11. Mikrovezérlők II.

Írta: Tóth János

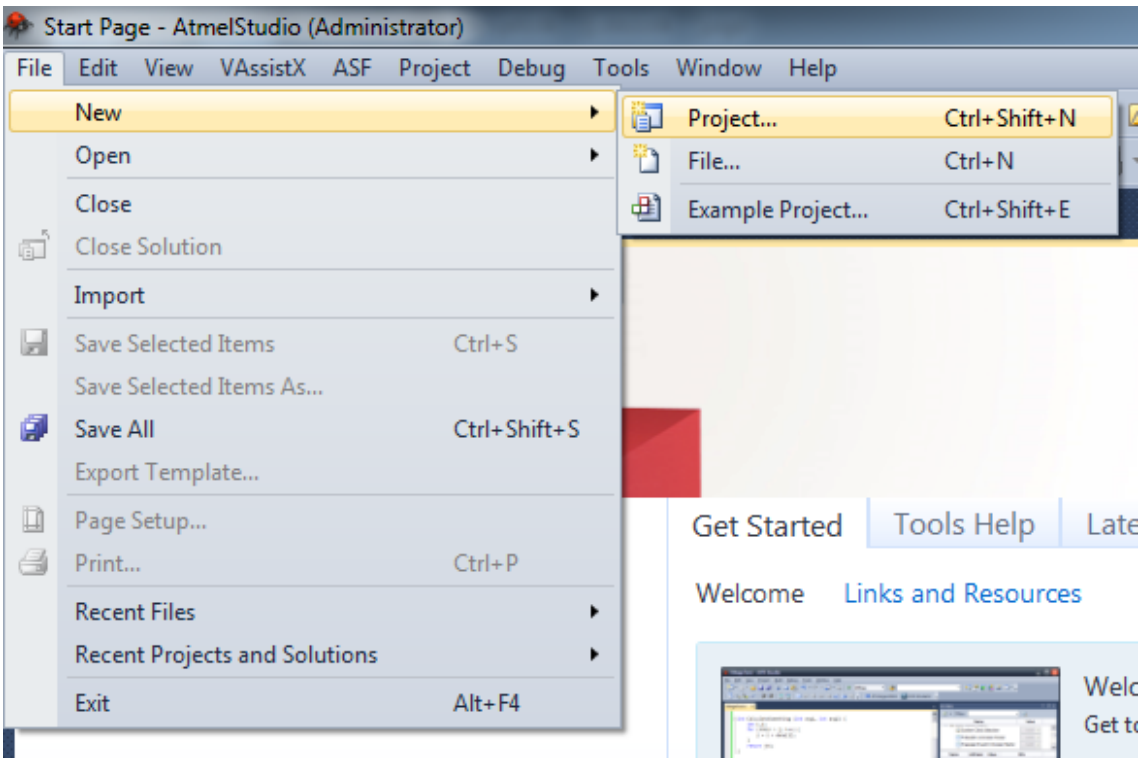
Lektorálta: Proksa Gábor

Ezen tananyagrész végére érve képesek lesztek megvalósítani K.I.T.T. futófényét a Knight Rider című sorozatból.

