

Erre a **Start és Stop** bitek szolgáltak. Mivel adásszünetben a csatorna folyamatos 1 szinten volt, így az alacsony, 0 szintű start bit jelölte egy új karakter kezdetét. A start bit után kiküldték az ötbites kódot, majd egy magas, 1 szintű stop bittel jelezték a karakter végét. Amennyiben nem küldtek új karaktert, akkor a stop bittel a csatorna visszatért az adásszüneti, magas (mark) szintre. Ha pedig nem csak egy, hanem több karaktert is küldtek, akkor a stop bit után egy újabb start bit következett, majd az ötbites, kódolt karakter, utána pedig ismét egy stop bit, és így tovább.



50. ábra. Az 'Y' karakter jelalakja a CCIT tábla szerint

A bitek küldése aszinkron módon történt, azaz az adó és a vevő egységek között nem létezett szinkronizáló jel. *Hogyan lehetett akkor hibamentesen venni az üzenetet?*

Aszinkron kommunikációnál a két félnek meg kell egyeznie egy közös gyorsaságban, amit a **Baud** (ejtsd: bód)fejez ki. A Baud Emilé Baudot-ról kapta a nevét (a Baudot kódolás feltalálója), mértékegysége pedig:

$$baud = \frac{szimbolum}{sec}$$

Jegyezd meg mindenképp, hogy a Baud **NEM** bit/sec! Speciális esetben, amikor csak két diszkrét érték között változik a jel egy csatornán, akkor a kettő egy és ugyanaz, de definíció szerint a Baud a modulált jelek továbbításának mennyiségét adja meg, 1 másodperc alatt. Amennyiben egy csatornán nem egybites (kétértékű), hanem mondjuk nyolcbites (256-értékű) modulációt használnak, akkor a baudráta nem változik, de a bitráta bit/sec értéke már igen!

Példa: soros porton adatokat küldünk 9600-as baudrátán. Tudjuk, hogy a soros porton két diszkrét érték között változik a jel, így a csatorna baudrátája 9600 szimbólum másodpercenként, az átvitel bitrátája pedig szintén 9600 bit/sec.

Ugyanakkora baudrátával küldünk adatokat, de most QAM-256 modulációval (majd Híradástechnikán lesz erről szó). A csatorna baudrátája ismét 9600 szimbólum másodpercenként, viszont a QAM-256 moduláció miatt egy szimbólum egy nyolcbites értéket jelenít meg. Azaz, a bitráta ebben az esetben már nem 9600, hanem ennek a nyolcszorosa: 76800 bit/sec!

Két szimbólum (most 0 és 1) között eltelt időt, nem meglepő módon, **szimbólumidőnek** nevezzük. A szimbólumidő a baudráta reciproka.

$$T_{symbol} = \frac{1}{baud} \quad [sec]$$

Ha mindkét fél ugyanakkora baudrátával üzemelt, akkor képesek voltak kommunikálni egymással. Az automatizált távírógépekben a belső logika figyelte a távvezeték jelszintjét, majd ha érkezett egy start bit, akkor elkezdődött a vétel.

Vétel során a csatornát mindig egy szimbólum közepénél mintavételeztek, így a két fél baudrátáinak közti kisebb-nagyobb eltérés nem okozott gondot. Amint megérkezett a stop bit is, a belső logika dekódolta a beérkezett bitsorozatot, és a megfelelő karaktert leütötte a papírra. A stop bit nem feltétlen 1, hanem legtöbbször 1.5 vagy 2 szimbólumidő volt, hiszen idő kellett a mechanikának, hogy visszatérjen az adásszüneti állapotba. A küldés is hasonlóan működött. A leütött karaktert a megfelelő bitsorozattá alakította a gép, majd megszaggatta a távvezetéken lévő jelt. Ezeknek a régi gépeknek a baudrátája nagyjából 10-20 baud volt.

A telegráfia fejlődésével szükségessé vált, hogy ne csak a nagybetűket és pár írásjelet lehessen továbbítani. Az ötbites CCIT kódolás elavult, és áttértek a 6, 7 és 8 bites kódolásokra. Ezeken felül megjelent a paritásbit is, amivel detektálni lehetett a hibásan fogadott karaktereket.

Az aszinkron soros protokoll alapelve a mai napig változatlan maradt. Csak a bitsorozat hossza és a baudráta nagysága növekedett meg, valamint új, bővített kódtáblákat használunk. Illetve a legfontosabb: ma már minden elektronikusan történik. Nincs motor a baudráta generáláshoz, nem kell elektromágnes a jel detektálásához, hanem mindent tranzisztorok végeznek egy integrált áramkörön belül.

Nem minden soros interfész képes az összes bithosszúságot kezelni, és a baudráta is csak bizonyos határok között változtatható. Attól, hogy két eszköz aszinkron soros kommunikációt folytat, még nem garantálja, hogy

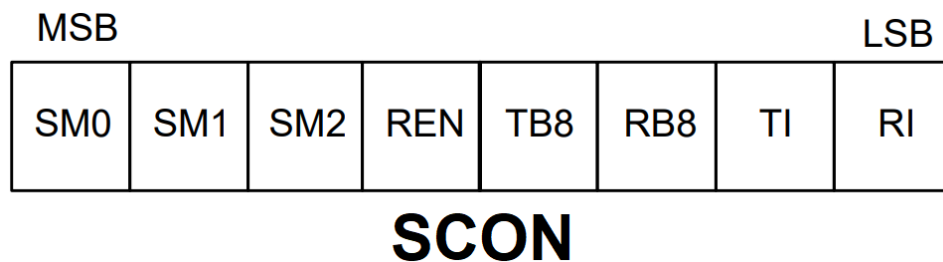
kompatibilisek lennének egymással! A bithossznak és a baudrátának meg kell egyeznie. Pontosan úgy, mint az emberi beszédnél. Két fél csak akkor érti meg egymást, ha ugyanazt a nyelvet beszélik (bithossz), mindketten értenek a témához (kódolási technika), és csak olyan gyorsan kommunikálnak, amit a másik is képes követni (baud).

2.8.3. Az UART interfész

Az UART (Universal Asynchronous Receiver/Transmitter : Univerzális aszinkron adó-vevő) interfész egy hardverelem, mely a 8051-es mikrokontroller része. Ez az interfész felelős a soros port implementálásért, és az aszinkron, soros kommunikációs protokoll kezeléséért. Az UART attól univerzális, hogy változtatható baudrátával és bithosszal is képes üzemelni, amit mi, a programozók állíthatunk be. Az UART interfész full-duplex, azaz egymástól függetlenül, egyszerre fogadni, és küldeni is képes. A 8051-es mikrokontroller UART interfésze 8 vagy 9 bit széles adatokat képes kezelni (10 vagy 11 bit, ha a start és stop biteket is számítjuk).

Az UART egyik feladata, hogy egy nyolcbites regiszterben lévő párhuzamos adatot bitsorozattá alakítson, és elküldje az üzenetet a TxD porton. A másik feladata pedig, hogy az RxD porton érkező bitsorozatot párhuzamosítsa, és eltárolja azt egy nyolcbites regiszterben. Ezt a két feladatot egyszerre is el kell tudnia látni, ettől lesz full-duplex az interfész.

Nekünk, programozóknak csak az interfész paramétereinek beállításával kell foglalkozni, hiszen a protokoll szerinti kommunikációt már automatikusan, önmagától képes elvégezni az UART modul. Nézzük meg, hogyan állíthatjuk be az UART interfészt az **SCON (Serial CONTROL)** SFR segítségével!



51. ábra. Az SCON SFR bitjei

Amint láthatod, az SCON regiszter mindegyik bitje fontos. Hasonlóan a PSW regiszter RS1 és RS0 bitjeihez, az SCON bitjeivel is speciális funkciókat lehet vezérelni, vagy megfigyelni. Most nem a regiszter bankok közti váltást, hanem az UART modul működését fogjuk beállítani! Nézzük, mire jók ezek a bitek! Ja, az SCON regisztert bitenként is lehet címezni!

Haladjunk az MSB (Most Significant Bit, legnagyobb helyiértékű bit) felől az LSB (Least Significant Bit, legkisebb helyiértékű bit) felé!

SM0	SM1	Üzem mód	Leírás	Baud
0	0	0	8 bites USART	$\frac{f_{CLK}}{12}$
0	1	1	8 bites UART	változtatható
1	0	2	9 bites UART	$\frac{f_{CLK}}{64}$ vagy $\frac{f_{CLK}}{32}$
1	1	3	9 bites UART	változtatható

Az **SM0** és **SM1** (Serial Mode) bitekkel az UART modul üzemmódját választhatjuk ki! A leggyakoribb az 1-es (SM0 = 0, SM1 = 1) üzemmód. Ebben az üzemmódban tudunk kommunikálni egy számítógép soros port-jával, ezt fogjuk használni. A 2-es és 3-as üzemmódokat ritkábban, a 0-ás módot pedig gyakorlatilag soha nem használjuk.

A baudráta az 1-es és 3-as üzemmódokban változtatható. Ilyenkor a mikrokontroller belső, T1-es időzítő egységét felhasználva lehet beállítani a kommunikáció sebességét. A következő mérésben majd lesz szó bővebben az időzítő működéséről, most csak alkalmazni fogjuk őket, egy inicializáló szubrutin segítségével.

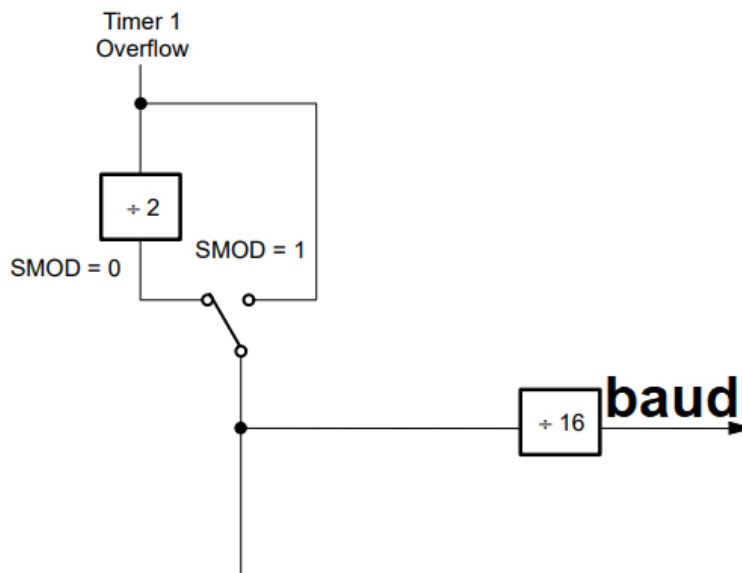
- **SM2** Ez a bit felelős a multiprocesszoros kommunikáció engedélyezéséért. Nem fogjuk használni az SM2 bitet.
- **REN** Ezzel a bittel engedélyezhetjük, hogy az UART modul fogadni is tudjon adatot. Ha REN = 0, akkor csak küldeni tudunk.
- **TB8** 9 bites UART esetén ez a kilencedik küldeni kívánt adatbit. Ezt sem fogjuk használni.
- **RB8** 9 bites UART esetén ez a kilencedik beérkezett bit. 1-es módban, ha SM2 = 0, akkor ez a vett stop bit értéke. Digit laboron ezt sem használjuk.
- **TI** Transmit Interrupt (adó megszakítás) jelzőbit (angolul: flag). Ez a bit jelzi, ha egy teljes keretet kiküldünk. **Ezt a bitet a programból kell kitörölni, és használni is fogjuk!**
- **RI** Receive Interrupt (vevő megszakítás) jelzőbit (angolul: flag). Ez a bit jelzi, ha megérkezett egy keret a soros porton. **Ezt a bitet is a programból kell kitörölni, és használni is fogjuk!**

Mit kellene beírni az SCON regiszterbe, hogy 8 bites, változtatható baudrátás, full-duplex UART-ot kapjunk? Az SM0 és SM1 bitekkel beállítjuk a 1-es üzemmódot, a REN bittel pedig engedélyezzük az adatok fogadását. Ennyi elég is lenne, de a TI bitet is 1-be kell állítsuk, mivel egy olyan szubrutint fogunk írni a karakterek kiküldésére, aminek szüksége van arra, hogy a legelső karakter kivitele előtt a TI értéke már 1 legyen!

Ezeket a biteket külön-külön is írhatnánk, hiszen bitcímezhető az SCON, de sokkal elegánsabb, ha két utasításra szedjük szét az inicializálást. Az egyikkel beállítjuk az üzemmódot, majd egy másikkal külön beállítjuk a TI bit értékét 1-be! Azaz:

```
1 MOV     SCON, #50H ;SM1 = 1, REN = 1 (1-es mod, full-duplex)
2 SETB    TI         ;A szubrutin miatt kell
```

Az üzemmód beállítása csak fél siker, ugyanis 2-es módban a baudráta változtatható, amit a T1-es időzítő túlsordulásával lehet beállítani a kívánt értékre. **9600-as baudrátát szeretnénk beállítani**⁸. A soros interfész számára az alábbi blokkdiagram szerint lehet előállítani a kívánt baudrátát:



52. ábra. A baudráta előállítása

⁸A 9600-as baud egy szabványos érték. A legtöbb soros port alapbeállításon ilyen sebességgel üzemel.

A baudráta generálása a mikrokontroller órajeléből történik. Végeredményként azt szeretnénk, ha a 11.0592 MHz-ből valamilyen úton-módon 9600 Hz lenne.

Kezdjük az 52. ábra tetején: **Timer 1 Overflow**. Valóban, a baudráta nagyságát a T1-es időzítő túlsordulásával lehet beállítani. Az időzítők minden nyavalyájáról majd a harmadik mérésen lesz szó, most egyenlőre annyit kell tudni, hogy az időzítő modul nem más, mint egy számláló hálózat. Számlálót már biztos terveztél valamelyik házi feladatban. Ez a számláló 8, 13 vagy 16 bites lehet attól függően, hogy milyen üzemmódban használjuk. *Ezt most hidd el:* a soros portnál 2-es üzemmódban használjuk a T1-es időzítőt, ahol nyolcbites, automatikus újratöltéssel rendelkező számlálóként viselkedik. A számlálót frekvenciaosztóként használjuk.

Mit jelet a túlsordulás? Azt, amikor a számláló kimenete csupa 1-ből csupa 0-ba vált át. Ha nyolcbites számlálóról van szó, akkor ez a $0xFF \rightarrow 0x00$ átmenetet jelenti. Hány órajel periódus szükséges ehhez? Ha végigszámlálunk nullától 255-ig, akkor 256, de ha $0xFD$ -től számolunk, akkor már csak 3 órajel periódus szükséges hozzá.

Sajnos ez még csak fél igazság, ugyanis a 8051-es időzítői (számlálói) nem órajel periódusonként, hanem **gépi ciklusonként** számolnak! Azaz, egy teljes számláláshoz nullától 255-ig valójában nem 256 órajel periódus, hanem ennek a tizenkétszerese (3072) szükséges. $0xFD$ -től kezdve pedig nem 3, hanem 36! Azaz általánosságban elmondható, hogy ekkora frekvenciával fog túlsordulni a T1-es számláló, ha nyolcbites, automatikus újratöltéses üzemmódban használjuk:

$$f_{T1_{overflow}} = \frac{f_{CLK}}{12 \cdot (256 - TH1)}$$

A **TH1** nevű változó az egyenletben egy nyolcbites SFR értéke. Pontosabban a T1-es számláló felső bájtjának aktuális értéke. Na, de mivel most csak nyolcbites, automatikus újratöltéses üzemmódban használjuk, ezért a TH1 értéke lesz az újratöltési érték. Azaz, amit beleírok a TH1-be, onnantól kezd el számolni a számláló. Ha ez például $0xFD$, akkor a számlálás a $0xFD \rightarrow 0xFE \rightarrow 0xFF \rightarrow 0xFD \rightarrow 0xFE \dots$ séma szerint fog történni. Érdekes módon a $0x00$ értéket nem fogja felvenni a számláló, de attól még túl fog csordulni, csupán egyből fel is töltődik a TH1 értékével.

Ha ezt a rémséget sikerült megemésztünk, innentől kezdve már csak egyszerű osztásokat kell elvégezni, és meg is kapjuk a kívánt baud értékét! Az **SMOD** egy bit értéke, mely a PCON (Power CONtrol) SFR-ben található a hetes biten (sajnos ez az SFR nem bitcímezhető). Ha ennek a bitnek az értéke 1, akkor a baudrátát megduplázzuk. Most ezzel a lehetőséggel nem fogunk élni, azaz $SMOD = 0$! Innen pedig már csak egy egyszerű osztás 16-tal, és meg is van a baud! Hogyan is néz ki az egész egy képletben?

$$baud = \frac{f_{CLK}}{2^{(1-SMOD)} \cdot 16 \cdot 12 \cdot (256 - TH1)}$$

Nézzük a jó oldalát, legalább integrálás nincs benne. **Számoljuk ki, mennyi lesz a baud értéke, ha $TH1 = 0xFD$, $SMOD = 0$, az órajel frekvenciája pedig 11.0592 MHz!**

$$\frac{11.0592 \cdot 10^6}{2^{(1-0)} \cdot 16 \cdot 12 \cdot (256 - 253)} = 9600$$

Elképesztő, egyszerűen hi-he-tet-len. Az eredmény pont 9600! **Emiatt van szükség erre a megjegyezhetetlen 11.0592 MHz-es órajelre.** Ezzel szépen, kereken lehet előállítani a standard baudrátákat a soros port-hoz! 12 MHz-es órajellel már nem lenne szép kerek érték. Jöjjön a teljes inicializáló szubrutin! **Ezt a szubrutint használni fogjuk!**

```

1 ;A soros port inicializalasa
2 ;8 bites UART, 9600-as baud
3 S_INIT_SERIAL_9600:
4     MOV     SCON, #50H
5     ANL     TMOD, #0FH
6     ORL     TMOD, #20H      ;T1 <-- mode 2
7     MOV     TH1, #0FDH
8     SETB    TR1            ;idozito start!
9     SETB    TI              ;szubrutin miatt TI = 1
10    RET

```

Újra megjelent a TMOD nevű regiszter! Nem véletlen ezzel lett bemutatva a bit maszkolás :) Ezzel a regiszterrel lehet az időzítőket konfigurálni. A cél a maszkolással, hogy csak a felső 4 bitet változtassuk meg 2-re!

Az SCON, TMOD és TH1 beállítását szerencsére nem kell mindig önállóan megszülni, hiszen a mikrokontroller felhasználói kézikönyvében fel vannak sorolva a leggyakoribb baudrátákhoz tartozó beállítások!

Baud Rate	fosc	SMOD	Timer 1		
			C/T	Mode	Reload Value
Mode 0 Max: 1.67MHz	20MHz	X	X	X	X
Mode 2 Max: 625k	20MHz	1	X	X	X
Mode 1, 3 Max: 104.2k	20MHz	1	0	2	FFH
19.2k	11.059MHz	1	0	2	FDH
9.6k	11.059MHz	0	0	2	FDH
4.8k	11.059MHz	0	0	2	FAH
2.4k	11.059MHz	0	0	2	F4H
1.2k	11.059MHz	0	0	2	E8H
137.5	11.986MHz	0	0	2	1DH
110	6MHz	0	0	2	72H
110	12MHz	0	0	1	FE8H

Timer 1 Generated Commonly Used Baud Rates

53. ábra. Szabványos baud értékekhez tartozó beállítások

Rendben! A soros portot már megfelelően tudjuk inicializálni, most pedig nézzük meg, hogyan küldhetünk, illetve miként olvashatunk adatokat! Ismerjük meg az **SBUF (Serial BUffer)** SFR regisztert!

Ha el szeretnénk küldeni egy adatot a soros porton, akkor azt az **SBUF (címe: 99H)** regiszterbe való írással tehetjük meg!

```

1 | MOV    A, #0
2 | MOV    SBUF, A ;küldjük ki az akkumulator ertekeit!
```

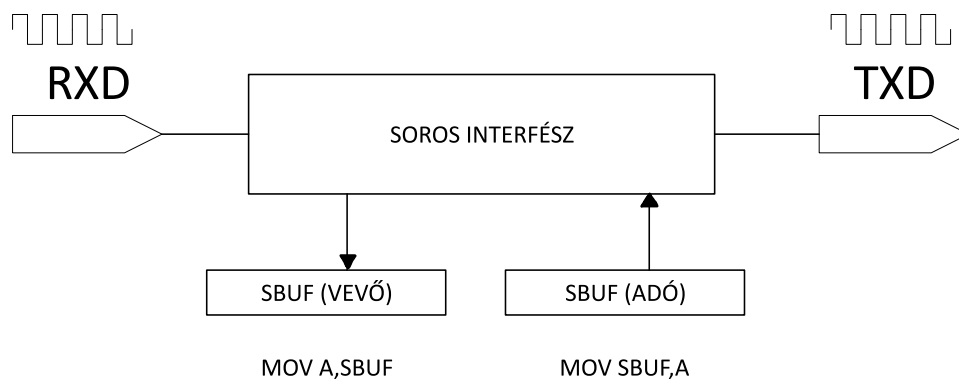
Ha pedig ki szeretnénk olvasni a beérkezett adatot, akkor azt az **SBUF (címe: 99H)** regiszterből való olvasással tehetjük meg!

```

1 | MOV    SBUF, A ;Kiolvassuk az akkumulatorba az adatot!
```

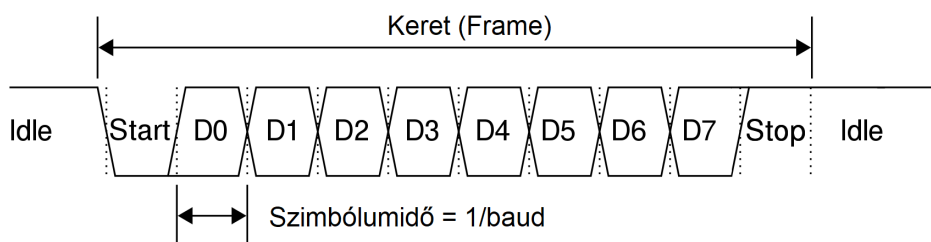
Várjunk csak! Küldéshez ÉS fogadáshoz is ugyanazt a regisztert használ-nánk? Mert a név és a cím is egyező. Hogyan lehet full-duplex az UART, ha az íráshoz és a fogadáshoz is az SBUF regisztert használjuk? Most vagy írok, vagy olvasok...

Bár mind a név, mind a cím megegyező, valójában két külön re-giszterről beszélünk! Az egyikbe csak írni tudunk (WO, Write Only), a másiból pedig csak olvasni lehet (RO, Read Only)! Az 54. ábra szemlélteti, miről is van szó.