FEJLESZTŐKÖRNYEZET BEMUTATÁSA

PROJECT LÉTREHOZÁSA

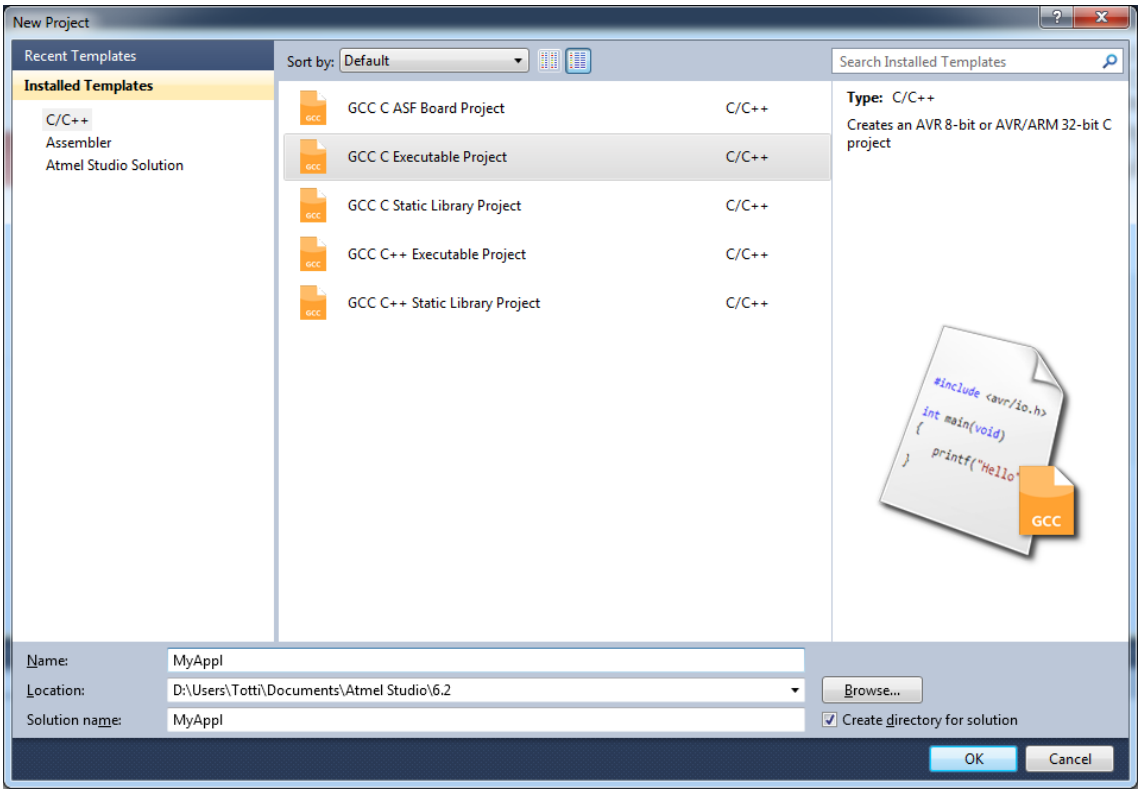
Az előző részekben feltelepítettük a fejlesztőkörnyezetet, most kicsit ismerkedjünk meg vele jobban, hozzunk létre egy saját projektet. Ezt több helyről is elérhetjük, számomra a legegyszerűbb a fájl menüből. Kiválasztjuk a *New* menüpontot majd a *Project*-et.



***1. ábra** - Új projekt létrehozása menüpont*

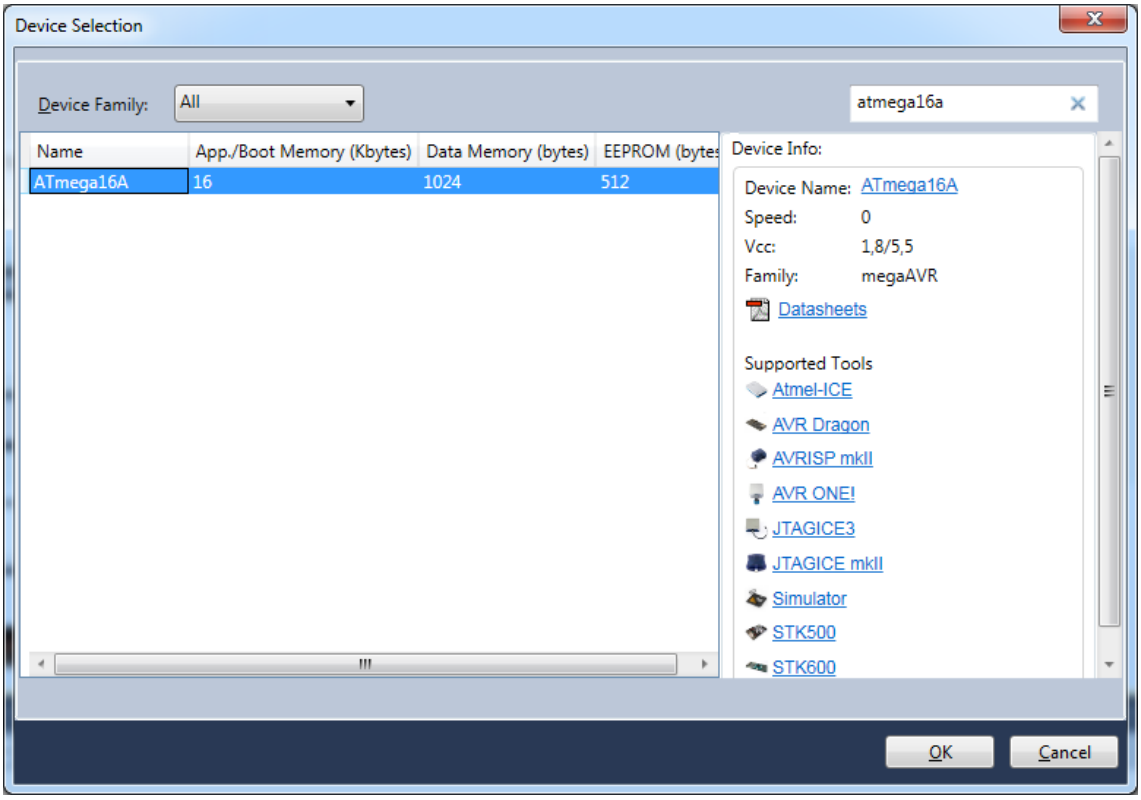
Ezután a projekt típusát kell megadnunk. Választhatunk C/C++ projektek közül, vagy akár assembly projektet is. Mi most válasszuk a C/C++-t, és azon belül is a "*GCC C Executable Project*"-et. Ezzel megadtuk, hogy C nyelven szeretnénk írni a kódunkat és azt, hogy a projekt futtatható (executable) legyen. A többi opcióval egyelőre nem foglalkozunk, azokhoz jóval mélyebb ismeretre lenne szükség. Adjunk nevet a projektnek az ablak alján található *Name* mezőben, én a MyAppl nevet választottam. A *Location* felirat

mellett megadhatunk egy saját könyvtárat, ahova szeretnénk menteni a projektet, de az alapértelmezett könyvtár is tökéletes. Az *OK* gombra kattintva lépjünk tovább.