54. ábra. Két SBUF létezik. Az egyik WO, a másik RO

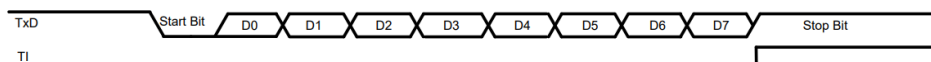
Hogyan is néz ki egy UART keret a 2-es üzemmódban? Az 55. ábrán láthatod, mi a protokoll.



55. ábra. UART jelalak, 2-es üzemmód

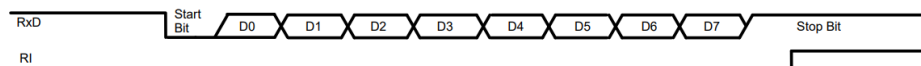
Az alapelv teljesen megegyezik az ősrégi protokollal, csupán itt már nem ötbites, hanem nyolcbites adatokkal történik az információcsere. Egy keret tartalmazza az adatbiteket, illetve az őket **keretező** START és STOP biteket is! Ez mind írásra, mind olvasásra igaz. Írás során az UART interfész a csak írási SBUF regiszter értékét egy bitsorozattá alakítja, majd kiküldi a keretezett sorozatot a TxD (P3.1) lábra. Olvasás során pedig, ha a protokoll szerinti bitsorozat érkezik az RxD (P3.0) lábon, akkor azt visszaalakítja az interfész egy nyolcbites adattá, és eltárolja az értéket a csak olvasható SBUF regiszterben!

Írás Hogyan történik az írás? Bármikor, amikor adatot mozgatunk a csak írható SBUF regiszterbe, akkor az UART modul megkezdí az adatok kivitelét a TxD (P3.1) lábon. Protokoll szerint az adatok kivitele a START bittel kezdődik, majd little endian elrendezésben elküldjük az adatot (LSB az első adat), végül pedig kiküldjük a STOP bitet. A TI bit a STOP bit kiküldésének pillanatában aktivizálódik, jelezve, hogy a teljes keretet kiküldtük. Ha új adatot szeretnénk küldeni, akkor törölnünk kell a TI bitet!



56. ábra. Írás a soros portra

Olvasás Olvasni csak akkor érdemes, ha már érkezett adat a soros porton. Ezt az RI bit jelzi, ami a beérkezett STOP bit után aktivizálódik. Ha az RI értéke 1, akkor tudjuk, hogy a csak olvasható SBUF regiszterben van valamilyen adat. Ezt a bitet is nekünk kell törölni. Ha kitörlés után valamennyi idővel ismét 1 lesz az RI értéke, akkor újabb adat érkezett.



57. ábra. Olvasás

Ha sikerült megérteni, hogyan is működik a protokoll, akkor a következő oldalon fejest ugorhatsz az UART interfész felépítésébe a 2-es üzemmódban! Ne ijedj meg, nem kell fejből tudni az egész felépítést, viszont úgy kerek a mérés, ha a teljes blokk diagramot is látod, az időfüggvényekkel együtt. Nagyon sokat lehet belőle tanulni! A bal felső sarok már ismerős lesz! A kapuk nem az IEC, hanem az ANSI szabvány szerinti szimbólumokkal lettek jelölve.

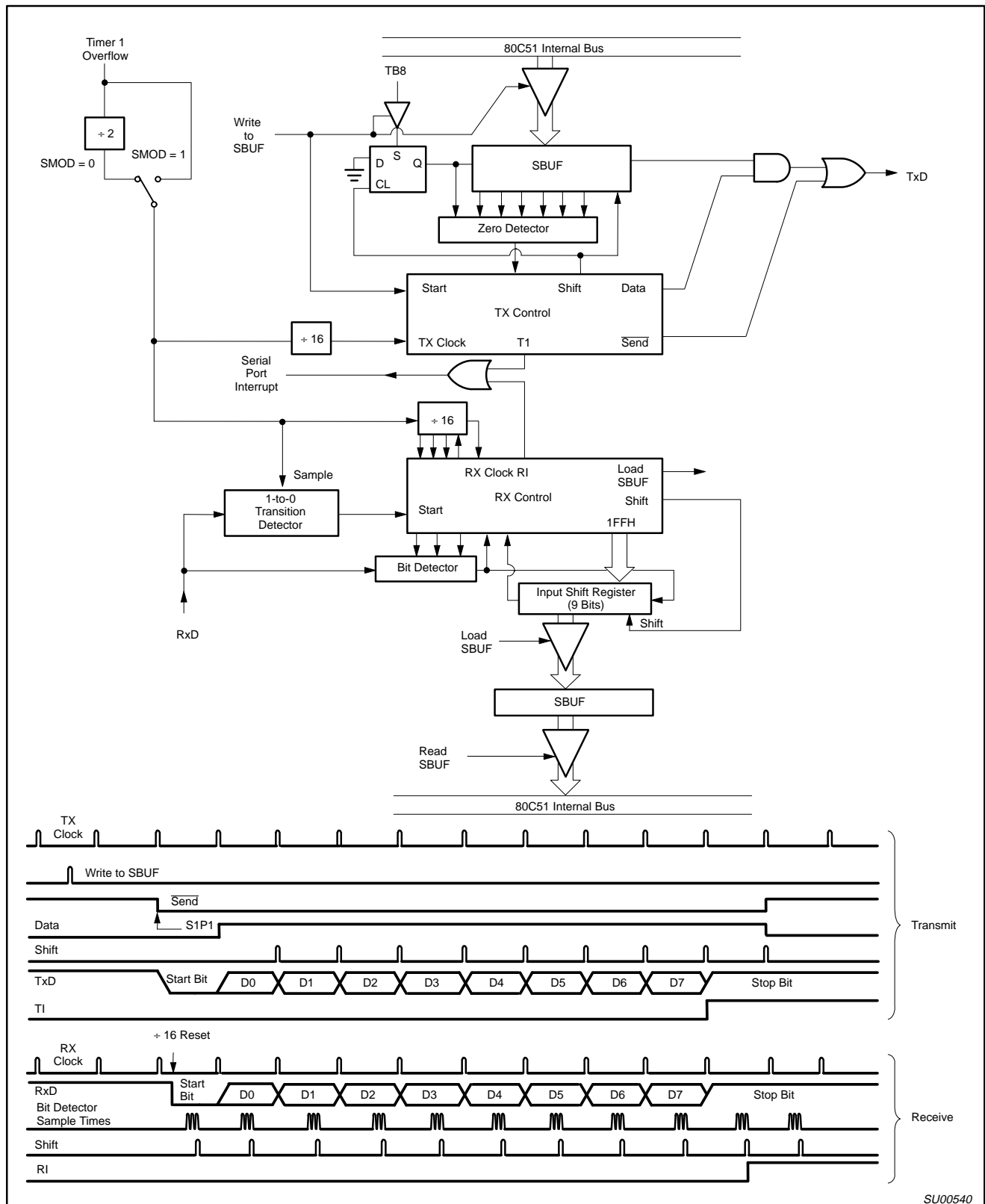


Figure 14. Serial Port Mode 1

A kisebb enciklopédia elolvasása után gyártsuk le a szubrutinokat, melyekkel egy-egy bájt olvasását és írását tudjuk kezelni! Felhasználjuk az RI és TI biteket is!

```

1 ;Ezzel a szubrutinnal lehet írni a soros portra
2 S_SERIAL_WRITE_BYTE:
3     JNB     TI,$      ;kuldhetek uj adatot?
4     CLR     TI        ;ha igen, tolrom a jelzobitet
5     MOV     SBUF,A    ;mit akarok kikuldeni?
6     RET
7 ;Ezzel a szubrutinnal lehet olvasni a soros portrol
8 S_SERIAL_READ_BYTE:
9     JNB     RI,$      ;erkezett adat?
10    CLR     RI        ;ha igen, tolrom a jelzobitet
11    MOV     A,SBUF    ;elmentem a fogadott adatot
12    RET

```

Megjelent egy új szimbólum, amivel eddig még nem találkoztunk, a \$. A dollár az adott utasítás kódmemóriában lévő címét jeleti, melyhez bizonyos megkötések mellett ofszetet is adhatunk (pl: a \$+3 az utasítás opkódja által elfoglalt bájt címénél 3-mal nagyobb címet jelenti). A JNB TI,\$ egy önmagára ugró utasítás, ha a TI értéke 0.

```

1     JNB     TI,$
2     ;Ugyanaz mint
3 CIMKE: JNB     TI,CIMKE

```

A \$ szimbólumot csak akkor használd, ha már fekete öves Assembly programozó leszel! Inkább maradj a címkék használatánál, mert az biztosan jól fog működni. Nagyon könnyen el lehet rontani a programot, ha rosszul használjuk a \$ szimbólumot. Most a feladat kedvéért direkt használtuk a dollárt, hogy láss ilyen is. Branch (elágazás) esetén az ofszet \$-128 és \$+127 között, LJMP ugrás esetén pedig megkötések nélkül állítható a 8051-es mikrokontrollernél. Sőt, a MOV DPTR,#\$ utasítással kiolvasható a programszámláló értéke is (mondjuk ez nem tényleges olvasás, mert fordítási időben értékelődik ki a \$ szimbólum).

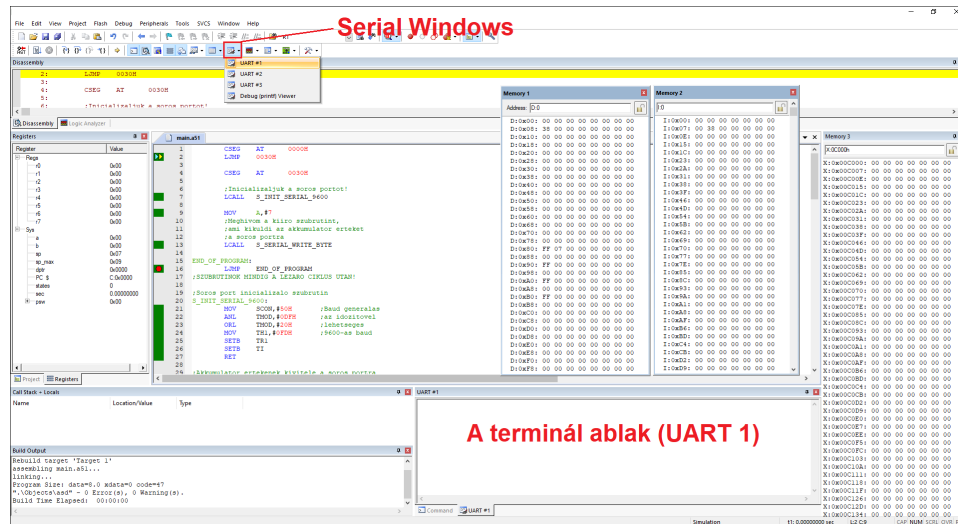
Itt az idő, hogy kiküldjünk egy karaktert a soros portra, amit a számítógép meg fog tudni jeleníteni a μ Vision termináljában! Küldjük ki a 7-es számot! Gépeld be a következő oldalon található programot, majd fordítsd le (Build), indítsd el a szimulációt (Debug), **de még ne futtasd le!**

```

1      CSEG      AT          0000H
2      LJMP      0030H
3
4      CSEG      AT          0030H
5
6      ;Inicializaljuk a soros portot!
7      LCALL     S_INIT_SERIAL_9600
8
9      MOV       A,#7          ;Mit akarok kikuldeni?
10     ;Meghivom a kiiró szubrutint,
11     ;ami kikuldi az akkumulátor értékét
12     ;a soros portra
13     LCALL     S_SERIAL_WRITE_BYTE
14
15 END_OF_PROGRAM:
16     LJMP      END_OF_PROGRAM
17 ;SZUBRUTINOK MINDIG A LEZÁRÓ CIKLUS UTÁN!
18
19 ;Soros port inicializáló szubrutin
20 S_INIT_SERIAL_9600:
21     MOV       SCON,#50H      ;Baud generalas
22     ANL       TMOD,#0DFH     ;az időzítővel
23     ORL       TMOD,#20H      ;lehetseges
24     MOV       TH1,#0FDH      ;9600-as baud
25     SETB      TR1
26     SETB      TI
27     RET
28
29 ;Akkumulátor értékének kivitele a soros portra
30 S_SERIAL_WRITE_BYTE:
31     JNB       TI,$           ;kuldhetek új adatot?
32     CLR       TI             ;ha igen, tolrom a jelzőbitet
33     MOV       SBUF,A         ;mit akarok kikuldeni?
34     RET
35
36 ;Beolvasás a soros portról az akkumulátorba
37 S_SERIAL_READ_BYTE:
38     JNB       RI,$           ;érkezett adat?
39     CLR       RI             ;ha igen, tolrom a jelzőbitet
40     MOV       A,SBUF         ;elmentem a fogadott adatot
41     RET
42
43     END

```

Hol található a terminál? Ha elindítottad a szimulátort a Debug gombbal, akkor navigálj a toolbar gombok között a Serial Windows → UART1 opcióra, majd kattints rá! Megjelent egy üres terminál ablak. Helyezd el a terminált valahova úgy, hogy jól látható legyen! Például így (a kép nagyítható):



58. ábra. A terminál ablak előhívása

Ha a terminál ablak üzemre kész, akkor **futtasd le a programot a Run gombbal (vagy F5-tel)!** Ne tegyél breakpointot sehova, majd megállítjuk a programot a piros X gombbal! Elméletileg egy 7-es számot kellene látnod a terminál ablakban. Elméletileg.

A 7-es szám kijelzése helyett, csengetett egyet a μ Vision. Ugyanazt a hangot adta ki, mint amit a fordítás után szokott, vagy amikor megjelenik a 2kB-os határt jelző ablak. Ha nem hallottad volna, akkor állítsd meg a szimulációt a Stop (piros X) gombbal, reseteld le a mikrokontrollert, majd nyomd meg ismét a Run (F5) gombot! Hallgatózz!

Mi miatt csengetett a fejlesztői környezet, mikor csak a 7-es számot szeretnénk volna megjeleníteni? **Elfelejtettük a karakterkódolást!** Ismerjük meg az ASCII kódtáblát, majd javítsuk meg a programot, hogy valóban a 7-es számot jelenítse meg!

2.8.4. ASCII? Az ki?

A karakterek kódolására többféle eljárás is létezik. Mi csak az ASCII táblával fogunk foglalkozni. Az ASCII az American Standard for Information Interchange (amerikai szabvány az információcseréhez) rövidítése. Az 1960-as években dolgozták ki annak érdekében, hogy egy egységes kódolás szerint lehessen információt cserélni. Nézzük, hogyan is néz ki az ASCII tábla!

ASCII Code Chart																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

59. ábra. Az ASCII kódtábla

Ahogy a 59. ábrán láthatod, az ASCII táblának 128 eleme van, melyek közül az első 32 karakternek, valamint az utolsónak furcsa neve van. Ezek a vezérlő karakterek, őket nem is látjuk a terminálban. A csengő (BEL) is vezérlő karakter, melynek az ASCII kódja 07H. Na, emiatt csengetett a terminál, mert ASCII kódolt karaktereket vár a mikrokontrollertől, nem natúr bináris értékeket!

Ahhoz, hogy valóban a 7-es számot lássuk a terminálon, nem 07H-val kell feltölteni az SBUF regisztert, hanem 37H-val, hiszen a 7-es számot megjelenítő karakternek az ASCII kódja a 37H! **Javítsd meg a programot! Most már tényleg a 7-es számot látod a terminálon?**

Valószínűleg furcsán néztél a második mérés címének olvasása közben, mert látszólag random hexadecimális számok voltak benne értelmes szöveg helyett. Most értelmet nyer minden! **Fejtsd vissza a 2. mérés címében található ASCII kódolt szöveget a 59. ábra segítségével! Az önálló feladatoknál szükség lesz rá! [LJMP MASODIK_MERES](#)**

Több információ az ASCII-ről: <https://hu.wikipedia.org/wiki/ASCII>

Csináljunk a 8051-ből egy loopback eszközt! Írjunk egy olyan programot, amely visszaküldi a soros porton az előzőleg fogadott karaktert! **Gé-
peld be, fordítsd, majd futtasd a Run (F5) gombbal a programot!** Futás közben kattints bele a terminál ablakba, majd nyomj le egy tetszőleges betűt a billentyűzeten! Visszaküldtük a karaktert? Figyeld meg az akkumulátor tartalmát a DATA memóriában a 0E0H címen! Milyen hexadecimális értékek kerülnek bele?

```

1      CSEG      AT      0000H
2      LJMP      0030H
3      CSEG      AT      0030H
4      ;Inicializaljuk a soros portot!
5      LCALL     S_INIT_SERIAL_9600
6 LOOPBACK:
7      LCALL     S_SERIAL_READ_BYTE    ;olvasok egy karaktert
8      LCALL     S_SERIAL_WRITE_BYTE   ;visszakuldom a terminalra
9      LJMP      LOOPBACK
10 END_OF_PROGRAM:
11      LJMP      END_OF_PROGRAM
12 ;SZUBRUTINOK MINDIG A LEZARO CIKLUS UTAN!
13 S_INIT_SERIAL_9600:
14      MOV       SCON,#50H             ;Baud generalas
15      ANL       TMOD,#0DFH           ;az idozitovel
16      ORL       TMOD,#20H             ;lehetseges
17      MOV       TH1,#0FDH            ;9600-as baud
18      SETB      TR1
19      SETB      TI
20      RET
21 ;Akkumulator ertekeinek kivitele a soros portra
22 S_SERIAL_WRITE_BYTE:
23      JNB       TI,$                 ;kuldhetek uj adatot?
24      CLR       TI                   ;ha igen, tolrom a jelzobitet
25      MOV       SBUF,A               ;mit akarok kikuldeni?
26      RET
27 ;Beolvasas a soros portrol az akkumulatorba
28 S_SERIAL_READ_BYTE:
29      JNB       RI,$                 ;erkezett adat?
30      CLR       RI                   ;ha igen, tolrom a jelzobitet
31      MOV       A,SBUF               ;elmentem a fogadott adatot
32      RET
33      END

```

Hogyan írhatunk ki egy teljes szöveget a terminálra? Vegyünk fel egy szöveget táblázatként! **Most csak a forráskód egy részletét látod!**

```

1 END_OF_PROGRAM:
2     LJMP     END_OF_PROGRAM
3 ;TABLAZATOK MINDIG A LEZARO VEGTELEN CIKLUS UTAN JONNEK!
4 TEXT1:  DB      "Kandosok vagyunk mi, f..sza gyerekek!",0

```

De jó! Nem kell egyenként beírni az ASCII kódokat, hanem elég, ha idézőjelek közé helyezzük a szöveget (számokra is működik, pl: "7")! Ezzel az assembler automatikusan átfordítja a szöveget az ASCII kódolt értékekre! **Fontos: a szöveg végére kell egy lezáró 0 érték, amivel jelezzük, hogy vége a szövegnek! Ez a lezáró nulla az ASCII kódtábla első eleme, a NUL!** A szöveges táblázatok pontosan ugyanúgy működnek, mint az igazságtábla, amit a logikai függvényes feladatnál láttunk!

Gyártani kellene egy olyan szubrutint, amely a `MOVC A,@A+DPTR` utasítással végiglépked a `TEXT1` táblázaton, majd minden elemet egyesével kiír a soros portra! Azaz, most olyan szubrutint gyártunk, ami egy másik, már létező szubrutint is meghív önmagán belül, hiszen a soros portra való kiírást már teljes mértékben kezeli az `S_SERIAL_WRITE_BYTE` szubrutin! Nézzük!

```

1 ;Ez a szubrutin kiírja a soros portra azt a szöveget
2 ;ahova a DPTR-t elmozgattam P1: TEXT1
3 ;Azaz, kell egy MOV DPTR,#TEXT1 az LCALL ele!
4 S_SERIAL_WRITE_TEXT_AT_DPTR:
5     PUSH     DPH                ;elmentem azon regisztereket
6     PUSH     DPL                ;melyeket megváltoztat
7     PUSH     ACC                ;a szubrutin
8 NEXT_CHAR:
9     CLR      A
10    MOVC     A,@A+DPTR          ;beolvasom az aktualis elemet
11    JZ       END_OF_TEXT_AT_DPTR ;ez a karakter NUL (0x00)?
12    LCALL    S_SERIAL_WRITE_BYTE ;Ezt már megírtuk egyszer!
13    INC      DPTR               ;következő karakter
14    LJMP     NEXT_CHAR
15 END_OF_TEXT_AT_DPTR:
16    POP      ACC                ;visszaállítom
17    POP      DPL                ;a regisztereket
18    POP      DPH
19    RET

```

Most pedig, írjuk meg a teljes programot, amivel meg tudjuk jeleníteni a kandós hallgatók jelmondatát! **Gépelj, fordíts, futtass! Megjelent a terminálon a szöveg?** A program nem fért ki egy oldalra sajnos, így át kell majd görgetned a következő oldalra, hogy lásd a végét.

```

1      CSEG      AT          0000H
2      LJMP      0030H
3      CSEG      AT          0030H
4      ;Inicializaljuk a soros portot!
5      LCALL     S_INIT_SERIAL_9600
6
7      MOV       DPTR,#TEXT1      ;Innen akarok kiírni szöveget
8      LCALL     S_SERIAL_WRITE_TEXT_AT_DPTR
9  END_OF_PROGRAM:
10     LJMP      END_OF_PROGRAM
11 ;TABLAZATOK, SZUBRUTINOK MINDIG A LEZARO CIKLUS UTAN JONNEK!
12 TEXT1:  DB      "Kandosok vagyunk mi, f..sza gyerekek!",0
13
14 S_INIT_SERIAL_9600:
15     MOV       SCON,#50H          ;Baud generalas
16     ANL       TMOD,#0DFH        ;az idoizitovel
17     ORL       TMOD,#20H        ;lehetseges
18     MOV       TH1,#0FDH        ;9600-as baud
19     SETB      TR1
20     SETB      TI
21     RET
22 S_SERIAL_WRITE_TEXT_AT_DPTR:
23     PUSH      DPH                ;elementem azon regisztereket
24     PUSH      DPL                ;melyeket megvaltoztat
25     PUSH      ACC                ;a szubrutin
26 NEXT_CHAR:
27     CLR       A
28     MOVC      A,@A+DPTR          ;beolvasom az aktualis
29     ;elemet
30     JZ        END_OF_TEXT_AT_DPTR ;ez a karakter NUL (0x00)?
31     LCALL     S_SERIAL_WRITE_BYTE ;Ezt mar megirtuk egyszer!
32     INC       DPTR                ;kovetkezo karakter
33     LJMP      NEXT_CHAR
34 END_OF_TEXT_AT_DPTR:
35     POP       ACC                ;visszaallitom
36     POP       DPL                ;a regisztereket
37     POP       DPH
38     RET

```

```

38
39 S_SERIAL_WRITE_BYTE:
40     JNB     TI,$      ;kuldhetek uj adatot?
41     CLR     TI        ;ha igen, tolrom a jelzobitet
42     MOV     SBUF,A    ;mit akarok kikuldeni?
43     RET
44
45 S_SERIAL_READ_BYTE:
46     JNB     RI,$      ;erkezett adat?
47     CLR     RI        ;ha igen, tolrom a jelzobitet
48     MOV     A,SBUF    ;elmentem a fogadott adatot
49     RET
50     END

```

A mérés során használt szubrutinokat nyugodtan mentsd ki valahova, hiszen univerzálisak! Az utolsó mérés során is használni fogjuk majd őket. Ha van kedved, nyugodtan írd meg 1-2 saját szubrutint is! Például: az XDATA memória bizonyos elemeit írd ki a soros portra, vagy a beolvasott adatot mentsd át az XDATA memóriába. A lehetőségek tárháza végtelen. Érdeemes megírni 1-2 hasznosnak gondolt szubrutint, mert később már elég lesz csak felhasználni őket (már amennyiben hibáktól mentesen működnek). Egész szubrutin könyvtárakat lehet előre megírni, pusztán abból a célból, hogy később egyszerűbb legyen a programozás!