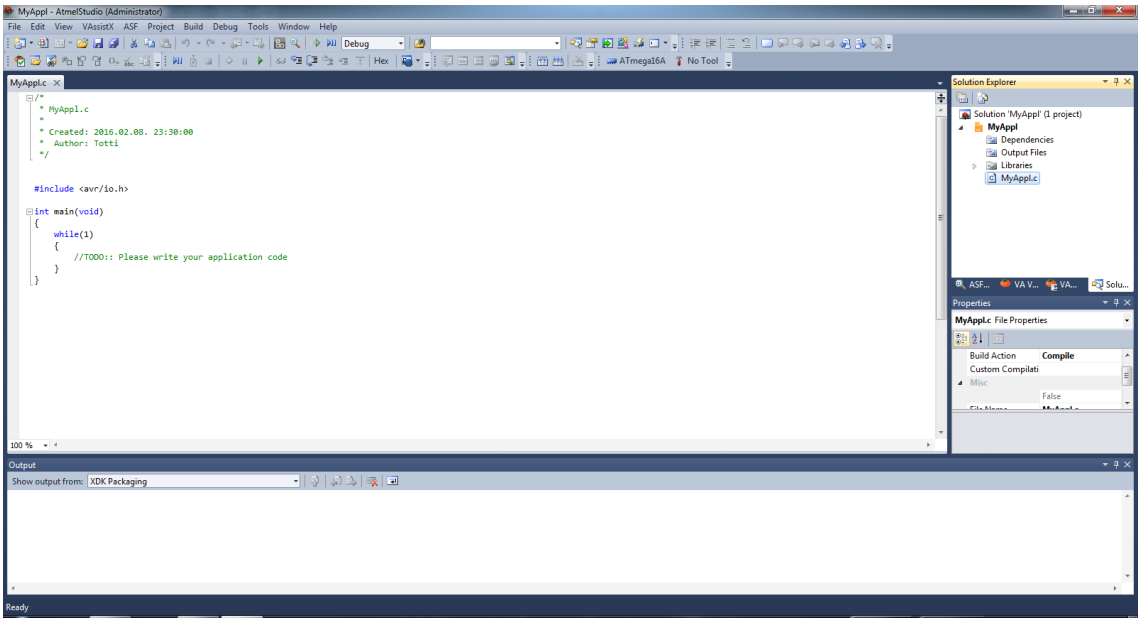


***2. ábra** - Projekt típusának a megadása*

A következő ablakban ki kell választanunk a mikrovezérlő típusát. Ezt legegyszerűbben a jobb felső sarokban lévő kereső mezővel tehetjük meg. Jelen esetben az *ATmega16A* kulcsszóra keressünk rá. Természetesen, ha más mikrovezérlőt használunk, akkor azt válasszuk ki. Az Eszköz információ (*Device Info*) alatt megtalálhatjuk a vezérlő adatlapját, illetve a támogatott programozó eszközöket is. Az *OK* gomb megnyomásával már létre is jön a projektünk.



3. ábra *- ATmega16A kiválasztása*



4. ábra *- Elkészült projekt*

A projekt létrehozásakor a fejlesztőkörnyezet létrehoz egy C fájlt a projekt nevével. Ez a fájl tartalmaz egy main függvényt, azon belül pedig egy végtelen `while(1){}` ciklust. Ez után el is kezdhetjük a fejlesztést a mikrovezérlőnkre.

FUSE BITEK JELENTÉSE ÉS ÁLLÍTÁSA

Mielőtt megismerkednénk a FUSE bitekkel, előtte tisztáznunk kell egy nagyon fontos fogalmat, amivel biztos fogtok találkozni a beágyazott fejlesztés során. Ez nem más, mint a regiszter. A regisztereket adat tárolására használjuk, melyek segítségével tudjuk a mikrovezérlőnk perifériáit konfigurálni, vagy esetleg egy perifériából adatokat kiolvasni. Egy regiszter nagyságát, azaz méretét általában bitekben adják meg, ami lehet 8, 16 vagy 32 bit (esetleg 64 bit az újabb mikrovezérlőkben). Egy regisztert úgy tudunk a legkönnyebben elképzelni, mint egy polcot, amit felosztunk a hosszának megfelelően egyenlő részekre. Minden egyes kis részbe be tudunk rakni vagy esetleg onnan kivenni valamit. Ezek a kis polcrészek jelentik az egyes biteket a mikrovezérlőnél. A következő ábrán a PORTA regiszter adatait láthatjuk a mikrovezérlő adatlapjából. Ez egy 8 bites regiszter, melynek minden egyes bitjét tudjuk írni és olvasni (Read/Write). Előfordulhat olyan regiszter is, amelynek egyes bitjeit csak olvasni tudjuk. A kezdeti érték (Initial Value) a mikrovezérlő bekapcsolásakor felvett értéket jelenti. Itt minden bit 0 értékű lesz.