2.9. Önálló feladatok

Milyen mérés is lenne az a mérés, ahol nem kellene megoldani pár feladatot önállóan? Most ismét rajtad a sor! Ha kell, puskázz, mehet a szimulátorozás, és még egy igazi kandós sláger is előkerül az egyik megoldásból! Használd nyugodtan a már megírt szubrutinokat, illetve ha kell, akkor generálj a Delay Calculator segítségével késleltető rutinokat!

- Csinálj [Knight Rider](#) féle futófényt a P1 porton! Késleltetésnek állíts be 0.25 másodpercet!
- Rajzold le a soros port TxD lábának időfüggvényét, ha 0x55, majd utána 0xAA adatot szeretnénk kiküldeni! Milyen jelalakot kell fogadni az RxD lábon ahhoz, hogy a "KANDO" szöveget kapjuk? Rajzolj időfüggvényt! A baudráta legyen 9600 szimbólum/másodperc!
- A labor során megfejtettél egy ASCII kódolt szöveget! Itt az idő, hogy felhasználj a benne rejlő kulcs értékét! A kódmemóriában lévő ASCII karaktersorozat egy YouTube videó linkét rejti [Caesar](#) kódolással! Vedd fel a kódot, mint konstans táblázat, olvasd be a karaktereket egyenként, dekódold őket a kulcs alapján, majd írd ki a visszafejtett szöveget a terminálra!

```
1 | CAESAR_CODE: DB "kwsv=22|rxwx1eh2ds<oVoYGGPf",0
```

Ha nehéz lenne kimásolni a szövegdobozból, akkor:

```
CAESAR_CODE: DB "kwsv=22|rxwx1eh2ds<oVoYGGPf",0
```

JÖJJÖN A KIHÍVÁS: Csinálj balra futó futófényt 1 másodperces késleltetéssel a P1 porton! Csinálj a P3 porton villogót 1 másodperces periódusidővel, 50%-os kitöltéssel! Ha érkezett karakter a soros porton, akkor annak az értékét írd ki a P2 portra, majd küldd vissza a számítógépnek a vett karaktert! **Ja igen, ezt a négy feladatot 1 programba kellene belesűríteni úgy, hogy mind a négyet végzi a mikrokontroller egyszerre!** Ez tényleg egy bazi nehéz feladat! Mondhatjuk úgy is, gyakorlatilag lehetetlen megcsinálni. Vagy mégse? Hajrá!

3. Harmadik mérés: Meg-sza-kí-tá-sok

A második mérés végén egy rettentő nehéz feladattal találkoztál, melyet lehetetlen helyesen megoldani megszakítások használata nélkül. Hogyan lehet egyszerre több, egymástól független feladatot végrehajtatni a mikrokontrollerrel?

Mik azok a megszakítások, mi az a vektortábla? Ma ezekre keressük a választ! Közben megismerkedünk az időzítők működésével, és megértjük, minek kellett beírni az előző két mérésen a LJMP 0030H utasítást a program elejére! Jöjjenek a megszakítások!

3.1. Miért kell nekünk a megszakítás?

Nézzük meg, miért nem lehet jól megoldani a második mérés utolsó feladatát megszakítások nélkül! Kezdjük a soros port olvasó szubrutinjával.

```
1 S_SERIAL_READ_BYTE :  
2     JNB     RI,$      ;erkezett adat?  
3     CLR     RI        ;ha igen, tolrom a jelzobitet  
4     MOV     A,SBUF    ;elmentem a fogadott adatot  
5     RET
```

Az S_SERIAL_READ_BYTE szubrutinban egy önmagára ugró utasítás szerepel, a JNB RI,\$. Ezzel az a probléma, hogy egészen addig nem fog ebből az üres ciklusból kilépni a program, amíg nem érkezett új adat a soros porton. Sajnos halvány lila gőzünk sincs arról, hogy mikor fog új adat érkezni. Lehet, hogy már azonnal tudunk olvasni, lehet hogy csak öt másodperc múlva, de az is megeshet, hogy egy hétig kell várni az új adatra. Most akkor álljon a program hetekig a JNB RI,\$ utasításnál? Biztos, hogy nem. Ennél azért tudunk jobbat is.

Tervezzük át a szubrutint! A cél az, hogy ne önmagára ugorjon a JNB RI,\$ utasítás, hanem a szubrutin végére. Ezzel a megoldással csak akkor fogjuk elvégezni az olvasást, ha már érkezett új adat (RI = 1), minden más esetben átugorjuk a szubrutin törzsét, így nem fog akár hetekig-hónapokig (vagy az örökkévalóságig) állni a program egy utasításnál. Sőt, akár már hozzáadhatjuk az írást is, hiszen azt ugyanúgy csak akkor kell elvégeznünk, ha érkezett új adat! Még a TI bitet sem kell figyelni, mivel a soros port mindkét irányban ugyanakkora sebességgel üzemel, ezért biztosan el fog telni

elég idő két SBUF írás között!

```
1 ;Teszteljük az RI bitet!  
2 S_SERIAL_LOOPBACK_WITH_POLLING:  
3     JNB     RI,NEW_DATA_NOT_AVAILABLE_IN_SBUF  
4     CLR     RI  
5     MOV     SBUF,SBUF    ;visszakuldjuk az adatot  
6 NEW_DATA_NOT_AVAILABLE_IN_SBUF:  
7     RET
```

Ahogy a módosított szubrutin nevéből is láthatod, ezt az olvasási formát **pollingnak (lekérdezésnek)** hívjuk. A lekérdezés lényege, hogy csak abban az esetben csinálunk valamit (nem feltétlenül olvasást), ha egy adott feltétel teljesül. Az S_SERIAL_READ_BYTE szubrutin nagyon hasonló ehhez. Sőt, az is lekérdezés alapú szubrutin, viszont az a polling egy speciális formáját alkalmazza, amit busy-wait pollingnak hívunk. Busy-wait polling esetén egészen addig áll a program egy üres cikluson, amíg az adott feltétel nem teljesül, míg a sima polling egy egyszerű elágazás, melynek a feltételét adott időközönként (most 0,5 másodpercenként) újra és újra teszteljük.

Rendben, most már van olyan szubrutinunk, amivel pollingos módszerrel tudjuk olvasni a soros portot, illetve egyből vissza is tudjuk küldeni az adatot. Lássuk, hogyan is nézne ki a teljes (félíg helyesen) működő program! Most nem az a cél, hogy ezt a programot egyedül begépeld, így az [alábbi GitHub repoban](#) megtalálod a programot! Másold ki a kódot, majd illeszd be a μ Vision editorjába! Ha kész, akkor fordítsd le (Build), majd a szimulátor elindítása után hívd elő a P1 és P3 portokat, illetve az UART1 terminált! **Ellenőrizd le, hogy be van-e pipálva a limit speed to real time opció a Target1 beállításainál**, majd futtasd le a programot a Run gombbal (vagy F5-tel)! Figyeld a portokat, és kezdj el írni a terminálba! Mindegy mit, csak gyorsan írd!

A program sajnos csak részben működik jól. Sőt, inkább mondjuk ki: nem működik jól. A futófény és a villogó hibátlan, viszont a soros porttal problémák vannak. Amennyiben csak minden fél másodpercben ütsz le egy karaktert, akkor az is működik rendesen, de amint gyorsabban szeretnél gépelni, már nem. Mi a hiba?

A soros porton 8N1 (start bit, 8 adatbit, nincs paritás, 1 stop bit: az UART modul 2-es módja) üzemmódban, 9600-as baudráta mellett másodpercenként maximum 960 keretet lehetne hibamentesen fogadni. Azaz, min-

den $\frac{1}{960}$ másodpercben érkezhethetne új adat, ehelyett csak fél másodpercenként tudunk új adatot fogadni a programban. A hiba a késleltető szubrutinnál keresendő.

Sajnos túl sokat (0,5 másodpercet) időzik a program a késleltető szubrutinban lévő DJNZ utasításoknál, emiatt csak 2Hz frekvenciával tudjuk lekérdezni a soros portot. Így a 9600-as baudráta által nyújtott átviteli sebességet nem tudjuk kihasználni, ami hatalmas probléma, de szerencsére van rá gyógyír! Az egyik opció, hogy kiszámoljuk mikor telik el 1 keretidő (1/960 másodperc) az utasítások között, és beszurjuk azokra a helyekre a soros portot pollingoló szubrutin LCALL hívását. **Na ne...** Ilyet egy mérnök nem csinál, inkább előveszi a mikrokontroller felhasználói kézikönyvét, és megoldja a problémát megszakításokkal! Amit nem lehet csak szoftveresen megoldani, azt megoldjuk hardveres segítséggel!

3.2. A 8051-es megszakításai

A Generic 8051-es összesen öt megszakítási forrással rendelkezik. Ezek közül egy a soros port megszakítása, kettő az időzítő modulok túlszordulásaé, az utolsó kettő pedig a P3.2 és a P3.3 lábakról érkező külső megszakítások. Megszakítás (angolul: **interrupt**) alkalmazásával egy bizonyos esemény bekövetkeztére tud reagálni a mikrokontroller **hardveresen** úgy, hogy a hardveres reakcióra mi programozók szoftverrel válaszolhassuk. Röviden, tömören: ha megszakítás érkezik egy forrástól, akkor arra egy a szubrutinhoz nagyon hasonló szerkezettel, a **megszakítási rutin** lefuttatásával tudunk reagálni. Ennek a megszakítási rutinnak az LCALL hívását pedig nem mi, hanem maga a mikrokontroller végzi **hardveresen**! Azaz, pontosan akkor történik meg a hívás, amikor az esemény bekövetkezett! Ez azt eredményezi, hogy bármit is csinált a mikrokontroller addig a pillanatig, azt félbe kell szakítsa, le kell futtatnia a megszakítási rutint, majd csak a visszatérés (RETI) után folytathatja azt, amit előtte csinált.

Te is használsz megszakításokat a mindennapjaid során. Tegyük fel, hogy éppen otthon ülsz, és valamilyen sorozatot nézel Netflixen tanulás helyett. Ez lesz a főprogramod amit futtatsz. Éppen az évad legnagyobb plot twistjénél jársz, és ebben a pillanatban valaki becsenget hozzád. Előre nem tudhattad, hogy keresni fognak, így **félbeszakítod (interruptolod)** a főprogramod, majd lefuttatod a megszakítási rutinod azzal, hogy ajtót nyitasz.

Jött a postás, meghozta a várva-várt mikrokontrolleres fejlesztői paneled. Átveszed a csomagod, elköszönsz, majd a RETI utasítással **visszatérsz** a képernyő elé, és folytatod a főprogramod futtatását ott, ahol az előbb félbe kellett szakítanod. Ez a megszakítás konyhanyelven.

Az intermezzo után nézzük, miben más egy megszakítási rutin a sima szubrutinhoz képest!

```

1 ;Soros port megszakitasi rutin
2 ISR_SERIAL_PORT:
3     CLR      TI
4     JNB      RI , ISR_NO_NEW_DATA_IN_SBUF
5     CLR      RI
6     MOV      SBUF , SBUF      ;visszakuldjuk az adatot
7 ISR_NO_NEW_DATA_IN_SBUF:
8     RETI     ;RETI kell a RET helyett!

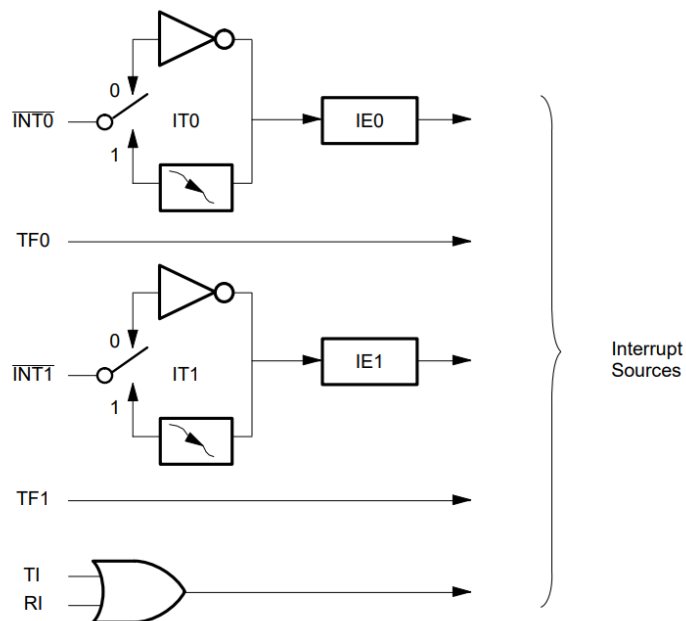
```

Első ránézésre nagyjából teljesen megegyezik az előző szubrutinnal, csupán más nevet adtunk neki. Amennyiben viszont jobban szemügyre veszed a megszakítási rutint, akkor láthatod, a RET helyett **RETI** utasítással térünk vissza. A RETI a **RETurn from Interrupt** (visszatérés megszakításból) rövidítése.

A megszakítási rutint célszerű megfelelően elnevezni. Mivel nem mezei szubrutinról van szó, ezért az **ISR** előtag használatával lehet a legjobban megkülönböztetni. Az ISR az **Interrupt Service Routine (megszakítást kiszolgáló rutin)** rövidítése. Ennek a rutinnak a lefuttatásával fogja a mikrokontroller **kiszolgálni** a soros port megszakításkérését. A kérdés most már csak az, hogyan tudjuk rávenni a mikrokontrollert, hogy meghívja ezt a rutint? Pillantsunk rá a 60. ábrára!

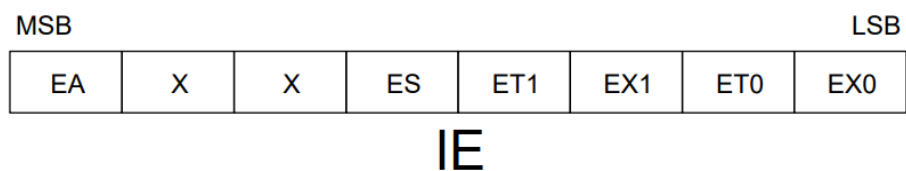
Itt láthatod, milyen forrásoktól érkezhetsz megszakítás. A felső négygel majd később foglalkozunk, de a legelső forrásnál a bemenetek (RI és TI) nevei nagyon ismerősek a soros port kezelésének megismerése után. A második mérésen azt olvashattad, hogy az RI flag (jelzőbit) a Receive Interrupt (vevő megszakítás), a TI flag pedig a Transmit Interrupt (adó megszakítás) rövidítése. **Bingó!** Tehát, a mikrokontroller ennek a két jelzőbitnek az értékét figyeli, és amennyiben bármely a kettő közül 1-es értéket vesz fel (megtörtént egy keret kivitele, vagy fogadása), akkor a soros portról megszakítási kérés érkezik, melyet feldolgozhatunk.

A második mérés során is használtuk az RI és TI jelzőbiteket, mégsem



60. ábra. Megszakítási források

történt megszakítás. Ahhoz, hogy interruptokat használhassunk, a megszakítási rendszert engedélyeznünk is kell. Ismerjük meg az **IE: Interrupt Enable (megszakítás engedélyezés)** regisztert!



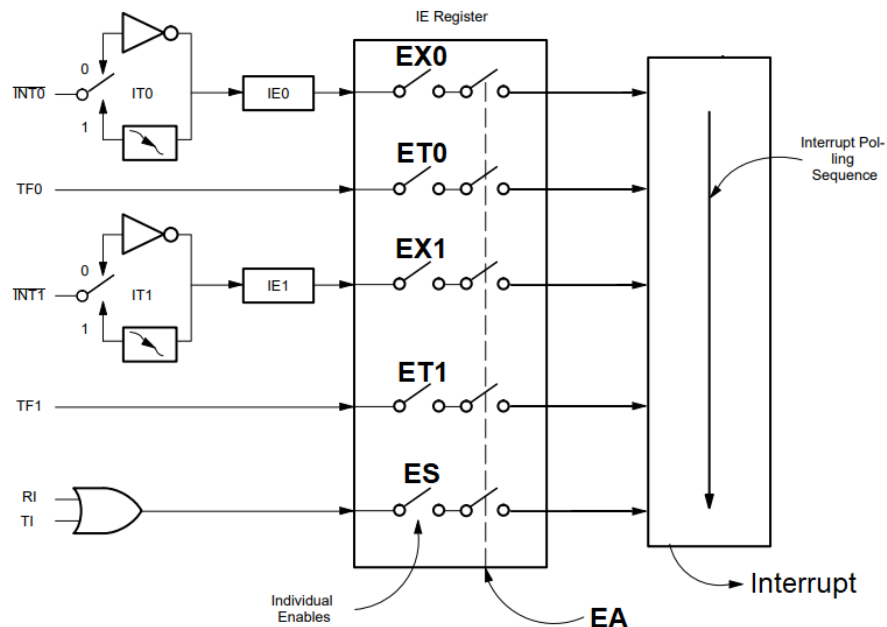
61. ábra. Interrupt Enable SFR

Az IE regiszterrel (címe: 0xA8) lehet egyenként, illetve globálisan is engedélyezni a megszakítások kérését. Nézzük, melyik bitjével mit állíthatunk!
Az IE regiszter is bitcímezhető!

- **EA, Enable All:** Amennyiben ennek a bitnek az értéke 1, úgy a megszakításokat globálisan engedélyezzük.
- **X:** Ezen biteket nem implementálták.

- **ES, Enable Serial:** Ha $ES = 1$, akkor a soros port megszakítását engedélyezzük.
- **ET1, Enable Timer 1:** Ha $ET1 = 1$, akkor a T1-es időzítő túlcsordulásával megszakítási kérés érkezik.
- **EX1, Enable External 1:** Ha $EX1 = 1$, akkor a P3.3 (INT1) lábon érkező külső megszakítás.
- **ET0, Enable Timer 0:** Ha $ET0 = 1$, akkor a T0-as időzítő túlcsordulásával megszakítási kérés érkezik.
- **EX0, Enable External 0:** Ha $EX0 = 1$, akkor a P3.2 (INT0) lábon érkező külső megszakítás.

A soros port megszakításának engedélyezéséhez csak két bitre van szükségünk: **ES** és **EA**. Az ES bittel külön engedélyezzük a soros portról érkező megszakításokat, az EA bittel pedig globálisan is engedélyezzük a megszakítási rendszert. Lássuk, hogyan is néz ki a 60. ábra kiegészítése!



62. ábra. A megszakítási rendszer egyszerűsített felépítése

A 62. ábrán azt láthatod, hogy egyenként és globálisan is engedélyezhetjük a megszakításokat, a rendszer végén pedig egy interrupt polling sequence (megszakítás lekérdező sorozat) történik. Ennek a polling rendszernek a működéséről majd később esik szó. A 8051-ben minden megszakítás **maszkolható megszakítás**. A maszkolhatóság azt jelenti, hogy egyenként eldönthetjük melyik megszakítást szeretnénk engedélyezni, és melyiket nem. Ha egy interrupt nem maszkolható, akkor nincs neki külön engedélyező bite.

Itt az idő, hogy megírjuk a karakter olvasását és visszaküldését megszakításokkal! Gépeled be az alábbi kódot, majd fordítsd le, és indítsd el a szimulációt a Run gombbal! **A soros portot inicializáló szubrutinból mindenképpen vedd ki a SETB TI utasítást, mert most azt nem fogjuk használni!** Üss le egy betűt a billentyűzeten, és nézd meg mi történik a terminál és az error (command) ablakokban!

```

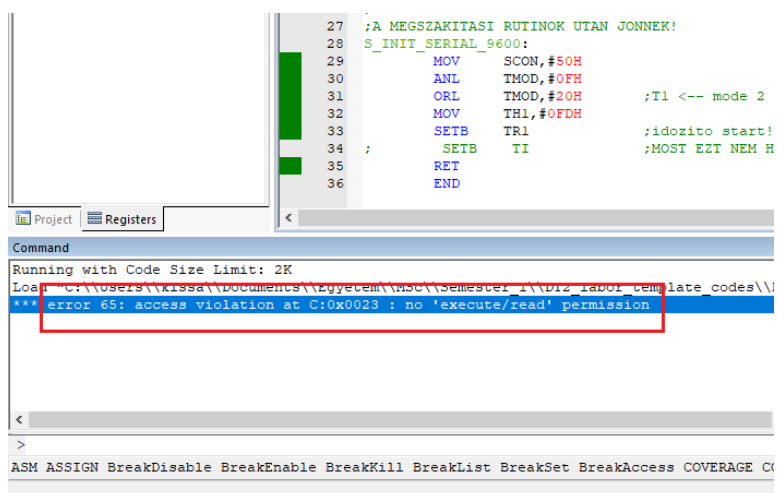
1      CSEG      AT      0
2      LJMP      0030H
3
4      CSEG      AT      0030H
5      LCALL     S_INIT_SERIAL_9600
6
7      SETB      ES      ;soros port megszakitas
8      SETB      EA      ;megszakitasi rendszer aktivalasa
9  END_OF_PROGRAM:
10     LJMP      END_OF_PROGRAM
11
12 ;MEGSZAKITASI RUTINOK MINDIG
13 ;A LEZARO CIKLUS UTAN JONNEK
14
15 ;Soros port megszakitasi rutin
16 ISR_SERIAL_PORT:
17     CLR       TI
18     JNB       RI, ISR_NO_NEW_DATA_IN_SBUF
19     CLR       RI
20     MOV       SBUF, SBUF ;visszakuldjuk az adatot
21 ISR_NO_NEW_DATA_IN_SBUF:
22     RETI      ;RETI kell a RET helyett!
23
24
25 ;SZUBRUTINOK MINDIG
26 ;A MEGSZAKITASI RUTINOK UTAN JONNEK!
```

```

27 S_INIT_SERIAL_9600:
28     MOV     SCON, #50H
29     ANL     TMOD, #0FH
30     ORL     TMOD, #20H      ;T1 <-- mode 2
31     MOV     TH1, #0FDH
32     SETB    TR1             ;idozito start!
33 ;     SETB    TI             ;MOST EZT NEM HASZNALJUK
34     RET
35     END

```

Ha minden igaz, akkor egyáltalán nem küldtük vissza azt a karaktert, amit leütöttünk a billentyűzeten, pedig a program elindult, a soros portot inicializáltuk, a megszakítási rutint is megírtuk, és még a megszakításokat is engedélyeztük. A kijelzés helyett megállt a program szimulációja, és egy error-t is kaptunk: **error 65: access violation at C:0x0023 : no 'execute/read' permission**. Gondolhatnánk, biztosan csak az a baj, hogy megállt a program a lezáró végtelen ciklusnál, de nem itt keresendő a probléma. Azt mondtuk, hogy megszakítás alkalmával **bármit** is csinált a program, azt félbe kell szakítsa, végre kell hajtsa a forráshoz tartozó megszakítási rutint, majd utána visszatérhet az előző feladatához. A végtelen ciklus is egy feladat.