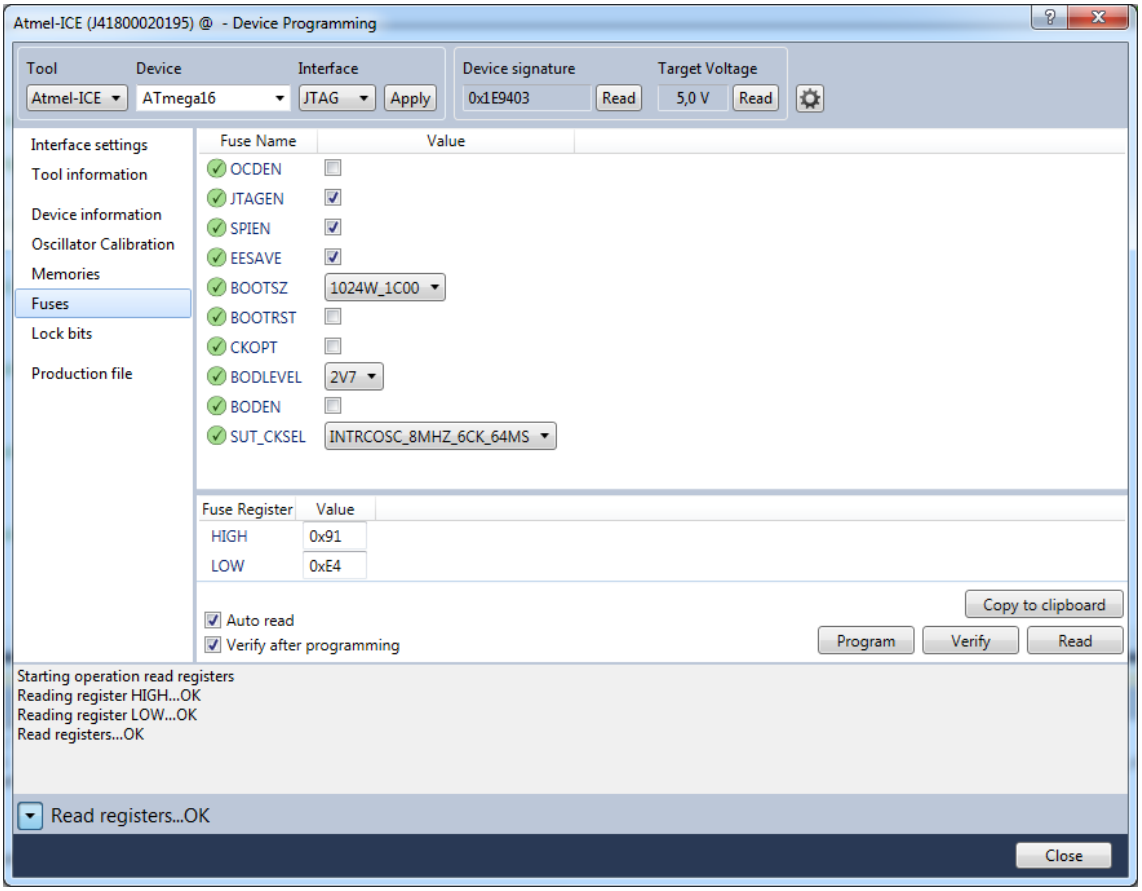
PORTA – Port A Data Register

Bit	7	6	5	4	3	2	1	0	
	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	PORTA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

***5. ábra** - Port A regiszter felépítése*

Már említettem, hogy a regiszterekben adatokat tárolunk. A mikrovezérlőben az adatok tárolására memóriát használunk. Ezekből már sejthető, hogy az egyes regiszterek a memória kis darabkái. Ahhoz, hogy ezeket a memória darabkákat elérhessük, szükségünk van azok címére, azaz arra, hogy hol találhatóak pontosan a teljes memórián belül. Hasonlóan a postásnak is szüksége van az utca, házszám, emelet, ajtó információkra, a településnév közlése vajmi keveset ér. Minden egyes regiszternek fix címe van. Ezeket a címeket elég nehéz lenne megjegyezni, valamint egy másik mikrovezérlőnél előfordulhat, hogy az egyes regiszterek teljesen más címre kerülnek. Ahhoz, hogy ilyenkor ne kelljen a teljes programot újraírni, és reménykedni, hogy minden egyes címet jól írtunk át, egy kis segítséghez folyamodhatunk. Mennyivel egyszerűbb, ha csak azt kell leírni, hogy a PORTA regiszter értéke legyen 0, ahelyett, hogy keresgetnénk a PORTA regiszter címét az adatlapban, majd erre a címre írnánk be a 0 értéket. A mikrovezérlőkhöz, a gyártók általában mellékelnek egy header fájlt, ahol megtalálhatóak az egyes regiszterek nevei és azok címei. Ezt a header fájlt tudjuk használni arra, hogy név szerint hivatkozzunk a regiszterekre és ne kelljen címeket keresgetni.