63. ábra. A PC mintha elromlott volna

A μ Vision azt írja, hogy egy olyan címről szeretnénk olvasni a kódmemóriából, ahol nincs hozzáférési engedélyünk (nem írtunk utasítást). Ez a cím a 0023H. Vizsgáljuk csak meg a program legelejét!

```

1      CSEG      AT      0
2      LJMP      0030H
3
4      ;"az elso 48 bajtot (30H) atugorjuk, mert
5      ;IDE JON MAJD VALAMI A HARMADIK MERESSEN"
6
7      CSEG      AT      0030H
8      LCALL     S_INIT_SERIAL_9600

```

A 0023H cím pont azon a területen van, amit átugrunk a kezdő LJMP 0030H utasítással. Mitől ugrott el ide a programszámláló? Ismerjük meg a megszakítási vektortáblát!

3.3. A 8051-es megszakítási vektortáblája

Sajnos a mikrokontrollernek göze sincs arról, hogy milyen névvel illettük meg a soros port megszakítási rutinját, és azt sem tudja, hogy milyen címen kezdődik ez a rutin a kódmemóriában. Pontosan hova is kellene címeznie a hardveres LCALL utasítást, hogy lefuttassa az ISR_SERIAL_PORT nevű rutint? Ezt a problémát úgy oldja meg a mikrokontroller, hogy minden megszakítási forráshoz egy **fix címet (vektort)** rendel hozzá. A vektorok segítségével mindig ugyanarra a címre fog elugrani a mikrokontroller, így mi programozók pontosan tudjuk, hova kell beszúrni egy LJMP utasítást, amivel elugorhatunk a futtatni kívánt megszakítási rutinhoz!

Mit kellene tennünk, hogy működjön a program? Ha a 0023H címre beírunk egy LJMP ISR_SERIAL_PORT utasítást, akkor szoftverből megoldjuk, hogy a hardveresen generált fix címről elugorjunk a tényleges rutinhoz! Ezt a CSEG AT direktívával tehetjük meg.

```

1      CSEG      AT      0
2      LJMP      0030H
3      ;Soros port megszakitasi vektora
4      CSEG      AT      0023H
5      LJMP      ISR_SERIAL_PORT ;elugrunk a rutinhhoz
6
7      CSEG      AT      0030H
8      LCALL     S_INIT_SERIAL_9600

```

Egészítsd ki a kódot a soros port megszakítási vektorának kezelésével, majd fordítsd le újra a programot, és futtasd a Run gombbal (vagy F5-tel)! Kezdj el gépelni a terminál ablakba, most már mindennek jól kell működnie!

Valószínűleg így sem fogsz tudni 960 karaktert elküldeni egy másodperc alatt, mivel a szimulátor nem tud ennyire gyorsan dolgozni. Egyszerre kell szimulálnia a soros portot, a párhuzamos portokat, a memóriákat, az órajel-let, a regisztereket és mindent mást is. Ha valós mikrokontrolleren történne a laborgyakorlat, akkor ez a probléma nem állna fent.

Figyelem: LJMP utasítást használunk LCALL helyett! Ezt azért így csináljunk, mert a mikrokontroller hardveresen indított egy LCALL utasítást a 0023H címre, azaz a megszakítási rutinunk valójában erről a címről indul! Az LJMP utasítással csupán elugrunk oda, ahol leírtuk milyen utasításokat kell végrehajtani. A RETI hatására pedig nem 0023H-ra ugrunk vissza, hanem oda, ahonnan a mikrokontroller hívta a megszakítási vektort! **Attól, hogy nem mi indítottuk a LCALL utasítást, ugyanúgy szükséges a verem (stack) megfelelő működése! A megszakítások során is elmentődik a PC értéke a stackre, és a RETI hatására állítódik vissza oda, ahonnan a mikrokontroller hívta a vektort!**

Ahogy korábban olvashattad, nem csak a soros porthoz tartozik megszakítási forrás. Ezen kívül még négy másik interrupt is felhasználható! Milyen címeken találhatóak a többi megszakítási forráshoz tartozó vektorok?

Megszakítási forrás neve	Vektor címe
IE0 (External 0)	0003H
TF0 (Timer 0 overflow)	000BH
IE1 (External 1)	0013H
TF1 (Timer 1 overflow)	001BH
RI+TI (Serial port)	0023H

A táblázatból láthatod, hogy minden egyes forráshoz 1-1 külön cím tartozik. Azaz, ha a soros portról érkezik megszakítás, akkor a 0023H címre, ha pedig mondjuk a T0-ás időzítőtől (TF0) érkezik megszakítás, akkor a 000BH címre indít a mikrokontroller hardveres LCALL utasítást. Emiatt a vektortábla miatt kellett átugranunk az első 48 bájtot a kódmemóriában. **Ezen vektorok címét nem lehet megváltoztatni, mindig ezekre a címekre fog a mikrokontroller LCALL-t indítani hardveresen.**

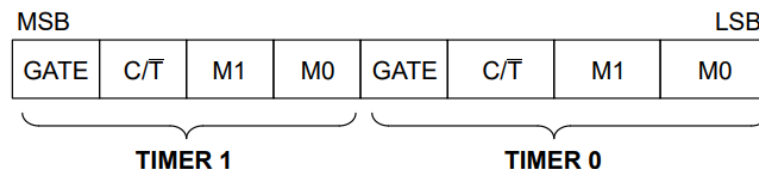
3.4. Időzítés megszakításokkal

Eddig késleltető szubrutinok hívásával oldottuk meg az időzítéseket. Ez is egy megoldás, de az időzítőket és a megszakításokat felhasználva, sokkal egyszerűbben lehet **pontos időalapot** beállítani. Nem utolsó sorban, a megszakításokkal megvalósított időzítések felszabadítják a főprogramot, így több feladatot is el tud látni a mikrokontroller addig, amíg le nem telik az időzítés.

Egy példa a megszakításos időzítés vs. szoftveres késleltetésre: Amikor berakod a pizzát a sütőbe, akkor beállítasz egy időzítést. Ha szoftveres késleltetést használnál, akkor egészen addig nem vennéd le a szemed a sütőben sülő pizzáról, amíg az készen nem lesz. Érthető, hogy ez a hozzáállás rettentően pazarló az erőforrásaiddal, ugyanis mosni és tanulni is szeretnél. Amennyiben a megszakításos időzítést választod, akkor egészen addig amíg a sütő nem jelez, felszabadulnak az erőforrásaid, és futtathatod a tanulás és mosás programokat is. Sőt, a mosás programot is egy időzítővel fogod megoldani, mert több megszakítást is tudsz kezelni. Így a főprogramod már csak a tanulás lesz, a másik két feladatot egy-egy időzítőhöz kötött megszakítással fogod elvégezni.

A hardveres időzítésekre a 8051-es mikrokontroller két időzítő modult (T0 és T1) biztosít, melyek viselkedése külön-külön programozható a TMOD (Timer MODE) és TCON (Timer CONtrol) regiszterek segítségével.

Ahogy a 64. ábrán láthatod, a TMOD regiszter felső 4 bitje a T1-es, az alsó 4 bitje pedig a T0-as időzítő viselkedését állítja be. Most csak a T0-as időzítővel fogunk foglalkozni, hiszen a T1-es időzítőt már baudráta generátorként használjuk a soros portnál. Érdeemes megjegyezni, hogy vannak olyan 8051-es változatok, melyben a baudráta generálása nem csak a T1-es számlálóval, hanem más egységekkel is lehetséges.



64. ábra. TMOD regiszter

Timer 0:

- *GATE*: kapuzó bit, mely ha 1-es értékű, akkor csak abban az esetben számlál a T0-ás számláló, ha az $\overline{INT0}$ (P3.2) láb H szintű. Ezt a funkciót nem használjuk a Digit 2 labor során.
- *C/ \overline{T}* : Counter/ \overline{Timer} üzemmód. Ha a bit értéke 1, akkor nem gépi ciklusonként, hanem a P3.4 lábról érkező felfutóélekkel léptethető a számláló.
- *M1M0*: üzemmód kiválasztó bitek. A T0-ás időzítő/számláló 4 különböző üzemmóddal rendelkezik, melyek az alábbiak: Digit labor során csak az 1-es és 2-es módot használjuk.
 - **00**: Ilyenkor a T0-ás időzítő 8 bites üzemmódban viselkedik, előosztással. Ezt a módot gyakorlatilag soha nem használjuk. A régebbi, 4048-as mikrokontroller időzítőjét valósítja meg, öt bites előosztóval.
 - **01**: A T0-ás időzítő 16 bites üzemmódban működik.
 - **10**: Nyolcbites, automatikus újratöltéses üzemmód.
 - **11**: Osztott, két nyolcbites számláló. Az alsó 4 bit a TF0 megszakításhoz, a felső 4 a TF1 megszakításhoz tartozik.

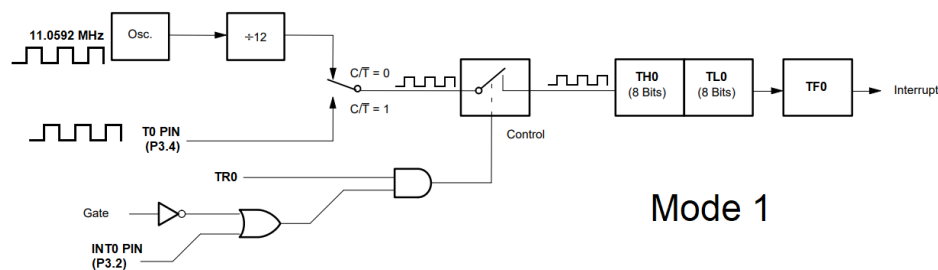
A 65. ábrán a TCON regiszter bitjeit láthatod. Csak a felső 4 bit tartozik az időzítőkhöz. Az alsó négyen az IE0, IE1, IT0, IT1 biteket már láthattad a 62. ábrán. Ők a külső megszakításokhoz tartoznak, de velük Digit 2 során nem fogunk foglalkozni.

A TF1 és TF0 bitek is 1-1 megszakítási vektorhoz tartoznak. Ezek a bitek fogják az egyes számlálók túlsordulását jelezni. A TR0 illetve TR1 bitek pedig a Timer Run bitek, melyekkel elindítható, vagy megállítható az adott időzítő modul.

MSB				LSB			
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

65. ábra. TCON regiszter

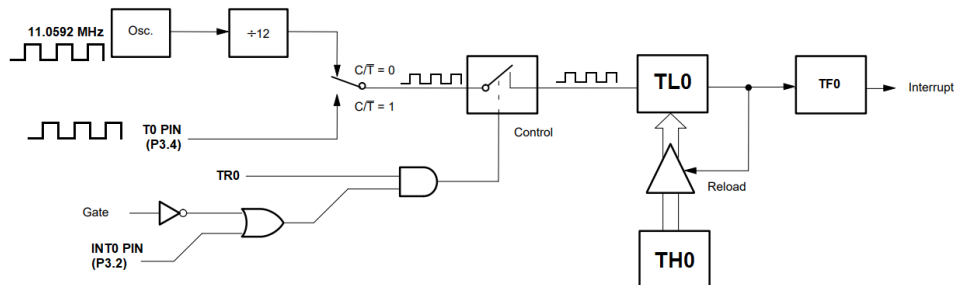
Egyes üzemmód esetén $M1:M0 = 01$. Ilyenkor a T0-ás modul 16 bites számlálóként viselkedik. A számláló felépítését a 66. nagyítható ábrán láthatod.



Kezdjük a bal felső sarokból a működést! Az oszcillátor által szolgáltatott órajel tizenkettedével (azaz a gépi ciklusidőnek megfelelő frekvenciával) vagy a T0 Pinről érkező jellel lehet léptetni a számlálót. A C/T bittel lehet kiválasztani, hogy számlálóként vagy időzítőként szeretnénk a T0-át használni. Amennyiben az oszcillátort választjuk, úgy időzítő lesz belőle, hiszen pontosan tudjuk mekkora az órajel frekvenciája. A Control kapu zárt állapotában megkezdődik a számlálás. A számláló/időzítő aktuális értékét a TH0 és TL0 regiszterekben találjuk. A TH0 (Timer High 0) a felső 8, míg a TL0 az alsó 8 bitet tartalmazza. A TF0 jelzőbit értéke 1 lesz, amennyiben túlcsordulás ($0xFFFF \rightarrow 0x0000$ átmenet) történik. Ha az interrupt rendszerben engedélyezzük az EA és ET0 biteket, akkor a TF0 flag megszakítást kér a mikrokontrollertől. A Control kapu vezérlésénél amennyiben a GATE értéke 0, úgy a TR0 bittel indíthatjuk, illetve állíthatjuk meg az időzítőt.

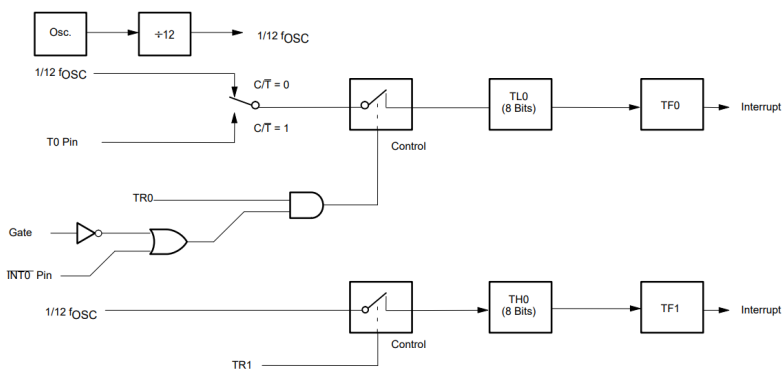
114

időzítővel, akkor ezt az üzemmódot használtuk.



67. ábra. T0 2-es üzemmódja

Hármas üzemmódban $M1:M0 = 11$. Ennél az üzemmódnál jönnek az érdekességek igazán, melyet a 68. ábrán láthatsz. A T0-as időzítő két, egyenként nyolcbites részre oszlik. A TH0 és a TL0 is 1-1 külön számláló ilyenkor. A TL0 működése már ismerős lesz, viszont a TH0 ilyenkor nagyon máshogyan működik. Őt a TR1 bittel tudjuk elindítani, és nincs is más vezérlése. Ráadásul a TH0 egy másik jelzőbitet, a TF1-et fogja 1-be állítani, ha túlcsoordul. Ez az üzemmód akkor hasznos, ha két külön számlálóra lenne szükségünk és az UART-ot is használjuk. Ilyenkor a T1-et nem tudjuk külön felhasználni, hiszen azzal a baudrátát generáljuk, viszont a T0 osztott (3-as) üzemmódjával két teljes értékű számlálót kaphatunk, két megszakítási forrással.

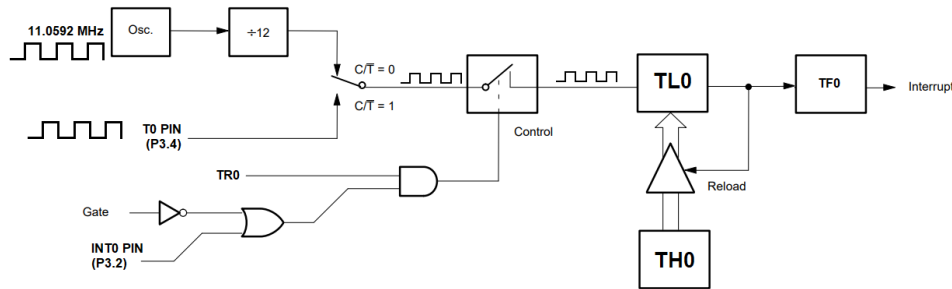


68. ábra. T0 3-as üzemmódja

A labor során részletesen csak az 1-es és 2-es üzemmódokkal fogunk foglalkozni, de aki éhezik a kihívásra, annak lesz feladat a 3-as üzemmóddal is! A következőekben megnézzük, hogyan lehet egy villogót csinálni megszakításokkal úgy, ahogyan a második labor utolsó feladata kérte.

3.4.1. 5ms-os Időalap generálása

Az állandó időközönkénti megszakításhoz egy megfelelő időalapot kell generálnunk, ami sajnos nem egyszerű feladat 11.0592 MHz-es oszcillátor esetén. Míg a baudrátánál nagyon kényelmes volt ez az oszcillátor frekvencia, most szenvedni fogunk kicsit. Nézzük meg, milyen határok között állítható a T0-ás időzítő túlcsordulásának frekvenciája 2-es üzemmódban!



$$f_{INTERRUPT} = \frac{f_{CLK}}{12 \cdot (256 - TH0)}$$

Ez a képlet már ismerős lehet a második laborról. Ott ugyanezt használtuk a T1-es időzítővel, csak még nem tudtuk hogyan is működnek a számlálók.

A legnagyobb frekvencia akkor érhető el, ha $TH0 = 255$. Ilyenkor minden gépi ciklusban túlcsordul a számláló, ami 921600 Hz frekvenciát jelent. Ilyen hatalmas megszakítási frekvenciákkal biztosan nem fogunk dolgozni, hiszen mire elvégeznénk a megszakítási rutin lefuttatását, már régen a következő megszakításnál kellene tartanunk. Ha belegondolsz, csak a RETI utasítás 2 gépi ciklust igényel, és még a vektor címén lévő LJMP-ről nem is beszéltünk. Általánosságban elmondható, hogy egy megszakítási rutin futási ideje nem lehet hosszabb vagy megegyező a megszakítások frekvenciájával, hiszen ilyen esetben értelmét veszti a megszakítás.

A legalacsonyabb frekvencia egyértelműen akkor érhető el, ha nem is használjuk az újratöltési értéket, azaz **szabadon futó** számlálóként tekintünk a T0-ra. Ilyenkor $TH0 = 0$. Ekkor a megszakítási frekvencia 3600 Hz.

A két szélsőérték között sajnos csak elvétve találunk olyan értékeket, melyekhez a túlsordulás frekvenciája egész számot ad eredményül. 11.0592 MHz-es oszcillátor mellett nem lehet szép értékeket generálni akármilyen TH0 értékre. Kiszámoltam az első 14 olyan TH0 értéket, melyhez egész számot kapunk, így nem Neked kell ezt kiszzenvedni most. Ezekből az értékekből kellene valahogyan 5 ms-ot, azaz 200 Hz-et csinálni.

TH0	0	16	31	56	64	76	96
f (Hz)	3600	3840	4096	4608	4800	5120	5766

TH0	106	112	128	136	156	160	166
f (Hz)	6144	6400	7200	7680	9216	9600	10240

Egy másik megoldási mód a következő: tudjuk, hogy egy gépi ciklus (T_{cycle})

$$\frac{1}{\frac{f_{CLK}}{12}} = \frac{1}{\frac{11.0592 \cdot 10^6 Hz}{12}} = 1,085069 \mu s$$

időt vesz igénybe, azaz egy időzítő ekkora időközönként tud léptetni az értékén. Azt is tudjuk, hogy pontosan $5 \cdot 10^{-3}$ szekundumos időközönként szeretnénk megszakítani ($T_{interrupt}$) a programunkat. Ebből felírható egy nagyon elegáns, általános képlet, mely leírja, pontosan hány léptetés (N) szükséges az időzítőnek, hogy akkora időközönként csorduljon túl, amit szeretnénk.

$$N = \frac{T_{interrupt}}{T_{cycle}}$$

Számoljuk ki, mennyi lesz N értéke, ha 5ms-os időalapot szeretnénk generálni!

$$N = \frac{5 \cdot 10^{-3} s}{1,085069 \cdot 10^{-6} s} = 4608$$

Sajnos beleütköztünk egy hatalmas problémába. 4608 lépést kellene megtennie az időzítőnek a kívánt túlsordulási frekvenciához (200Hz), viszont most nyolcbites üzemmódban használjuk, ahol a maximális lépésszám 256, melyhez 3600Hz tartozik. Ennél alacsonyabb frekvenciával nem lehet megszakításokat kérni a kettes üzemmódban. Ha az 1-es üzemmódot használnánk, akkor probléma nélkül meg tudnánk tenni a nagyobb frekvenciaosztást, de most szoftveresen kell belenyúlnunk a megszakításokba.

Szerencsére a 3600Hz tizenhatalmadik pont 200Hz! Azaz, ha csak minden tizenhatalmadik megszakítás alkalmával lépünk be a rutin törzsébe, akkor

megoldjuk minden problémánkat. Így fog kinézni az 5 milliszekundumonként végrehajtott T0 túlsordulásának megszakítási rutinja.

```

1 ISR_TIMER0_OVERFLOW:
2     ;megtörtent az interrupt 18 alkalommal?
3     ;ha nem, akkor kilepünk belöle
4     DJNZ     R6,ISR_TIMER0_OVERFLOW_END
5     MOV      R6,#18
6     ;Innen indul a 200Hz-es megszakítás torzse
7     PUSH     ACC
8     MOV      A,P3
9     CLR      A           ;p3 porton villogo
10    MOV      P3,A
11    POP      ACC
12 ISR_TIMER0_OVERFLOW_END:
13    RETI

```

Vedd észre, hogy ismét az ISR előtag szerepel a rutin nevében, és ugyanúgy a RETI utasítással térünk vissza. Továbbá a TF0 jelzőbitet nem nekünk kell kitörölni, ugyanis ezt automatikusan megteszi a mikrokontroller. A soros portnál ez nem volt igaz az RI és TI flagekre, mivel ott nekünk a programból kellett kitörölni a biteket.

Ha sikerült megemészteni, mit is csinál ez a megszakítási rutin, akkor konfiguráljuk fel a T0-ás időzítőt, és a megszakítási rendszert!

```

1 S_INIT_TO_MODE2_200HZ:
2     ANL      TMOD,#0FOH ;maszkolas
3     ORL      TMOD,#02H  ;mode2
4     SETB     TR0        ;Timer Run 0
5     MOV      R6,#18     ;R6-tal szamlaljuk a 18
6     RET       ;tulcsordulast
7 ;nem is kell ujratoltesi ertekek, mert 18*256 = 4608
8 S_INIT_INTERRUPTS:
9     SETB     ETO        ;Enable Timer0
10    SETB     EA          ;Enable All
11    RET

```

Az S_INIT_TO_MODE2_200HZ szubrutinnal inicializáljuk a T0-ás időzítőt 2-es üzemmódra bitmaszkolás segítségével, majd a TR0 bittel aktivizáljuk őt. Az R6 regiszterben fogjuk számlálni, hogy megtörtént-e már a 18 túlsordulás. Az S_INIT_INTERRUPTS szubrutinban pedig beállítjuk a megfelelő biteket, hogy a T0-ás időzítő megszakíthassa a főprogram futását.

Most már csak azt kell megtudnunk, hol található a T0-ás időzítő megszakítási vektora. A [korábban megismert](#) táblázatban azt láthattad, hogy a TF0 (Timer overFlow 0) vektor a 000BH címen található, tehát ide kell egy LJMP ISR_TIMER0_OVERFLOW utasítást beszúrnunk! Lássuk, hogyan is néz ki az egész program!

```

1      CSEG      AT      0000H
2      LJMP      0030H
3
4      ;T0 idozito tulcsordulas vektora
5      CSEG      AT      000BH
6      LJMP      ISR_TIMER0_OVERFLOW
7      ;Innen indul a foprogram
8      CSEG      AT      0030H
9
10     LCALL     S_INIT_TO_MODE2_200HZ
11     LCALL     S_INIT_INTERRUPTS
12 END_OF_PROGRAM:
13     LJMP      END_OF_PROGRAM
14
15 ;MEGSZAKITASI RUTINOK MINDIG
16 ;A LEZARO CIKLUS UTAN JONNEK
17 ISR_TIMER0_OVERFLOW:
18     ;Ez a resz 3600Hz-es megszakitas
19     ;megtortent az interrupt 18 alkalommal?
20     ;ha nem, akkor kilepünk belöle
21     DJNZ      R6,ISR_TIMER0_OVERFLOW_END
22     ;Innen indul a 200Hz-es megszakitas torzse
23     MOV       R6,#18
24     LCALL     S_BLINKY_ON_P3
25 ISR_TIMER0_OVERFLOW_END:
26     RETI
27 ;SZUBRUTINOK
28 S_INIT_TO_MODE2_200HZ:
29     ANL       TMOD,#0FOH
30     ORL       TMOD,#02H
31     SETB      TR0
32     MOV       R6,#18 ;R6-tal szamlaljuk a 18
33     RET       ;tulcsordulast
34 S_INIT_INTERRUPTS:
35     SETB      ET0 ;Enable Timer0
36     SETB      EA ;Enable All
37     RET

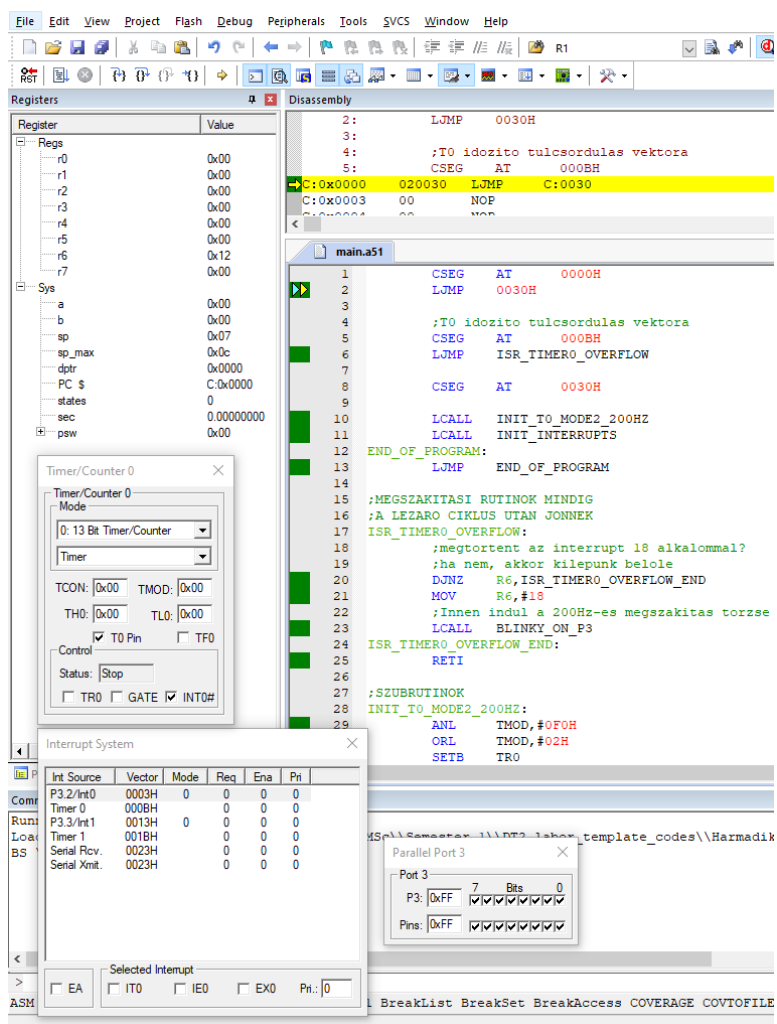
```

```

38 S_BLINKY_ON_P3:
39     PUSH    ACC
40     MOV     A,P3
41     CPL     A           ;p3 porton villogo
42     MOV     P3,A
43     POP     ACC
44     RET
45     END

```

A szokásos módon gépeld be a programot, fordítsd le (Build), majd indítsd el a szimulátort! Hívd elő a P3 portot, az Interrupts és a Timer → Timer 0 paneleket a Peripherals ablakból!



69. ábra. Timer0 és Megszakítás panelek

A Timer/Counter 0 panelen láthatod, hogy éppen milyen értékek vannak az időzítővel kapcsolatos SFRekben illetve, hogy milyen üzemmódban működik a modul. Ezek az értékek a program léptetése, futtatása során automatikusan frissülnek. Az Interrupt system ablakban azt láthatod, hogy melyik forrás kér megszakítást (Request), illetve melyik forrás engedélyezett (Enable). Ezen felül még egy prioritást (Priority) is látni fogsz, de ezzel most nem foglalkozunk.

Az ablakok kihelyezése után léptesd (Step, F11) a programot! Figyeld meg, mi történik. Mikor fog elugrani a programszámláló a 000BH címre? A léptetés után tegyél breakpointot az ISR-en belül a DJNZ utasításhoz! Futtasd (Run, F5) a programot! Milyen időközönként fog újra és újra elugrani a program a megszakítási rutinhoz? Számold ki a Registers ablakban található idő (sec) alapján!

$$T_{interrupt} = T_2 - T_1$$

Most pedig, helyezd át a breakpointot az S_LCALL BLINKY_ON_P3 utasításhoz! Futtasd a programot ismét, majd számold ki, így mennyi idő telik el két megszakítás között! Pontosan 5ms-ot kapsz? Hogyan lehetne 10, 20 vagy 100 ms-os időzítésre kiterjeszteni a megszakítást?

Oldjuk meg, hogy a villogót megvalósító szubrutin csak minden fél másodpercben fusson le! Az időalapunk 5ms, viszont fél másodperc 500ms. Terjesszük ki ismét a rutint, de most használjunk egy másik regisztert is hozzá. Ezt azért célszerű így csinálni, mivel ebben az esetben az időalapunk megmarad 5ms-nak, azaz nem csak egyetlen feladat ütemezésére használhatjuk a megszakítást. A feladat, hogy csinálunk egy altörzset a rutinon belül, ami már csak minden 1800. túlsordulás alkalmával fut le.

```

1 ISR_TIMER0_OVERFLOW:
2     ;Ez a rész 3600Hz-es megszakítás
3     DJNZ     R6,ISR_TIMER0_OVERFLOW_END
4     ;Innen indul a 200Hz-es megszakítás torzse
5     MOV      R6,#18
6     DJNZ     R7,ISR_TIMER0_OVERFLOW_END
7     ;Innen pedig a 2Hz-es torzs!
8     MOV      R7,#100 ;100*18 = 1800 --> 2Hz
9     LCALL    S_BLINKY_ON_P3
10 ISR_TIMER0_OVERFLOW_END:
11     RETI

```

3.4.2. Generáljunk más időalapot!

Minden szuper! 5ms-os időzítést, illetve ennek az egész számú többszöröseit hiba nélkül tudunk generálni. Mi a helyzet, amennyiben például 100 μ s-os időzítést szeretnénk csinálni?

$$N = \frac{T_{interrupt}}{T_{cycle}}$$

ha most $T_{interrupt} = 100\mu s$ akkor

$$\frac{100 \cdot 10^{-6}s}{\frac{1}{11.0592 \cdot 10^6 Hz} \cdot 12} = 92,16$$

Sajnos 92,16 gépi ciklust nem lehet számlálni, csak 92 vagy 93 ciklust. Mi lenne, ha 11.0592MHz helyett 12MHz-es órajelünk lenne?

$$\frac{100 \cdot 10^{-6}s}{\frac{1}{12 \cdot 10^6 Hz} \cdot 12} = 100$$

Ó, hát igen. Mennyivel egyszerűbb dolgunk lenne... Ez a nagy hátránya, a soros kommunikáció hibamentessége érdekében választott oszcillátor frekvenciának. Ott kellemesen elvultunk, most pedig rettenetes módon szenvedünk. Az újabb 8051-es variánsokban ezt a problémát át is hidalták azzal, hogy az UART modul 12MHz-es órajelből is képes előállítani a standard baudrátákat.

A 92,16-os problémát javítani lehet egy kis okoskodással. $92,16 = 92 + \frac{4}{25}$. Ha 25 eseményre bontjuk szét a megszakítást, akkor amennyiben 21 alkalommal 92-vel, négy alkalommal pedig 93-mal töltjük fel TH0 regisztert, úgy:

$$92 \cdot \frac{21}{25} + 93 \cdot \frac{4}{25} = 92,16$$

Ez a megoldás működőképes, de csak nagyobb léptékkal mérve fog megfelelő átlagot szolgáltatni. Olyan időalapot kell keressünk, ami bombabiztos, emberi fejjel is jól lehet használni (2, 5, 10 és ezek többszörösei, valamilyen nagyságrendben) és képesek vagyunk előállítani hibamentesen a mikrokontroller órajeléből. Ilyenből sajnos nincs sok. A legkisebb egész idő, mellyel megszakításokat lehet generálni 625 μ s. Ehhez 576 alkalommal kell léptetni a számlálót 11.0592MHz-es oszcillátorral. 576-ot nem lehet 8 biten eltárolni, így két természetes szám szorzatára kell bontanunk. Például: $192 \cdot 3$. Azaz, a számláló 192 lépést tegyen meg egy túlcsorduláshoz, a megszakítási rutin

törzset pedig csak minden harmadik alkalommal hajtsuk végre. Így garantáljuk az "atomóra" pontosságú $625\mu\text{s}$ -os időzítést, ami a legkisebb olyan időalap, mely egész érték 11.0592MHz frekvencián.

```

1 S_INIT_T0_MODE2_625us:
2     ANL     TMOD, #0FOH
3     ORL     TMOD, #02H
4     MOV     TH0, #(256-192) ;Tud ilyet is az assembler!
5                                     ;Pontosan annyi kerül a TH0-ba
6                                     ;amennyi 192-vel kevesebb mint
7                                     256
8     MOV     R6, #3             ;kiterjesztes (3*192 = 576)
9     SETB    TRO
10    RET

```

Jön a varázslat: szerencsére csak egyszer kell inicializálni TH0 értékét, ugyanis a TL0 regiszter ami számlál, erre az értékre (256-192) fog feltöltődni minden túlsordulás után. Csak az R6 regiszterrel való kiterjesztést kell manuálisan beírni, minden mást automatikusan végez a számláló modul!

```

1 ISR_TIMER0_OVERFLOW:
2     ;208,333333 us-os torzs (625/3)
3     DJNZ    R6, ISR_TIMER0_OVERFLOW_END
4     ;625us-os torzs
5     MOV     R6, #3
6 ISR_TIMER0_OVERFLOW_END:
7     RETI

```

3.4.3. (Majdnem) 5ms-os időzítés 1-es üzemmódban

Tizenhat bites üzemmódban nincs automatikus újratöltés, és nem csak a TL0, hanem a TH0 is számlál! Ismét igaz az alábbi számítás:

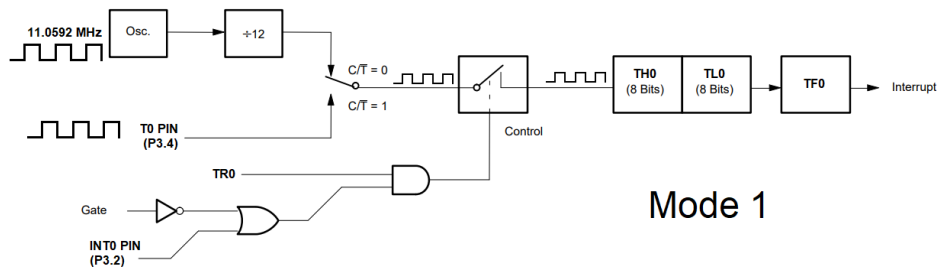
$$N = \frac{5 \cdot 10^{-3}}{1,085069 \cdot 10^{-6}} = 4608$$

4608 alkalommal kell léptetni a számlálót, hogy 5ms-os időzítést kapjunk, viszont 1-es üzemmódban ez az érték elfér 16 biten, így nem kell egy regiszterrel kiterjeszteni az időalapot. Decimális számot az életben nem fogunk tudni értelmesen beírni két darab nyolcbites regiszterbe sok-sok osztogatás nélkül, szóval váltsuk át a 4608-at hexadecimális számrendszerbe!

$$4608_{DEC} = 1200_{HEX}$$

Ezzel már tudunk mit kezdeni. Tizenhat bites üzemmódban 65536 lépést tud megtenni a számláló egy túlsordulásig, ami hexadecimálisan 0x10000. Nekünk ennél 0x1200-val kell kisebb értéket írunk a TH0:TL0 regiszterpárosba, és meg is kapjuk az 5ms-os időzítést!

$$0x10000 - 0x1200 = 0xEE00$$



Innen fel is írhatjuk a megszakítási rutint!

```

1 ISR_TIMER0_OVERFLOW:
2     ;Most nekünk kell feltolteni a számlálót
3     MOV     TH0, #(100H-12H) ;=0xEE
4     MOV     TL0, #00H        ;=0x00
5     LCALL   S_BLINKY_ON_P3
6 ISR_TIMER0_OVERFLOW_END:
7     RETI

```

Figyelem, ugyanazt a megszakítási rutint használjuk, hiszen a forrás (TF0) és a vektor címe nem változik, csak a T0 üzemmódja! A számlálót inicializáló szubrutin pedig:

```

1 S_INIT_T0_MODE1_5ms:
2     ANL     TMOD, #0F0H
3     ORL     TMOD, #01H ;mode 1
4     MOV     TH0, #(100H-12H)
5     MOV     TL0, #00H
6     SETB    TR0
7     RET

```

A teljes program pedig:

```

1     CSEG    AT      0000H
2     LJMP    0030H
3
4     ;T0 idozito tulcsordulas vektora
5     CSEG    AT      000BH
6     LJMP    ISR_TIMER0_OVERFLOW

```

```

7
8          CSEG      AT      0030H
9
10         LCALL     S_INIT_TO_MODE1_5ms
11         LCALL     S_INIT_INTERRUPTS
12 END_OF_PROGRAM:
13         LJMP      END_OF_PROGRAM
14 ;MEGSZAKITASI RUTINOK MINDIG
15 ;A LEZARO CIKLUS UTAN JONNEK
16 ISR_TIMER0_OVERFLOW:
17         ;Most nekunk kell feltolteni a szamlalat
18         MOV       TH0,#(100H-12H) ;=0xEE
19         MOV       TLO,#00H      ;=0x00
20         LCALL     S_BLINKY_ON_P3
21 ISR_TIMER0_OVERFLOW_END:
22         RETI
23 ;SZUBRUTINOK
24 S_INIT_TO_MODE1_5ms:
25         ANL       TMOD,#0FOH
26         ORL       TMOD,#01H ;mode 1
27         MOV       TH0,#(100H-12H)
28         MOV       TLO,#00H
29         SETB      TR0
30         RET
31 S_INIT_INTERRUPTS:
32         SETB      ET0          ;Enable Timer0
33         SETB      EA          ;Enable All
34         RET
35 S_BLINKY_ON_P3:
36         PUSH      ACC
37         MOV       A,P3
38         CPL       A           ;p3 porton villogo
39         MOV       P3,A
40         POP       ACC
41         RET
42         END

```

Teszteld, hogy valóban megfelelő időközönként érkezik-e a megszakítás! Pontosan 5ms-ot kellene kapnod, de nem ezt fogod tapasztalni. Elkövettünk egy hatalmas hibát. **A szoftveres újratöltés egy óriási hülyeség.** Míg hardveres újratöltésnél azonnal megtörténik a számláló feltöltése a túlsor-
dulás alkalmával, szoftveres esetben csak a MOV utasítás után történik meg

ugyanez.

Ezzel a nagy probléma, hogy valójában nem lesz pontos az időalap. Pár gépi ciklusnak megfelelő mikroszekundummal el fog csúszni az időzítés. Gondolj bele: túlcsordul a számláló, és érkezik egy megszakítási kérés. A mikrokontroller még befejezi az éppen végrehajtás alatt lévő utasítást, mely 1, 2 vagy 4 gépi ciklust vehet igénybe. Legrosszabb esetben már 4 ciklusidőnyi mikroszekundummal elcsúsztunk. A vektor címén egy LJMP utasítás is található, mellyel újabb 2 ciklusidővel megcsúszunk, és csak innentől kezdjük el feltölteni a számlálót a szükséges értékkel. Sőt, maga feltöltés is időt vesz igénybe, ami a `MOV direct, #data` esetén újabb 2 gépi ciklus. Ha a feltöltés előtt még lennének utasítások, akkor azokat is bele kellene számolni a tényleges újratöltési érték meghatározásához. Hogyan tudjuk garantálni, hogy a megfelelő értékkel legyen feltöltve a számláló? Sehogyan. Öt gépi ciklustól kezdve gyakorlatilag bármennyi idő eltelhet a feltöltésig.

Most éppen szerencsés eset van, mivel a TL0 regiszternek 00H-tól kell számolnia, és 256 lépést biztosan nem fog megtenni, hogy a TH0 regiszter is lépjen egyet. Kivételesen csak a TH0 regisztert kell újratöltenünk, a TL0 regisztert pedig hagyhatjuk számlálni.

```
1 ISR_TIMER0_OVERFLOW :  
2     ;Most nekunk kell feltolteni a szamlalot  
3     MOV     TH0, #(100H-12H) ;=0xEE  
4     LCALL   S_BLINKY_ON_P3  
5 ISR_TIMER0_OVERFLOW_END :  
6     RETI
```

Ezzel a kicsi módosítással sem fogjuk megkapni a pontos 5 ms-os időzítést, de közelebb leszünk hozzá. A tanulság: **szoftveres újratöltést nem használunk, mert bizonytalan időzítést fog eredményezni! Csak speciális esetben (ha TL0 újratöltési értéke = 0) lehet alkalmazni, de úgy sem lesz teljesen pontos az időalap!** Egy idő-multiplexelt ledes kijelző meghajtásánál például egyáltalán nem probléma, ha nem 100%-osan pontos a kijelző léptetése, hiszen úgy sem látjuk szabad szemmel a pár mikroszekundumnyi csúszást. Egy jel periódusidejének mérése esetén viszont, nem lehetünk pontatlanok. Ott pontos időalapra van szükség.

Miért nem lehetünk biztosak a szoftveres újratöltés pontosságában? Írd le a jegyzőkönyvbe a magyarázatot, rajzzal együtt.

3.5. Oldjuk meg a második labor lehetetlen feladatát!

"Csinálj balra futó futófényt 1 másodperces késleltetéssel a P1 porton! Csinálj a P3 porton villogót 1 másodperces periódusidővel, 50%-os kitöltéssel! Ha érkezett karakter a soros porton, akkor annak az értékét írd ki a P2 portra, majd küldd vissza a számítógépnek a vett karaktert! **Ja igen, ezt a négy feladatot 1 programba kellene belesűríteni úgy, hogy mind a négyet végzi a mikrokontroller egyszerre!** *Ez tényleg egy bazi nehéz feladat!*"

Most nem egy, hanem két megszakítást is alkalmazni fogunk egyszerre! A soros portot és a T0-ás időzítő vektorait is felhasználjuk! A soros port megszakítását már megírtuk, most csupán ki kell egészítenünk a P2-es portra való kiírással!