A regiszterek után térjünk is át a FUSE bitekre. Meglepő vagy sem de a FUSE bitek is egy regiszterben találhatóak, mégpedig a FUSE regiszterben. A FUSE bitek segítségével tudjuk a mikrovezérlő legfontosabb beállítását elvégezni. Vigyázat, egy rosszul beállított FUSE bit, akár teljesen megszüntetheti a kapcsolatot a felhasználó és a mikrovezérlő között, ilyen esetben nem fogjuk tudni programozni a mikrovezérlőnket. A lenti ábrán az ATmega 16 FUSE bitei láthatóak.



***6. ábra** - FUSE regiszter beállítása*

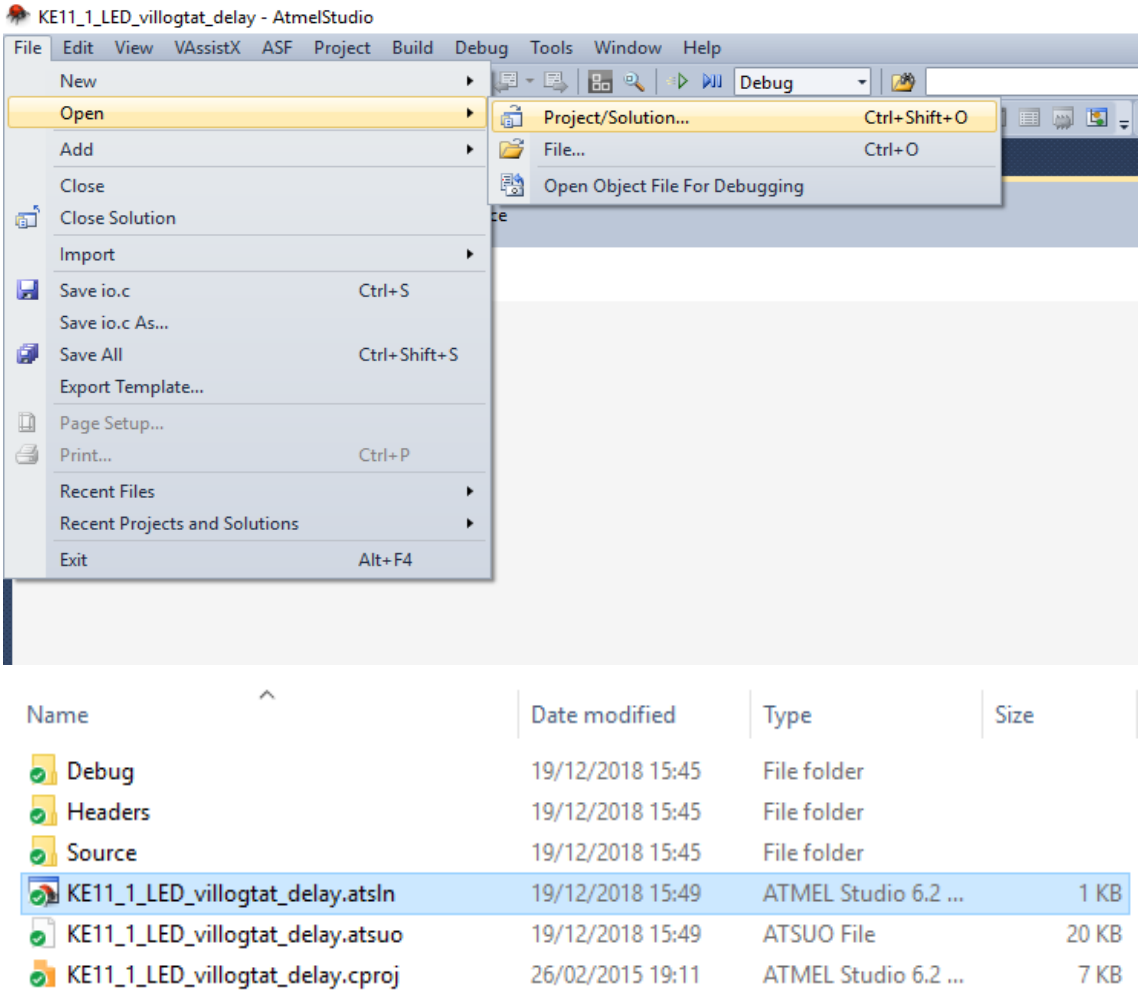
Az egyes bitek leírása megtalálható az adatlapban, most csak azokra térnék ki, amikre a későbbiekben is szükségünk lehet. Az egyik legfontosabb bit, a SUT_CKSEL, ami nem is egy bit, hanem 6 bitet tartalmazó beállítási lehetőség. Itt tudjátok beállítani, hogy a mikrovezérlő milyen oszcillátorról üzemel. A mi példánkban ez a beépített 8MHz-es belső oszcillátor, de a későbbiekben építhettek külső pontosabb oszcillátorral ellátott kapcsolást, ilyen esetben itt tudjátok kiválasztani a használt oszcillátor órajelét. Ez a beállítás nagyon fontos, hiszen ha rossz értéket adtok meg, akkor az időzítések el fognak csúszni, az egyes perifériák nem fognak működni megfelelően.