Kezdjük a soros porttal! Inicializáljuk, majd megírjuk a megszakítási rutint:

```
1 S_INIT_SERIAL_9600 :
2     MOV     SCON ,#50H
3     ANL     TMOD ,#0FH
4     ORL     TMOD ,#20H
5     MOV     TH1 ,#0FDH
6     SETB    TR1
7     RET

1 ISR_SERIAL_PORT :
2     CLR     TI
3     JNB     RI , ISR_SERIAL_NO_NEW_DATA_IN_SBUF
4     CLR     RI
5     MOV     SBUF , SBUF
6     MOV     P2 , SBUF
7 ISR_SERIAL_NO_NEW_DATA_IN_SBUF :
8     RETI
```

Ezután haladjunk a futófény és a villogó szubrutinjaival!

```
1 S_BLINKY_ON_P3 :
2     PUSH    ACC
3     MOV     A , P3
4     CPL     A           ;p3 porton villogo
5     MOV     P3 , A
6     POP     ACC
7     RET
```

```

1 S_CHASER_ON_P1:
2     PUSH    ACC
3     MOV     A,P1
4     RL      A           ;futofeny balra
5     MOV     P1,A
6     POP     ACC
7     RET

```

Inicializáljuk a T0-ás időzítőt 2-es módra!

```

1 S_INIT_T0_MODE2:
2     ANL     TMOD,#0FOH
3     ORL     TMOD,#02H ;mode 2
4     MOV     R6,#18 ;kiterjesztes
5     SETB    TR0
6     RET

```

Majd írjuk meg az időzítést kezelő ISR rutint! Ki kell terjesszük az 5ms-os időalapot fél másodpercre, amit egy 100-as osztással tehetünk meg. Ennél az altörzsnél meghívjuk a villogót, majd egy következő altörzsben 1 másodpercenként meghívjuk a futófényt.

```

1 ISR_TIMER0_OVERFLOW:
2     ;3600Hz
3     DJNZ    R6,ISR_TIMER0_OVERFLOW_END
4     MOV     R6,#18
5     ;200Hz
6     DJNZ    R7,ISR_TIMER0_OVERFLOW_END
7     ;2Hz
8     MOV     R7,#100 ;100*5ms = 0.5 sec
9     LCALL   S_BLINKY_ON_P3
10    CPL     F0
11    JNB     F0,ISR_TIMER0_OVERFLOW_END
12    ;1Hz
13    LCALL   S_CHASER_ON_P1
14 ISR_TIMER0_OVERFLOW_END:
15    RETI

```

Inicializáljuk a megszakítási rendszert!

```

1 S_INIT_INTERRUPTS:
2     SETB    ET0         ;Enable Timer0
3     SETB    ES          ;Enable Serial
4     SETB    EA          ;Enable All
5     RET

```


Beillesztjük a megszakítási vektortáblába a szükséges ugrásokat. A 0023H címről a soros port rutinjára, a 000BH címről pedig a T0-ás időzítő rutinjára ugrunk. **Most már nevezzük a valódi nevén a kezdő LJMP utasítás címét (0000H) is: ő a reset vektor.** A reset vektor speciális megszakítás, ugyanis ebből nem térünk vissza RETI utasítással, de megszakításként kezelendő, hiszen bármikor megnyomhatom a reset gombot! Sőt, a továbbfejlesztett 8051-es változatokban megjelent a Watchdog áramkör is, amivel bizonyos időközönként reset vektorokat lehet kérni, ezzel feloldva a véletlenül rosszul futó programokat. Erről majd később, szakirányon lesz bővebben szó úgy is.

```

1      ;Reset vektor (most mar nevezzuk neven)
2      CSEG      AT      0000H
3      LJMP      0030H
4      ;T0 idozito tulcsordulas vektora
5      CSEG      AT      000BH
6      LJMP      ISR_TIMER0_OVERFLOW
7      ;Soros port megszakitasi vektora
8      CSEG      AT      0023H
9      LJMP      ISR_SERIAL_PORT
10     ;program kezdete
11     CSEG      AT      0030H

```

Végül pedig hívjuk meg az inicializáló szubrutinokat, és írjuk meg a főprogramot!

```

1      MOV       P1,#1
2      LCALL     S_INIT_SERIAL_9600
3      LCALL     S_INIT_TO_MODE2
4      LCALL     S_INIT_INTERRUPTS
5  END_OF_PROGRAM:
6      LJMP     END_OF_PROGRAM

```

Jé, full üres a főprogram! A mikrokontroller minden feladatát megszakításokkal végzi, az ideje többi részét pedig üresjáratban tölti. Ki gondolta volna, hogy így is meg lehet oldani ezt a nehéz feladatot? :) Dobjunk bele a főprogramba egy utasítást, amivel el tudjuk küldeni aludni (IDL: Idle) a 8051-est, amikor éppen nem csinál semmit, ne fogyasszon feleslegesen!

```

1  END_OF_PROGRAM:
2      MOV       PCON,#01H      ;IDL mode ON
3      LJMP     END_OF_PROGRAM

```

Hegeszd össze a programot a részletek alapján, majd teszteld, hogy minden működik-e rendesen! Valóban pontosan 0.5 és 1 másodperceként fognak lefutni a portokat manipuláló szubrutinok? A soros portnál valószínűleg nem fogsz tudni olyan gyorsan gépelni, mintha valós mikrokontrolleren futna a program. Vedd fel a jegyzőkönyvbe a teljes programot! Ha futás közben megállítod a programot (Stop, piros X) majd elkezded léptetni a főprogramot, akkor furcsa viselkedést fogsz tapasztalni. Mi történt a programmal? *Minden jól működik, de valami miatt megállt a főprogram végrehajtása.*

3.5.1. Mi történik, ha túl hosszú a megszakítási rutin?

Amennyiben több ideig tartana egy ISR rutin lefuttatása, mint két megszakítási kérés között eltelt idő, úgy értelmét veszti az időzített megszakítás. A példákban bemutatott programoknál egy-egy rutin lefuttatása lényegesen kevesebb időt vett igénybe, mint a számláló túlsordulásához szükséges idő, így nem történt hiba. Nézzünk egy elrettentő példát, hogyan lehet elrontani a programot!

```

1      CSEG      AT      0
2      LJMP      0030H
3      CSEG      AT      000BH
4      LJMP      ISR_TIMER0_OVERFLOW
5
6      CSEG      AT      0030H
7
8      LCALL     S_INIT_TIMER0
9      LCALL     S_INIT_INTERRUPTS
10     MOV       P1 , #1
11     MOV       SP , #0FH
12 LOOP1:
13     INC       P1
14     LCALL     S_DELAY
15     LJMP      LOOP1
16
17 END_OF_PROGRAM:
18     LJMP      END_OF_PROGRAM
19
20 ISR_TIMER0_OVERFLOW:
21     INC       A
22     LCALL     S_FILL_XDATA_RAM
23     RETI

```

```

24
25 S_INIT_TIMER0:
26     ANL     TMOD ,#0FH
27     ORL     TMOD ,#02H
28     SETB    TRO
29     RET
30 S_INIT_INTERRUPTS:
31     SETB    ETO
32     SETB    EA
33     RET
34 S_FILL_XDATA_RAM:
35     PUSH    DPH
36     PUSH    DPL
37     PUSH    PSW
38     MOV     PSW ,#08H ;BANK1
39     MOV     R2 ,#0
40     MOV     R3 ,#0
41     MOV     DPTR ,#0
42     MOVX    @DPTR ,A
43     INC     DPTR
44     DJNZ    R2 ,-$-2
45     DJNZ    R3 ,-$-4
46     POP     PSW
47     POP     DPL
48     POP     DPH
49     RET
50
51 S_DELAY: ;20us
52     MOV     R2 ,#71
53     DJNZ    R2 ,$
54     RET
55
56     END

```

Ennek a programnak semmi értelme nincs, de nagyon jól szemlélteti, mi történik akkor, ha picit túl hosszúra írjuk a megszakítási rutint. A főprogramban minden ≈ 20 mikroszekundumonként szeretnénk egy számlálót léptetni a P1 porton. A T0 időzítő segítségével pedig 3600Hz-es frekvenciával szeretnénk feltölteni az egész XDATA memóriát oly módon, hogy minden megszakítás megérkezésekor egyel nagyobb értékek kerüljenek a rekeszekbe.

A probléma ott kezdődik, hogy az egész XDATA feltöltéséhez jóval, de jóval több idő szükséges mint az itt beállított $\frac{1}{3600}$ szekundum. Ilyenkor

a megszakítási rutin futása közben érkezik egy újabb megszakítási kérés, de azt nem tudjuk feldolgozni addig, amíg az éppen kiszolgálás alatt lévő megszakítási rutin le nem futott. Emiatt a memória feltöltésének ütemezése teljesen elcsúszik, az időzített megszakítás értelmét veszti. Továbbá a rutin végigfut, de amint visszatérünk belőle a RETI utasítással, már aktív a következő kérés is. A főprogram emiatt két megszakítás között csak 1 utasítást fog tudni elvégezni, azaz a késleltető szubrutint is teljesen elrontottuk.

Teszteld a programot! Mérd meg, mekkora időközönként számlál a P1 port. Köze nem lesz a 20 mikroszekundumhoz. A megszakítás milyen időközönként fut le? Elméletben minden 3600. gépi ciklus alkalmával szeretnénk elvégezni az XDATA feltöltését, de ennél jóval több idő szükséges ehhez.

3.6. Konkurens megszakítások kezelése

Mi történik akkor, ha egyszerre érkezik két megszakításkérés? Orwelli szólással: minden megszakítás egyenlő, de egyes megszakítások egyenlőbbek a többinél. Ilyenkor a megszakítási rendszeren belül, egy belső prioritási sorrend kerül érvényre. A hardver minden gépi ciklusban lekérdezi (polling) az egyes forrásokhoz tartozó kérések aktivitását, majd felállít egy egyszintű sorrendet. Ezt a sorrendet sajnos nem tudjuk befolyásolni, de logikusan lett felépítve a rendszer. Az egyszintű prioritás sorrendje a következő (balról jobbra csökken a prioritás):

$$IE0 \rightarrow TF0 \rightarrow IE1 \rightarrow TF1 \rightarrow RI+TI$$

Azaz, ha például egyszerre érkezik kérés a T0-ás időzítőtől és a soros porttól is, akkor a T0-ás időzítő kérését fogjuk előbb feldolgozni. Ha abból a megszakításból visszatértünk a RETI utasítással, akkor elindul a soros port megszakításának kiszolgálása, de csak miután elvégzett legalább 1 utasítást a mikrokontroller a főprogramból. **Ez a rész érdekesség, a labor során nem fogunk foglalkozni ezzel.**

3.7. Megszakítás megszakítása

Mi van akkor, ha éppen egy megszakítási rutint végzünk, de egy másik megszakítási kérés is érkezik? A sütős-mosós-tanulós példát folytatva: A tanulós főprogramod megszakítja a mosógéptől jövő időzítő túlsordulása. Elkezdted az ISR futtatását azzal, hogy kipakolsz a gépből, de közben a sütő időzítője is túlsordult, és az is kér egy megszakítást tőled. Befejezed előbb a mosógépből való kipakolást, vagy nagyobb prioritással kezeled az elkészült pizzát? Valószínűleg nem szeretnéd, ha odaégne a pizza, így megszakítod az éppen futó ISR-t egy másik megszakítással. A pizza megszakítás kiszolgálása után visszatérsz a mosógépes megszakításhoz, majd ha azzal is végzel, akkor visszatérsz a főprogramhoz.

Ezt több szintű megszakításnak hívjuk, és a 8051-es is tud ilyet. Két prioritási szint létezik. Az alacsonyabb prioritási szinten (0) kezelt megszakításokat, megszakíthatja egy magasabb prioritási szinten (1) lévő megszakítás, viszont a magasabb szinten lévő megszakítást már nem lehet megszakítani. Ezt a funkciót az IP (Interrupt Priority, megszakítás prioritás) regiszterrel lehet beállítani. A feladatnak szükséges módon állíthatjuk be, melyik

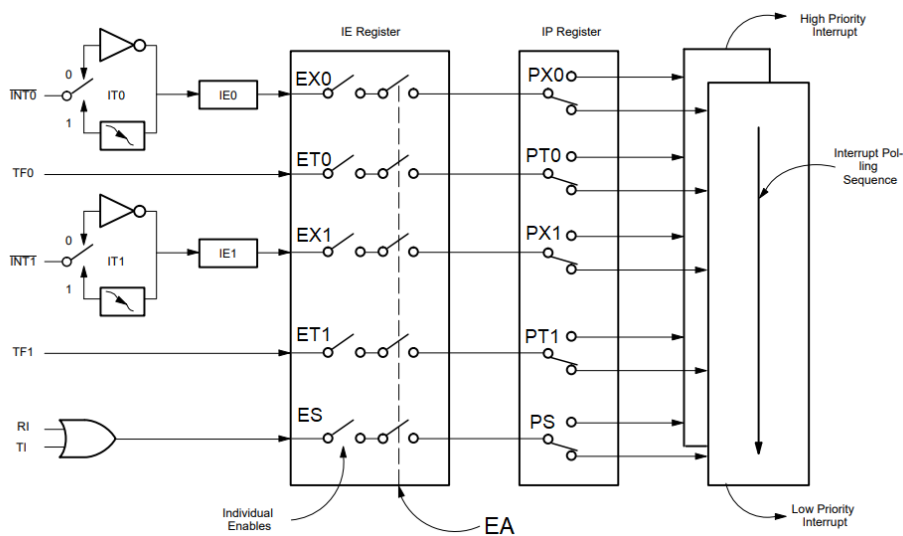
megszakítás élvezzen prioritást a többivel szemben. Amennyiben a magasabb szintű megszakítások között versenyhelyzet lépne fel, úgy az egyszintű sorrend érvényesül.

$$IE0 \rightarrow TF0 \rightarrow IE1 \rightarrow TF1 \rightarrow RI+TI$$

7	6	5	4	3	2	1	0
x	x	x	PS	PT1	PX1	PT0	PX0

IP regiszter

A bitek elnevezése az IE (Interrupt Enable) regiszterével megegyezők, csupán itt nem "E" (Enable) hanem "P" (Priority) előtagot kaptak, illetve a globális engedélyezéshez nem tartozik prioritás bit, hiszen egy megszakítási rendszer létezik csak. A 62. ábrán látott egyszerűsített megszakítási rendszert teljes egészében a 70. ábrán láthatod! **Ez a funkció is csak**



70. ábra. A teljes megszakítási rendszer felépítése

érdekességnek került bele az útmutatóba, bővebben csak PIR-en fogunk vele foglalkozni.

3.8. Alacsony fogyasztási üzemmód

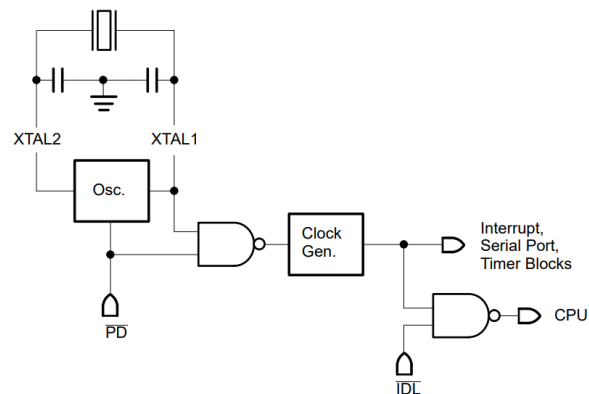
A legtöbb mikrokontroller (és mobil processzor) élete legnagyobb részét alvásban (IDL, idle) tölti. Ebben az állapotban az órajel csak a perifériákhoz (soros port, időzítők) és a megszakítási rendszerhez jut el, miközben a végrehajtó egységnél lekapcsol. Ez azt eredményezi, hogy a program futása ténylegesen megáll. A mikrokontroller nem fog több utasítást végrehajtani a low power mód aktiválása után egészen addig, amíg fel nem kelti valamilyen megszakítás ebből az állapotból. Miután felkelt a mikrokontroller, végrehajtja a megszakítási rutin lefuttatását, majd visszaugrik a főprogramra. Innen tovább kezdi futtatni a programot egészen addig, amíg új alvás parancs nem érkezik. **Emiatt nem merül le a telefon fél óra alatt, és pontosan emiatt lehet napokig használni az AirPods fülest is töltés nélkül.**

Egy másik alacsony fogyasztási üzemmód a teljes megállás (PD, power down), mely aktiválásakor az oszcillátor jelét egyik modulhoz sem továbbítjuk. Ilyenkor még a megszakítások sem keltik fel a mikrokontrollert. Ahhoz, hogy újra üzemképes legyen a processzor, hardveres reset szükséges.

Ezeket a funkciókat a PCON (Power CONtrol) SFR állításával lehet elérni, mely nem bitcímezhető. Ebben a regiszterben található még az SMOD bit is, mellyel a soros port baudrátáját lehet megduplázni. A GF bitek pedig szabadon felhasználhatók. **Ez is csak mint érdekesség került az útmutatóba.**

7	6	5	4	3	2	1	0
SMOD	X	X	X	GF1	GF0	PD	IDL

PCON regiszter



3.9. Önálló feladatok:

A megszakítások és az időzítők használata nem egy egyszerű témakör, így csak pár könnyű önálló feladat lesz most!

- Csinálj felfelé számlálót a P1 porton 35 ms-os időzítéssel. Használd a T0-ás időzítő 2-es módját!
- **Szabadon futó** időzítővel csinálj egy Knight Rider futófényt a P2 porton és az XDATA memóriába ágyazott ledemen! Használd a T0-ás időzítőt, 1-es módban! Mekkora frekvenciával fog ismétlődni a futófény? Most pontos lesz az időzítés?
- Csinálj változtatható gyorsaságú lefelé számlálót a P2 porton! Használd a T0-ás időzítőt 2-es módban, időalapnak állíts be 100ms-ot! A gyorsaságot 100ms-tól 2.5s-ig ($100ms \cdot [0-250]$) tudd változtatni a soros porton bevitt szám függvényében! Ehhez használd a következő oldalon található S_SERIAL_DECIMAL_IN szubrutint! Ha tudod, akkor oldd meg ugyanezt, csak a soros port megszakításával! Mit csinál a szubrutin? Milyen megkötések mellett működik?

AKI ÉHEZIK A KIHÍVÁSRA: A főprogramban valósítsd meg az alábbi kombinációs hálózatot!

$$F = \sum_{i=0}^3 (0, 1, 4, 5) \quad C \div 2^2 \quad B \div 2^1 \quad A \div 2^0$$

A T0-ás időzítő 3-as módját felhasználva, csinálj 8 bites Johnson számlálót a P2 porton 25Hz-es ismétlődési frekvenciával a TF1 megszakítási vektorral. Ezek mellett csinálj a TF0 megszakítási vektorral 4 bites Gray-kód számlálót a P1 porton 200Hz ismétlődési frekvenciával! A soros port megszakításával vidd be a főprogramban futtatott kombinációs hálózatnak a bemenetet, majd írd ki a terminálra az adott bemenethez tartozó kimenet értékét, illetve jelenítsd is meg a P0 porton! A főprogram csak új adat érkezése esetén fusson le, használj IDL módot! Az alábbi reguláris kifejezéssel leírt formátumban írd ki a terminálra, de csak akkor, ha érkezett új bemeneti kombináció!

```
^((?!\\.\\*))*Bemenet.*:[0-7] \\1.*:[0-1]$
```



```

1 S_SERIAL_DECIMAL_IN:
2     CALL    S_SERIAL_READ_BYTE
3     CLR     C
4     SUBB    A,#30H ;ASCII-->binaris
5     MOV     B,#100
6     MUL     AB
7     MOV     R2,A
8     LCALL   S_SERIAL_READ_BYTE
9     CLR     C
10    SUBB    A,#30H ;ASCII-->binaris
11    MOV     B,#10
12    MUL     AB
13    ADD     A,R2
14    MOV     R2,A
15    LCALL   S_SERIAL_READ_BYTE
16    CLR     C
17    SUBB    A,#30H ;ASCII-->binaris
18    ADD     A,R2
19    RET

```

4. Ami a laborokból kimaradt: Hogyan legyünk igazi fekete öves ASM programozók?

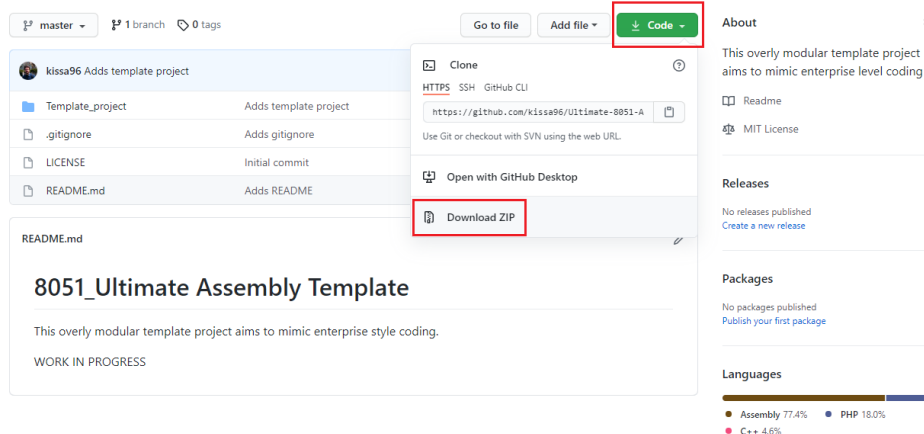
Ez a pár oldal, már csak egy nagyobb falat extra azoknak, akiket érdekel, hogyan is lehet kimaxolni egy assembly programot. 180 fokos fordulatot veszünk, és felépítünk egy olyan környezetet, melynek minden kis részlete moduláris.

Ugorj fejest a mély vízbe, ha érdekel. Több fájlos program feltételes fordításokkal, kereszthivatkozásokkal, direktívákkal, relokálható szegmensekkel, makrókkal, aliasokkal, változókkal, startup kóddal, már fordított kódokkal és minden egyébbel. Ennél többet nem igen lehet (és nem is érdemes) kihozni egy Assembly programból. Ismerjük meg az enterprise szintű kódolás csodás világát!

4.1. Ultimate-8051-ASM-template-project

Az új mérési útmutató mellé, egy teljes sablon projekt is tartozik, mely bár **bőven nem a Digit 2 szintjét tükrözi**, de rengeteget lehet belőle tanulni! Ha van fent a gépeden Git, és használod is, akkor klónozz be ide. Ha nincs, akkor nyisd meg a linket, és töltsd le a kódot.

<https://github.com/kissa96/Ultimate-8051-ASM-template-project.git>



71. ábra. A sablon repója

Bontsd ki a becsomagolt mappát, majd navigálj a `Template_project` almappába. Itt láthatod az összes fájlt, ami a sablonban szerepel, illetve magát a projekt fájlt is (.uvproj). Indítsd el a projekt fájlt!

Ez a sablon a laborban használt 80c552-es mikrokontrollerre készült, de minimális módosításokkal bármely 8051-esre portolható. A projekt megnyitása után a fájllistában bal oldalt, ha kinyitod a mappákat, akkor nem csak a main.a51 állományt láthatod, hanem még legalább 30 másikat is. Minek hozunk létre ennyi fájlt, hiszen

„Jóvanazúgy, ha belerakunk minden kódot egy forrásba! Fordul a program úgy is, most minek t...köljünk a felépítéssel?”

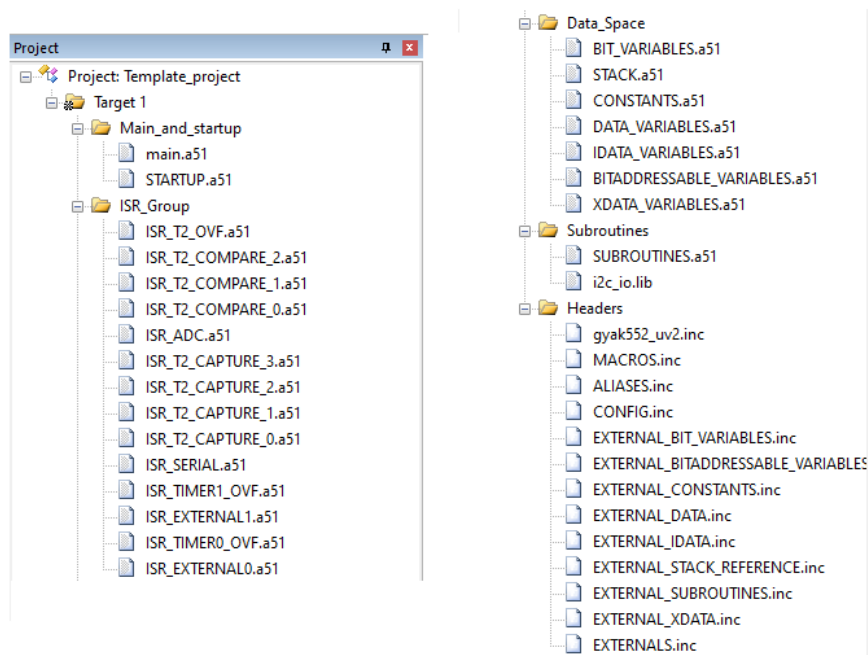
Ismeretlen hallgató

Ez addig jó megoldás, amíg csak egyedül dolgozunk, és csak pici programokat írunk. Amint várható bármilyen szintű változtatás a kódban, vagy újbóli felhasználás, már nagyon nem mindegy, mit hova írunk. Amíg csak 1-2 szubrutinnal vagy megszakítással dolgozunk, addig könnyű a kód karbantartása, hiszen még olvasható minden, de egy komoly program esetében ez már nem igaz.

Amint láthatod, minden szét lett szedve. Külön-külön fájlokban találhatóak a szubrutinok, a megszakítási rutinok, a főprogram, a startup program, a változók, stb. A programozás 100%-ban ugyanúgy történik mint a laboron, csupán most nem egyetlen fájlban dolgozunk. A nagy előnye a moduláris felépítésnek, hogy egyszerre több ember is dolgozhat a projekten, illetve minden programrészletnek meg van a fix helye, így nem kell keresgélni. Ha szubrutint szeretnénk írni, akkor azt a SUBROUTINES.A51, míg ha például a T0-ás időzítő megszakítási rutinját szeretnénk módosítani, akkor azt a ISR_TIMER0_OVF.A51 fájlban tehetjük meg. **Kattints bele a SUBROUTINES.A51 fájlba dupla klikkel, tanulmányozzuk át mi micsoda!**

Minden program, melyet másnak átadunk, kell rendelkezzen valamilyen licenccel, melyben engedélyezzük a használatát bizonyos megkötések mellett, illetve felelősséget is vállalhatunk a működésért. A licenc mindig a forráskód elején található.

A forráskód lényegi része direktívákkal indul, melyek segítségével fájlokat adhatunk hozzá a kódhoz, feltételesen fordíthatunk, illetve szimbólumokat



72. ábra. A projektben megtalálható források

```

/*
MIT License
Copyright (c) 2021 Attila Kiss

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
*/

```

73. ábra. MIT licenc

is definiálhatunk. A **\$NOMOD51** direktíva segítségével „kikapcsoljuk” a generic 8051-es mikrokontroller regiszter definícióit, mert a projekt 80552-

est használ, mely regisztereinek címeit a REG552.INC fájlban találhatjuk. Ezt a **\$INCLUDE()** direktívával tudjuk hozzáadni a forráskódhoz. Ilyenkor minden ami a REG552.INC fájlban található, az láthatatlan módon beillesztődik a forráskódba.

```

8  ;/*  BYTE Register  */
9  P0      DATA  080H
10 P1      DATA  090H
11 P2      DATA  0A0H
12 P3      DATA  0B0H
13 P4      DATA  0C0H
14 P5      DATA  0C4H
15
16 PSW     DATA  0D0H
17 ACC     DATA  0E0H
18 B       DATA  0F0H
19 SP      DATA  081H
20 DPL     DATA  082H
21 DPH     DATA  083H
22 PCON    DATA  087H
23 TCON    DATA  088H
24 TMOD    DATA  089H
25 TLO     DATA  08AH

```

74. ábra. REG552.INC – Regiszterek definíciói

A 74. ábrán a REG552.INC állomány tartalmának egy részletét láthatod. Itt definiáljuk milyen címet is rejt az ACC, PSW, SP és az összes többi név. Az **\$INCLUDE** direktívával ezt az egész fájlt hozzáadtuk a SUBROUTINES.A51 forráshoz, azaz használhatjuk ezeket a fedőneveket a kódban.

A többi include direktívával további fájlokat adunk hozzá a programhoz. Az **EXTERNAL...** állományokból a változók szimbólumait (memóriák címeit), a **MACROS** állományból a definiált makrókat, az **ALIASES** fájlból pedig egyéb fedőneveket importálunk. A **CONFIG.INC** tartalmazza az összes fordítással kapcsolatos definíciót. Itt állíthatjuk be a program felépítését.

```

23  $NOMOD51
24  $INCLUDE (ALIASES.INC)
25  $INCLUDE (CONFIG.INC)
26  $INCLUDE (REG552.INC)
27  $INCLUDE (GYAK552_UV2.INC)
28  $INCLUDE (MACROS.INC)
29  $INCLUDE (EXTERNAL_DATA.INC)
30  $INCLUDE (EXTERNAL_IDATA.INC)
31  $INCLUDE (EXTERNAL_XDATA.INC)
32  $INCLUDE (EXTERNAL_BIT_VARIABLES.INC)
33  $INCLUDE (EXTERNAL_BITADDRESSABLE_VARIABLES.INC)
34  $INCLUDE (EXTERNAL_CONSTANTS.INC)
35  $INCLUDE (EXTERNAL_STACK_REFERENCE.INC)

```

75. ábra. Direktívák

Az include direktívák után egyből egy újabb direktíva következik, mely a feltételes fordítást teszi lehetővé. A lefordított és linkelt programban csak akkor fog szerepelni a SUBROUTINES.A51 fájl törzse, ha az ASSEMBLE_SUBROUTINES szimbólum értéke 1. Azaz, ha valamilyen oknál fogva nem szeretnénk használni szubrutinokat, akkor kikapcsolhatjuk ennek a fájlnek a tartalmát a CONFIG.INC állományból, ahol a szimbólumot definiáltuk. Ilyenkor build közben olyan, mintha az include és a fájl lezáró END direktívákon kívül semmi nem lenne a forrásban. Az **\$IF** lezárása az **\$ENDIF** direktíva.

```

37
38 $IF(ASSEMBLE_SUBROUTINES = 1)
39
      .
      .
      .
      .
205
206
207 $ENDIF
208      END

```

76. ábra. Feltételes fordítás

A szubrutinokat összegyűjtő fájl törzsében deklarálunk egy relokálható kódszegmenst. A relokálható (áthelyezhető) szegmensekkel szó szerint legózni fog a linker. Nem mondjuk meg pontosan hol (milyen címen) legyenek az utasítások, csak azt adjuk meg, hogy léteznek, és a linker majd oda teszi le őket a memóriában, ahová optimális.

A relokálható szegmensek ellentettje az abszolút szegmens, melyet a laborok során használtunk bőven a CSEG AT direktívával. Ilyenkor pontosan megmondjuk milyen címen legyenek az utasítások, nem lehet legózni velük. Szegmenst nem csak a kódmemóriára lehet deklarálni, hanem bármely másik memóriaterületre is (DATA, IDATA, XDATA, BIT, BITADDRESSABLE)!

Abszolút szegmenst a **CSEG** (code segment), **XSEG** (xdata segment), **ISEG** (idata segment), **DSEG** (data segment) és **BSEG** (bit segment) direktívákkal lehet deklarálni. Ilyenkor csak a direktíva és egy cím megadása szükséges (pl: CSEG AT 0000H, XSEG AT 0C000H)

Relokálható szegmens létrehozásához előbb deklaráljuk a szegmens

nevét és a típusát, majd át kell váltanunk a szegmensre, hogy írni tudjunk bele. Azzal, hogy külön szegmenst hozunk létre a szubrutinoknak, egy nagy építőkockaként tekinthetünk rájuk.

```
1      SUBROUTINES SEGMENT CODE ;a szegmens létrehozása
2      RSEG          SUBROUTINES ;atváltunk a szegmensre
3      ;innentol barmit irunk, az ebben a szegmensben lesz
```

A szegmens létrehozása után, ha rápillantasz a szubrutinokra, akkor a legtöbb már ismerős lesz a laborokból. Itt is megtalálhatók a soros portot kezelő rutinok, akár csak a futófényeket és beágyazott perifériákat kezelő szubrutinok. A kérdés, hogyan tudjuk rávenni az Assemblert, hogy ezeket a rutinokat a főprogramból (vagy más .A51 fájlokból is) elérhessük?

Erre szolgálnak a **PUBLIC** direktívák, melyek segítségével a szubrutinok címeit (kezdőcímeinek szimbólumait) tudjuk exportálni a többi fájl felé. Előbb felsoroljuk az összes szubrutint a PUBLIC direktívával, majd utána definiáljuk a rutinokat. Így szépen látható milyen rutinokat tartalmaz a fájl, és amennyiben érdekes lenne pontosan mit is csinál egy rutin, akkor legörgetve meg tudjuk nézni.

```
44      PUBLIC S_NOP
45      PUBLIC S_INIT_SERIAL_PORT
46      PUBLIC S_SERIAL_WRITE_BYTE
47      PUBLIC S_SERIAL_READ_BYTE
48      PUBLIC S_SERIAL_WRITE_TEXT_AT_DPTR
49      PUBLIC S_GET_NEXT_INSTR_PC_VALUE_IN_DPTR
50      PUBLIC S_INIT_INTERRUPT_SYSTEM
51      PUBLIC S_WRITE_XDATA_PERIPH_LEDS_FROM_ACC
52      PUBLIC S_READ_XDATA_PERIPH_BUTTONS_INTO_ACC
53      PUBLIC S_INIT_TIMER_0
54      PUBLIC S_CHASER_ON_P1
55      PUBLIC S_COUNTER_ON_P2
```

77. ábra. Szimbólumok exportálása

Nézzük meg, hogyan lehet változókat létrehozni! **Keresd meg a DATA_VARIABLES.A51** nevű fájlt a projekt alatt, majd nyisd meg! Ennek a forrásfájlnak a tartalma sokkal egyszerűbben emészthető. Itt hozzuk létre az összes DATA memóriában található változót. Míg a laborok alatt konkrét címekkel dolgoztunk (például 50H), fekete övesként illet nem tesszünk. Mindig egy külön relokálható szegmenst hozunk létre a változóknak is, melyeknek egyenként nevet is fogunk adni. Így sokkal kevesebb fejfájással tudunk kódolni, mivel a változó címét a linker majd kitalálja, nekünk elég

csak a nevét használni.

A többfájlos program esetében fontos, hogy csak azokat a külső szimbólumokat adjuk hozzá a fájlokhoz, melyeket el is szeretnénk érni. A DATA memóriában létrehozott változóknál csak a CONFIG állományra van szükségünk, hiszen egyedül a feltételes fordításhoz szükséges szimbólumok értékét szeretnénk tudni. Nincs értelme hozzáadni a REG552, ALIASES vagy bármely másik include állományt, illetve nem is szabad, hiszen még véletlenül sem szeretnénk elérni például a szubrutinokat egy tisztán változókat definiáló fájlból!

```
24      $INCLUDE (CONFIG.INC)
25  $IF   (ASSEMBLE_VARIABLES = 1)
26  $IF   (ASSEMBLE_DATA_VARIABLES = 1)
27
28  ;VARIABLES FOR DATA MEMORY SPACE
29      DATA_SEG SEGMENT DATA
30      RSEG     DATA_SEG
31      PUBLIC   D_VAR
32      PUBLIC   D_TO_OVF_EXTEND
33      PUBLIC   D_TO_OVF_HALF_SEC
34
35  D_VAR:                DS      1
36  D_TO_OVF_EXTEND:     DS      1
37  D_TO_OVF_HALF_SEC: DS      1
38
39  $ENDIF
40  $ENDIF
41      END
```

78. ábra. Változók létrehozása és exportálása

A fájl elején most nem egy, hanem két szimbólum értékéhez kötött a feltételes fordítás. A DATA változók fordítását külön is ki-be kapcsolhatjuk, de globálisan is engedélyezhetjük az összes típusú változó (XDATA, IDATA, DATA, CODE, BIT) fordítását. Ezután létrehozunk egy újabb relokálható szegmenst, viszont most nem a CODE, hanem a DATA területen! A PUBLIC direktívákkal felsoroljuk és exportáljuk az összes változó szimbólumát (nevét) melyet ez a fájl tartalmaz, majd utána létre is hozzuk őket. A DS direktíva használatával lehet a szimbólumok számára helyt foglalni a memóriában. Most mindegyik DATA változónak 1-1 bájtot foglalunk. D_VAR: DS 1 → a D_VAR nevű szimbólumnak foglalj le egy bájt területet.

Észrevehetted, hogy míg a szubrutinoknál minden szimbólum neve S__

előtaggal szerepelt, úgy most a DATA változók D_ prefixet kaptak. Ez csupán egy kódolási irányelv, mert így pontosan tudjuk milyen szimbólum milyen típusú, nem kell össze-vissza keresgelnünk. Ha csinálunk egy szimbólumot mely például a CNT_T0 névre hallgat, akkor gőzünk sincs, hogy az most milyen memóriacímet reprezentál. Jelenthet IDATA, XDATA, CODE, BIT és DATA memóriát is. Ha viszont D_CNT_T0 névvel illetjük, akkor egyből tudjuk, hogy ez egy változó a DATA területen. Hasonlóan a DATA memória változóihoz, a többi területet is adott prefix jelöl, melyek az alábbiak. Természetesen mindenki saját jelölésrendszert használ, ez csak egy megoldás a sok közül.

Előtag	Memória típusa
D_	DATA
I_	IDATA
B_	BIT
C_	CODE
DBA_	DATA BITADDRESSABLE
X_	XDATA
_	DBA_ változó bitjei

Ahhoz, hogy ezeket a szimbólumokat más fájlokból elérhessük, az export mellé import direktívákat is használnunk kell! **Nyisd meg az EXTERNAL_DATA.INC állományt!** Ebben a fájlban semmit sem hozunk létre, csupán felsoroljuk az importálandó DATA szimbólumok neveit.

```

23 #ifndef EXTERNAL_DATA_H
24 #define EXTERNAL_DATA_H
25
26     $INCLUDE (CONFIG.INC)
27
28     $IF (ASSEMBLE_VARIABLES = 1)
29     $IF (ASSEMBLE_DATA_VARIABLES = 1)
30
31         EXTRN    DATA    (D_VAR)
32         EXTRN    DATA    (D_TO_OVF_EXTEND)
33         EXTRN    DATA    (D_TO_OVF_HALF_SEC)
34
35     $ENDIF
36 $ENDIF
37
38 #endif

```

79. ábra. Szimbólumok importálása

Az importálás az **EXTRN** direktívával lehetséges. Itt a fordítónak azt jelezzük, hogy lesz egy külső szimbólumunk valahol, amit keressen meg, és úgy fordítsa a programot, hogy ezt is vegye figyelembe. Természetesen a **PUBLIC** és **EXTRN** direktívákban megadott szimbólumok neveinek egyeznie kell, másképp nem fog fordulni a program.

Az összes többi szimbólum importálását is hasonló fájlokban soroljuk fel, melyeket egy nagy fájlban összesítünk. Az **EXTERNALS.INC** állományban **\$INCLUDE** direktívákkal mindegyik import fájlt behúzzunk, és innentől elég csak ezt hozzáadni azon forrásokhoz, melyekben használni szeretnénk a külső szimbólumokat. A szubrutinok forráskódjában kivételesen nem adtuk hozzá ezt az egész fájlt, hanem egyesével soroltuk fel az állományokat, hiszen nem adhatjuk úgy hozzá a szubrutin szimbólumok neveit egy forráshoz, hogy még csak abban a fájlban hozzuk létre őket. Ezt forward reference-nek hívják, és kiakad tőle az assembler.

Az **#ifndef EXTERNALS_H** nem valódi assembler direktíva. Ezt a C preprocesszor használja, de mivel a μ Vision ezt tudja, ezért ASM fájlknál is felhasználható. Ez a direktíva teszteli, hogy lett-e már definiálva az **EXTERNALS_H** szimbólum korábban. Amennyiben nem, úgy definiáljuk, és csak ebben az esetben adjuk hozzá az összes include fájlt az adott programhoz. Így, ha valahol már egyszer behúztuk az **EXTERNALS.INC** állomány tartalmát, akkor egy esetleges újbóli hozzáadásnál nem akadna ki a fordító. Ugyanis, ha már egyszer definiálva lett az **EXTERNALS_H** szimbólum, onnantól kezdve minden újabb import során figyelmen kívül hagyjuk a fájl tartalmát. Ezeket include guardoknak hívjuk, melyeket kötelező jelleggel használunk többfájlos program esetében.

```
23 #ifndef EXTERNALS_H
24 #define EXTERNALS_H
25
26     $INCLUDE (EXTERNAL_SUBROUTINES.INC)
27     $INCLUDE (EXTERNAL_DATA.INC)
28     $INCLUDE (EXTERNAL_IDATA.INC)
29     $INCLUDE (EXTERNAL_XDATA.INC)
30     $INCLUDE (EXTERNAL_BIT_VARIABLES.INC)
31     $INCLUDE (EXTERNAL_BITADDRESSABLE_VARIABLES.INC)
32     $INCLUDE (EXTERNAL_CONSTANTS.INC)
33     $INCLUDE (EXTERNAL_STACK_REFERENCE.INC)
34
35 #endif
```

80. ábra. **EXTERNALS.INC** : Az összes külső szimbólum felsorolása

Nézzünk is rá egy olyan fájlra, melyben használjuk a külső szimbólumokat! Nyisd meg az **ISR_TIMER0_OVF.A51** forrást!

```

24      $NOMOD51
25      $INCLUDE (ALIASES.INC)
26      $INCLUDE (CONFIG.INC)
27      $INCLUDE (REG552.INC)
28      $INCLUDE (MACROS.INC)
29      $INCLUDE (EXTERNALS.INC)
30
31      $IF      (ASSEMBLE_ISR_ALL = 1)
32      $IF      (ASSEMBLE_ISR_TIMER0_OVF = 1)
33
34      IF IN_PIR_LAB <> 1
35          CSEG      AT      000BH
36      ELSE
37          CSEG      AT      400BH
38      ENDIF
39      LJMP      ISR_VECT_TIMER_0_OVERFLOW
40      ISR_TIMER_0 SEGMENT CODE
41      RSEG      ISR_TIMER_0
42      ISR_VECT_TIMER_0_OVERFLOW:
43          USING    0
44          DJNZ     D_TO_OVF_EXTEND,ISR_VECT_TIMER0_OVERFLOW_END
45          MOV      D_TO_OVF_EXTEND,#36
46          ;Below this line, everything gets executed exactly every 10ms
47          ;If and only if a clock frequency of 11.0592MHz is used
48          LCALL    S_WRITE_XDATA_PERIPH_LEDS_FROM_ACC
49          LCALL    S_READ_XDATA_PERIPH_BUTTONS_INTO_ACC
50          DJNZ     D_TO_OVF_HALF_SEC,ISR_VECT_TIMER0_OVERFLOW_END
51          MOV      D_TO_OVF_HALF_SEC,#50
52          ;Below this line, everything gets executed exactly every 0.5 sec
53          ;If and only if a clock frequency of 11.0592MHz is used
54          LCALL    S_CHASER_ON_P1
55          CPL      B_TO_OVF_SEC_EXTEND_BIT
56          JNB      B_TO_OVF_SEC_EXTEND_BIT,ISR_VECT_TIMER0_OVERFLOW_END
57          ;Below this line, everything gets executed exactly every 1 sec
58          ;If and only if a clock frequency of 11.0592MHz is used
59          LCALL    S_COUNTER_ON_P2
60      ISR_VECT_TIMER0_OVERFLOW_END:
61          RETI
62      $ENDIF
63      $ENDIF
64      END

```

81. ábra. T0 megszakítási rutinja

Az eddig megismert dolgokat mind-mind felhasználjuk ebben a fájlban. Az **\$INCLUDE** direktívákkal hozzáadjuk a külső szimbólumokat, az **\$IF** direktívákkal pedig feltételesen fordítjuk a fájl tartalmát. A vektortábla ebben a projektben nem a főprogramban, hanem külön-külön az ISR rutinoknál lett megírva. Ezeket abszolút szegmenskezeléssel lehet megoldani, pontosan úgy, ahogyan laboron tanultuk. Csak egy különbség van most. A vektortáblát vagy a 0000H címtől, vagy a 4000H címtől kezdve definiáljuk.

Erre azért van szükség, mert a Programozható Irányítások (PIR) laborban a mikrokontrolleren egy feltöltő és debug (monitor) program található. Ez a program juttatja fel a mikrokontrollerre a mi általunk írt programot⁹

Amennyiben szimulátort használunk, úgy a vektortáblát a kódmemória elején kell elhelyezni, úgy ahogyan a harmadik laboron tanultuk. Ha viszont a mikrokontrolleres gyakorlón tesztelnénk a programot, úgy az általunk írt program nem 0000H, hanem 4000H-tól fog kezdődni, mivel a kódmemória elején található a monitor program. Emiatt a vektortáblát is el kell tolnunk 0-ról 4000H-ra. Ezt egy szimbólum értékével tudjuk beállítani, amit a CONFIG.INC állományban tudunk megváltoztatni. Ha az IN_PIR_LAB értéke nem egyenlő 1, úgy a T0 időzítő vektora a 000BH, különben 400BH címen található.

Érdemes megjegyezni, hogy a vektorok változatlanul a hardveres címen találhatók, azaz a 0003H–0023H területre indít LCALL hívást a mikrokontroller, viszont a monitor átirányítja a hívást a felhasználói programhoz (vektor címe + 4000H).

A vektor címén található ugrás abszolút szegmenssel való deklarációja után, magának az ISR rutinnak már relokálható szegmenst nyitunk. Egy újabb direktívával, a **USING**-gal kiválasztjuk, hogy melyik kontextusban (regiszterbankban) szeretnénk dolgozni a rutinon belül. Ezzel nem váltunk még bankot, csupán jelezzük a fordítónak, hogy amennyiben az **ARn: Absolute Register n** szimbólumot használjuk mint operandus, akkor az melyik bankot címezze. A laborok során nem tértünk ki arra, hogyan lehet az R0 – R7 regisztereket kezelni a PUSH és POP utasításokkal. Mivel négy bank létezik, és nincs külön PUSH/POP Rn utasítás, ráadásul a stack kezelő utasítások direkt memóriacímet várnak, ezért ott az AR szimbólumokat kell használjuk. Az AR0 az R0 regiszter címét hordozza attól függően, hogy

⁹Az első laboron azt olvashattad, hogy a kódmemóriát nem tudjuk írni, mert nincs is rá utasítás. Ez így is van, viszont a CODE és XDATA memória egyetlen AND kapuval összefűzhető, ezáltal Harvard helyett Neumann architektúrát tudunk kialakítani. Így nem lesz külön XDATA és külön kódmemória, hanem a kettő egy és ugyanaz. A MOVX utasítással írni és olvasni is, míg a MOVC utasítással csak olvasni tudjuk a memóriát. A feltöltés során a uVision UART-on küldi el a program Intel HEX formátumú gépi kódját, amit a monitor program dekódol, és beleírja a gépi kódot a Neumann memóriába a MOVX utasítással. A programszámláló változatlan módon címzi ezt a memóriát. Bővebben: https://www.keil.com/support/man/docs/mon51/mon51_intro.htm

milyen számot írtunk előtte a USING direktíva operandusába. Ez 0, 1, 2 és 3 lehet. Ha nem használtuk a direktívát, akkor automatikusan a 0-ás bankban található címeket fogja beilleszteni a fordító.