A beállítások elmentéséhez a *Program* gombra kell kattintani, ilyenkor a FUSE regiszter értékei elmentődnek a mikrovezérlőben, és a következő indítás során már ezeket az értékeket használja a mikrovezérlő.

LED VILLOGTATÁS

Megismerkedtünk a fejlesztőkörnyezettel és a FUSE bitekkel. Ezután nézzük meg, hogy a 9-es tananyagrészben megépített kapcsolás pontosan miért is működik.

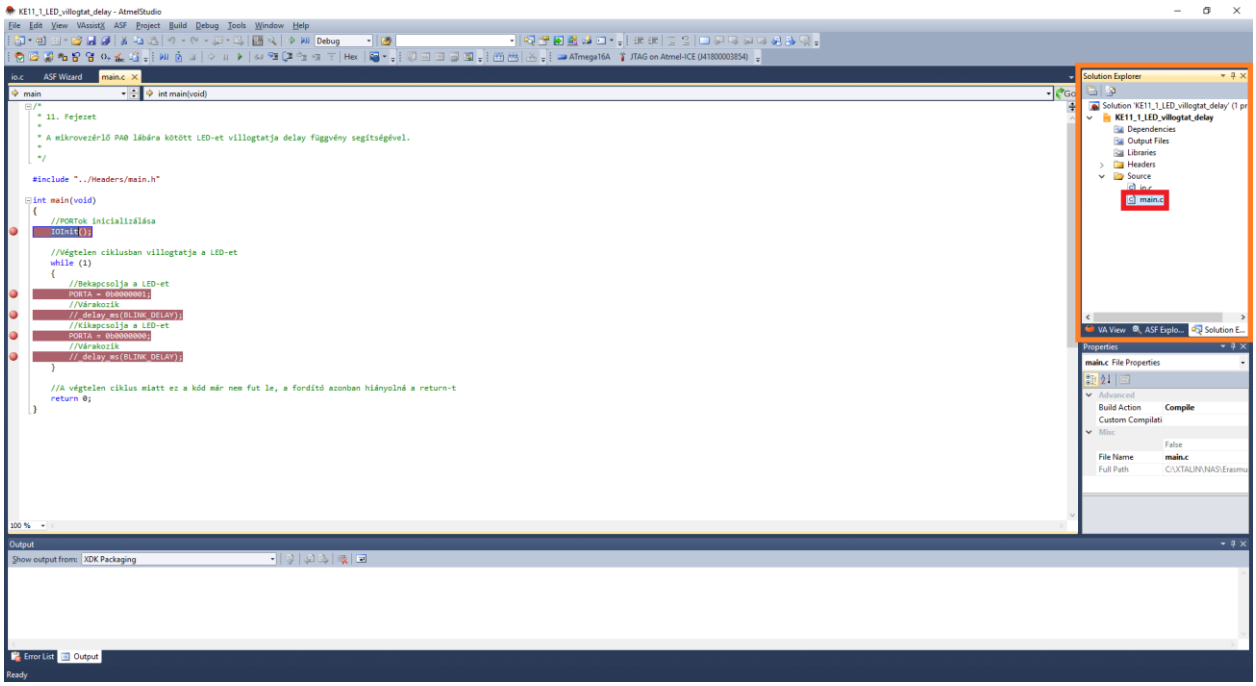
Töltsük be a *KE11_1_LED_villogtat_delay* projektet a fejlesztőkörnyezetbe. Ezt *a File → Open → Project/Solution* menüpont alatt érhetjük el, majd a felugró ablakban kiválasztjuk az adott projektet.