USING	AR0	AR1	AR2	AR3	AR4	AR5	AR6	AR7
0	00H	01H	02H	03H	04H	05H	06H	07H
1	08H	09H	0AH	0BH	0CH	0DH	0EH	0FH
2	10H	11H	12H	13H	14H	15H	16H	17H
3	18H	19H	1AH	1BH	1CH	1DH	1EH	1FH

Innentől kezdve csak a kilométer hosszú szubrutin és változó neveket kell kibogozni, és nagyon egyszerűen megfejthető a rutin működése. Az utolsó laboron ugyanezt a feladatot oldottuk meg, csak kevésbé beszédesebb módon. Ott az Rn regisztereket, valamint az F0 bitet használtuk a kiterjesztéshez, most viszont létrehoztunk erre külön-külön változókat. Így pontosan tudjuk mit is csinál a változó, nem fogjuk véletlenségből átírni máshol a tartalmát. A D_ előtagos változókat a DATA_VARIABLES.A51, míg a B_ prefixes változókat a BIT_VARIABLES.A51 fájlban, a szubrutinokat pedig a SUB-ROUTINES.A51 fájlban találhatjuk meg.

Kissé feleslegesnek tűnhet, hogy ennyire szétszedtük a programot, de amint megszokja az ember, hogy így sokkal áttekinthetőbb és logikusabb egy program felépítése, valamint rettentő egyszerűen lehet nagyobb módosításokat elvégezni a kódon, onnantól vissza sem akar nézni a mindent is dobjunk bele egy fájlba alapú programozáshoz.

A következőekben nézzük bele a STARTUP fájlba, melyet az [Arm Ltd.](#) szolgáltat a C51 compiler mellé. Ebben a fájlban tudjuk inicializálni a memória tartalmát miután elindul a mikrokontroller. Erre azért van szükség, mivel a RAM (az SFR terület nem!!) véletlenszerű értékekkel töltődik fel bekapcsolás után, amit célszerű kinullázni, hogy minden rekeszben egy ismert érték legyen indulás után. Természetesen van olyan alkalmazási terület, ahol ezt nem szeretnénk megcsinálni, így a CONFIG.INC állományból kikapcsolható a STARTUP.A51 fordítása is. Ilyenkor csak a Stack Pointer inicializálása, és a LJMP ugrás a főprogramra ami a startup fájl részét képezi, a memória inicializálása mintha ott sem lenne.

A STARTUP.A51 fájl végén egy ugrás található a ?C_START címkére. Ezt a címkét a MAIN.A51 fájlban, a főprogramban találjuk. Nézzük meg,

hogyan néz ki a főprogram!

A már megismert módon hozzáadjuk a forráshoz a használt külső szimbólumokat, makrókat, configokat, és fedőneveket. Utána a **NAME** direktívával elnevezzük a programrészlet nevét, melynek gyakorlati haszna nincs, de jól néz ki. Ha nem használjuk ezt a direktívát, akkor a programrészlet neve a fájl neve lesz. Ez a főprogram esetén MAIN lenne. Az elnevezés után a **PUBLIC** direktívával exportáljuk a ?C_START szimbólumot, hogy a STARTUP.A51 fájl is ismerje, hiszen enélkül nem tudnánk elugrani a főprogramra. A USING-gal beállítjuk, hogy a nullás regiszterbankot címezzük az ARn szimbólumokkal, melyeket egy-egy PUSH-POP utasítással tesztelünk is. Az ALIASES.INC fájlban deklarálva lett, hogy az RnBk szimbólumokkal egyesével is címezni lehessen egy adott bank adott regiszterét. Ezt is teszteljük egy-egy PUSH és POP utasítással. Ezek után már ismerős elemek jönnek. Szubrutinokat hívunk, változókat érünk el, valamint az időzítőt és a soros portot konfiguráljuk. Az egyetlen újdonság az M_SLEEP makró meghívása.

```
24      $NOMOD51
25      $INCLUDE(REG552.INC)
26      $INCLUDE(CONFIG.INC)
27      $INCLUDE(ALIASES.INC)
28      $INCLUDE(MACROS.INC)
29      $INCLUDE(EXTERNALS.INC) ;contains every external symbol
30
31      NAME    BLACK_BELT_TEMPLATE_CODE
32
33
34      ;Starting address of program can be modified in CONFIG.INC
35      PUBLIC ?C_START
36      IF IN_PIR_LAB <> 1
37          CSEG    AT      0100H
38      ELSE
39          CSEG    AT      4100H
40      ENDIF
41      P_USER_PROGRAM_START:
42      ?C_START:
43          USING    0      ;Select register bank for use with <ARn> : Absolute Register n
44          PUSH     AR0     ;pushes R0 from the selected register bank
45          POP      AR0     ;PUSH and POP instructions use direct addressing respectively
46          PUSH     ROB0    ;ROB0 is a new alias for R0 of Bank0
47          POP      R7B3
48
49          LCALL    S_INIT_SERIAL_PORT      ;Definition of subroutines can be found inside SUBROUTINES.A51
50          LCALL    S_INIT_TIMER_0
51          MOV      DPTR,#C_LOREM_IPSUM
52          LCALL    S_SERIAL_WRITE_TEXT_AT_DPTR
53          LCALL    S_GET_NEXT_INSTR_PC_VALUE_IN_DPTR
54
55          MOV      DPTR,#X_VAR
56          MOVX     A,@DPTR ;ACC now contains whatever was inside X_VAR
57
58          MOV      D_TO_OVF_HALF_SEC,#50
59
60          MOV      P1,#1
61          LCALL    S_INIT_INTERRUPT_SYSTEM
62          MOV      TLO,#0FAH ;To speed up the occurrence of the first interrupt
63      END_OF_PROGRAM:
64          M_SLEEP ;Definition of macros can be found inside MACROS.INC
65          LJMPL   END_OF_PROGRAM
66
67      END
```

82. ábra. A főprogram

A makrók segítségével rövidített (illetve parametrizálható) módon illeszthetünk be utasításokat a programba. A makró nem szubrutin, azaz nincs LCALL és RET. Pillants rá a MACROS.INC fájl tartalmára!

```

23
24 #ifndef MACROS_H
25 #define MACROS_H
26
27     M_FILL_RAM_WITH_ACC MACRO
28     MOV     R0,#0FFH
29     MOV     @R0,A
30     DJNZ    R0,$-1
31     MOV     00H,A
32     ENDM
33
34     M_SLEEP    MACRO
35     MOV     PCON,#01H
36     ENDM
37
38
39 #endif

```

83. ábra. Makrók létrehozása

Két sablon makrót definiáltunk. Az M_FILL_RAM_WITH_ACC feltölti az egész RAM tartalmát az Akkumulátor értékével, míg az M_SLEEP makró elküldi alvó állapotba a mikrokontrollert. Bárhol a sablon projektben, ahol hozzáadjuk a forráshoz a MACROS.INC állományt az \$INCLUDE direktívával, ott használhatjuk ezeket a rövidítéseket. Vannak olyan utasítássorozatok, melyeket nem lehet megoldani szubrutinokkal. A RAM feltöltése egy uniform értékkel például pontosan ilyen. Mivel az LCALL és RET használatához a stack integritása is szükséges, a 8051-ben pedig nem lehet a verem átírása nélkül feltölteni a teljes memóriát, így szubrutin hívása helyett makrót definiálunk erre a feladatra. Természetesen egy makrón belül minden további nélkül lehet szubrutint hívni, ha szeretnénk.

Utolsó lépésként pedig, ismerjük meg a sokszor emlegetett CONFIG.INC állományt! Ebben a fájlban lehet beállítani az összes feltételes fordítással kapcsolatos szimbólum értékét, valamint a startup fájlban használt utasítások paramétereit. Továbbá itt adható meg mekkora területet foglaljunk le a stack számára az IDATA területen.

Az IN_PIR_LAB szimbólum értékével (0 vagy 1) dönthető el, hova linkeljük a megszakítási vektortáblát, valamint a főprogramot (0000H vagy 4000H). A Target 1 beállításainál konstans módon lett megadva a 4000H

mint kódmemória kezdőcím, emiatt a relokálható szegmensek is garantáltan e cím felett kezdődnek majd. Ez amiatt jó, hogy garantáltan olyan címre legózza be a linker a szegmenseket, mely a gyakorló panelen is elérhető a felhasználói program által. A **\$SET** direktívákkal lehet beállítani a feltételes fordításokat. A változókra, szubrutinokra, ISR rutinokra külön-külön, valamint csoportonként 1-1 globális kapcsoló is létezik. Az I2C_IO_LIB egy könyvtár, melyben az I²C (egy kommunikációs protokoll) perifériát kezelő rutinok lettek megírva. Ezen rutinoknak csak a neveit tudjuk, melyeket az EXTERNAL_SUBROUTINES.INC állományban találhatunk, de a belső működésüket nem ismerjük¹⁰.

Írd át a config fájlt, úgy, hogy az IN_PIR_LAB szimbólum értéke 0 legyen! Fordítsd le a programot, majd indítsd el a szimulációt! Léptess a programon, majd figyeld meg mikor, melyik fájlra ugrik a szimulátor! Ha szeretnéd, lefordíthatod a programot max configgal is. Ilyenkor minden feltételes fordítást vezérlő szimbólum értéke 1, azaz mindent is fordítunk.

Próbálj megírni ebben a környezetben pár programot, melyet a laborok során megoldottunk!

Nem tértünk ki a sablon projekt minden egyes kis részletére, de az itt elolvasottak alapján, már át lehet látni mit miért csináltunk, és mi hol található. Akit még ennél is bővebben érdekel a terület, annak nagyon hasznosak az alábbi linkek:

A51 Assembler

https://www.keil.com/support/man/docs/a51/a51_intro.htm

MON51 monitor program

<https://www.keil.com/support/man/docs/mon51/>

C51 C fordító

https://www.keil.com/support/man/docs/c51/c51_intro.htm

μVision IDE felhasználói kézikönyv

<https://www.keil.com/support/man/docs/uv4cl/>

8051-es utasításkészlete

<https://www.keil.com/support/man/docs/is51/>

¹⁰Kis okoskodással természetesen visszafejthető ez is debuggolás közben a disassembly ablakból


```

24 #ifndef CONFIG_H
25 #define CONFIG_H
26
27 IN_PIR_LAB      EQU      0
28
29 $SET            (ASSEMBLE_STARTUP                = 1)
30 $SET            (ASSEMBLE_SUBROUTINES            = 1)
31 $SET            (ASSEMBLE_I2C_IO_LIB              = 0)
32 $SET            (ASSEMBLE_VARIABLES              = 1)
33 $SET            (ASSEMBLE_ISR_ALL                = 1)
34
35
36 $SET            (ASSEMBLE_ISR_T2_OVF              = 0)
37 $SET            (ASSEMBLE_ISR_T2_COMPARE_0        = 0)
38 $SET            (ASSEMBLE_ISR_T2_COMPARE_1        = 0)
39 $SET            (ASSEMBLE_ISR_T2_COMPARE_2        = 0)
40 $SET            (ASSEMBLE_ISR_T2_CAPTURE_0        = 0)
41 $SET            (ASSEMBLE_ISR_T2_CAPTURE_1        = 0)
42 $SET            (ASSEMBLE_ISR_T2_CAPTURE_2        = 0)
43 $SET            (ASSEMBLE_ISR_T2_CAPTURE_3        = 0)
44
45 $SET            (ASSEMBLE_ISR_ADC                 = 0)
46
47 $SET            (ASSEMBLE_ISR_SERIAL              = 0)
48
49 $SET            (ASSEMBLE_ISR_TIMER1_OVF          = 0)
50
51 $SET            (ASSEMBLE_ISR_EXTERNAL1           = 0)
52
53 $SET            (ASSEMBLE_ISR_TIMER0_OVF          = 1)
54
55 $SET            (ASSEMBLE_ISR_EXTERNAL0           = 0)
56
57
58 $SET            (ASSEMBLE_BIT_VARIABLES           = 1)
59 $SET            (ASSEMBLE_CONSTANTS               = 1)
60 $SET            (ASSEMBLE_DATA_VARIABLES          = 1)
61 $SET            (ASSEMBLE_IDATA_VARIABLES         = 1)
62 $SET            (ASSEMBLE_BITADDRESSABLE_VARIABLES = 1)
63 $SET            (ASSEMBLE_XDATA_VARIABLES         = 1)
64
65 STACKLEN1      EQU      20
66
67 IDATALEN        EQU      80H
68
69 XDATASTART      EQU      0000H
70 XDATALEN        EQU      3E00H
71
72 XDATA_PERIPH    EQU      0C000H
73 XDATA_PERIPHLEN EQU      4000H
74
75 PDATASTART      EQU      0H
76 PDATALEN        EQU      0H
77
78 IBPSTACK        EQU      0
79 IBPSTACKTOP     EQU      0xFF +1
80
81 XBPSTACK        EQU      0
82 XBPSTACKTOP     EQU      0xFFFF +1
83
84 PBPSTACK        EQU      0
85 PBPSTACKTOP     EQU      0xFF +1
86
87 PPAGEENABLE     EQU      0
88 PPAGE           EQU      0
89 PPAGE_SFR       DATA    0A0H
90 #endif

```

84. ábra. Konfigurációs fájl

A 8051 MIKROKONTROLLER CSALÁD UTASÍTÁSKÉSZLETE

Utasítás	Adatátviteli utasítások	bájt	cikl
MOV A,Rn	Regisztert mozgat az A-ba	1	1
MOV A,direct	Direkt bájtot mozgat az A-ba	2	1
MOV A,@Ri	Indirekt RAMot mozgat az A-ba	1	1
MOV A,#data	Adatot mozgat az A-ba	2	1
MOV Rn,A	A-t mozgat a regiszterbe	1	1
MOV Rn,direct	Direkt bájtot mozgat a Rn-be	2	2
MOV Rn,#data	Adatot mozgat a regiszterbe	2	1
MOV direct,A	A-t mozgat egy direct bájtba	2	1
MOV direct,Rn	Rn-et mozgat egy direct bájtba		
MOV direct,direct	Direkt bájtot mozg. direkt bájtba	3	2
MOV direct,@Ri	Indirekt RAM-ot direct bájtba	2	2
MOV direct,#data	Adatot direct bájtba	3	2
MOV @Ri,A	A-t indirect RAM-ba	1	1
MOV @Ri,direct	Direkt bájtot indirect RAM-ba	2	2
MOV @Ri,#data	Adatot indirect RAM-ba	2	1
MOV DPTR,#data16	Adatmutató feltöltése 16bit-tel	3	2
MOVC A,@A+DPTR	A-ba a prog.mem-ből(eltolt cím)s	1	2
MOVC A,@A+PC	A-ba a prog.mem-ből(eltolt cím)	1	2
MOVX A,@Ri	A-ba a külső RAM-ból(cím 8 bit)	1	2
MOVX A,DPTR	A-ba a külső RAM-ból(cím16bit)	1	2
MOVX @Ri,A	A-t a külső RAM-ba(cím 8 bit)	1	2
MOVX @DPTR,A	A-t a külső RAM-ba(cím 16 bit)	1	2
PUSH direct	Egy bájt behelyezése a zsákba	2	2
POP direct	Egy bájt kivétele a zsákból	2	2
XCH A,Rn	Regiszter és A cseréje	1	1
XCH A,direct	Egy direct bájt és A cseréje	2	1
XCH A,@Ri	Indirekt RAM és A cseréje	1	1
XCHD A,@Ri	Indirekt RAM és A alsó 4 bitének cseréje	1	1

Utasítás	Aritmetikai utasítások	bájt	cikl
ADD A,Rn	A=A+regiszter	1	1
ADD A,direct	A=A+direct bájt	2	1
ADD A,@Ri	A=A+indirect RAM	1	1
ADD A,#data	A=A+adat	2	1
ADDC A,Rn	A=A+regiszter+Cy	1	1
ADDC A,direct	A=A+direct bájt+Cy	2	1
ADDC A,@Ri	A=A+indirect RAM+Cy	1	1
ADDC A,#data	A=A+adat+Cy	2	1
SUBB A,Rn	A=A-regiszter-Cy	1	1
SUBB A,direct	A=A-direct bájt-CY	2	1
SUBB A,@Ri	A=A-indirekt RAM-Cy	1	1
SUBB A,#data	A=A-adat-Cy	2	1
INC A	A=A+1	1	1
INC Rn	Regiszter=Regiszter+1	1	1
INC direct	Direkt bájt=direkt bájt+1	2	1
INC @Ri	Indirekt RAM = indirekt RAM+1	1	1
INC DPTR	DPTR = DPTR+1	1	2
DEC A	A=A-1	1	1
DEC Rn	Regiszter=Regiszter-1	1	1
DEC direct	Direkt bájt=direkt bájt-1	2	1
DEC @Ri	Indirekt RAM = indirekt RAM-1	1	1
MUL AB	BA=A*B	1	4
DIV AB	A=A/B maradék B-ben	1	4
DA A	ACC decimális korrekciója	1	1

Utasítás	Logikai utasítások	bájt	cikl
ANL A,Rn	A= A.ÉS.regiszter	1	1
AN A,direct	A= A.ÉS.direkt bájt	2	1
ANL A,@Ri	A= A.ÉS.indirekt RAM	1	1
ANL A,#data	A= A.ÉS.adat	2	1
ANL direct,A	Direkt bájt= A.ÉS.direkt bájt	2	1
ANL direct,#data	Direkt bájt= direkt bájt.ÉS.adat	3	2
ORL A,Rn	A= A.VAGY.regiszter	1	1
ORL A,direct	A= A.VAGY.direkt bájt	2	1
ORL A,@Ri	A= A.VAGY.indirekt RAM	1	1
ORL A,#data	A= A.VAGY.adat	2	1
ORL direct,A	Direkt bájt= A.VAGY.direkt bájt	2	1
ORL direct,#data	Direkt bájt= direkt bájt.VAGY.adat	3	2
XRL A,Rn	A= A.KIZÁRÓ VAGY.regiszter	1	1
XRL A,direct	A= A. KIZÁRÓ VAGY.direkt bájt	2	1
XRL A,@Ri	A= A. KIZÁRÓ VAGY.indirekt RAM	1	1
XRL A,#data	A= A. KIZÁRÓ VAGY.adat	2	1

Utasítás	Logikai utasítások (folytatás)	bájt	cikl
XRL direct,A	Direkt bájt= direkt bájt.KIZÁRÓVAGY.A	2	1
XRL direct,#data	Direktbájt= direktbájt.KIZÁRÓVAGY.adat		
CLR A	Törli az A-t	1	1
CPL A	Komplementálja az A-t	1	1
RL A	Balra forgatja az A-t	1	1
RLC A	Balra forgatja az A-t a Cy-n keresztül	1	1
RR A	Jobbra forgatja az A-t	1	1
RRC A	Jobbra forgatja az A-t a Cy-n keresztül	1	1
SWAP A	Felcseréli az A alsó és felső 4 bit-ét	1	1

Utasítás	Bit-manipulációs utasítások	bájt	cikl
CLR C	Törli a Cy-t	1	1
CLR bit	Törli a bitet	2	1
SETB C	Egybe állítja a Cy-t	1	1
SETB bit	Egybe állítja a bitet	2	1
CPL C	Komplementálja a Cy-t	1	1
CPL bit	Komplementálja a bitet	2	1
ANL C,bit	Cy=Cy.ÉS.bit	2	2
ANL C,/bit	Cy=Cy.ÉS.bit negáltja	2	2
ORL C,bit	Cy=Cy.VAGY.bit	2	2
ORL C,/bit	Cy=Cy.VAGY.bit negáltja	2	2
MOV C,bit	Cy=bit	2	1
MOV bit,C	Bit=Cy	2	2

Utasítás	Program és gépi vezérlő utasítások	bájt	cikl
ACALL addr11	Szubrutin hívás 11 bites cím esetén	2	2
LCALL addr16	Szubrutin hívás 16 bites cím esetén	3	2
RET	Visszatérés a szubrutinból	1	2
RETI	Visszatérés megszakítás kiszolgáló rutinból	1	2
AJMP addr11	Feltétel nélküli ugrás 11 bites címre	2	2
LJMP addr16	Feltétel nélküli ugrás 16 bites címre	3	2
SJMP rel	Feltétel nélküli ugrás közelre	2	2
JMP @A+DPTR	Feltétel nélküli ugrás indirekt címre	1	2
JZ rel	Ugrás, ha A=0	2	2
JNZ rel	Ugrás, ha A nem 0	2	2
JC rel	Ugrás, ha a Cy=1	2	2
JNC rel	Ugrás, ha a Cy=0	2	2
JB bit,rel	Ugrás, ha a bit=1	3	2
JNB bit,rel	Ugrás, ha a bit=0	3	2
JBC bit,rel	Ugrás, ha a bit=1 és törölje a bitet	3	2
CJNE A,direct,rel	Összehasonlítja az A-t és a címet és ugrik, ha nem egyenlő	3	2
CJNE A,#data,rel	Összehasonlítja az A-t és az adatot és ugrik ha nem egyenlő	3	2
CJNE Rn,#data,rel	Összehasonlítja a regisztert és az adatot és ugrik ha nem egyenlő	3	2
CJNE @Ri,#data,rel	Összehasonlítja az indirekt RAM-ot és az adatot és ugrik ha nem egyenlő	3	2
DJNZ Rn,rel	Csökkenti az Rn tartalmát 1-el és ugrik, ha Rn nem egyenlő 0-val	2	2
DJNZ direct,rel	Csökkenti az direct bájt tartalmát 1-el és ugrik ha nem egyenlő 0-val	3	2
NOP	Üres utasítás	1	1

READ-MODIFY-WRITE utasítások

Utasítás	Példa	Utasítás	Példa
ANL port	ANL P2,A	DEC port	DEC P1
ORL port	ORL P1,A	DJNZ port	DJNZ P1,rel
XRL port	XRL P1,A	MOV portbit,C	MOV P2.1,C
JBC porbit	JBC P2.2,rel	CLR portbit	CLR P1.0
CPL portbit	CPL P1.1	SETB portbit	SETB P1.0
INC port	INC P1		

A jelzőbiteket (flag-eket) állító utasítások

Utasítás	CY	OV	AC	Utasítás	CY	OV	AC
ADD	X	X	X	SETB C	1		
ADDC	X	X	X	CLR C	0		
SUBB	X	X	X	CPL C	X		
MUL AB	0	X		ANL C,bit	X		
DIV AB	0	X		ANL C,/bit	X		
DA A	X			ORL C,bit	X		
RRC A	X			ORL C,/bit	X		
RLC A	X			MOV C,bit	X		
				CJNE	X		