***7. ábra** - Projekt megnyitása*

Ezt követően a *Solution Explorer* ablakban (jobb felül) láthatjuk a projekthez tartozó fájlokat, nyissuk meg a *main.c* fájt. Ebben a fájlban van a *main* függvény, az itt található utasításokat fogja sorban végrehajtani a mikrovezérlő bekapcsolás után.

A legelső utasítás egy másik függvény meghívása. Ez az `IOInit()` függvény. Nézzük meg miért van szükség erre. A 9-es tananyagrészben volt már szó arról, hogy a PA0...7, PB0...7, PC0...7, PD0...7 lábak mindegyike lehet digitális bemenet és digitális kimenet. Ahhoz, hogy a LED világítson, logikai magas értéket kell kiadnunk a PA0 lábon, ami azt jelenti, hogy egy belső kapcsolóeszköz összeköti a kiválasztott lábat a tápfeszültséggel. Ha logikai alacsony érték van a PA0 lábon, akkor a láb a földpotenciállal van összekötve, így a LED nem fog világítani.



8. ábra *- main.c fájl*

Szintén a 9-es részben volt szó arról, ha az egyik lábon valamilyen logikai értéket szeretnénk kiadni, akkor digitális kimenetnek kell beállítani az adott lábat. Erre a DDR nevű regiszter lesz a segítségünkre. DDR a Data Direction Register angol szó rövidítése, amit ha lefordítunk magyarra azt kapjuk, hogy adat irány regiszter. Az adatlapban megnézve, minden portnak van ilyen regisztere, például az A port regiszterét a DDRA névvel tudjuk elérni. Most nézzük meg, hogy pontosan milyen értékeket tudunk ebben a regiszterben állítani.

Az adatlap szerint ha a regiszter adott bitjének az értéke 1 akkor az adott láb digitális kimenetként fog működni. Tehát ahhoz, hogy a PA0-s láb digitális kimenet legyen a DDRA regiszterben a nulladik helyen álló bitet 1-re kell állítani. Ezt úgy tehetjük meg a legegyszerűbben, ha bináris számként adjuk meg a regiszter értékét. Bináris ábrázolás esetén minden bit értéket 0-val vagy 1-el ábrázoljuk. Kattintsunk az `IOInit()` függvényre jobb egérgombbal, majd a legördülő menüből válasszuk a *Goto Implementation* lehetőséget (vagy használjuk az Alt+G billentyűparancsot). Ekkor a fejlesztőkörnyezet megkeresi, hogy hol van a függvény definíciója, és felkínálja a találatokat. Válasszuk a felső opciót (*io.c*), és megtekinthetjük, hogy mit is csinál pontosan ez a függvény. Nézzük az alábbi sort:

```
DDRA = 0b00000001;
```

Itt állítja be a DDRA regiszter értékét a függvény. A 0b prefix segítségével mondjuk meg a fordítónak, hogy az utána következő számsort binárisan értelmezze. A példakódban láthatjuk, hogy 8 darab szám követi, ez 8 bitnek felel meg. A legelső helyen a legmagasabb helyiértékű bit található, jelen esetben a 7. bit, ez megegyezik azzal, ahogyan a tízes számrendszerbeli számokat írjuk. A példakódban látható hogy csak a legutolsó szám 1-es a többi 0, tehát a 0. bitet a regiszterben 1-re állítja az összes többit 0-ra.

Ez a fajta regiszter állítási mód akkor ajánlatos, ha egyszerre szeretnénk több bitet módosítani az adott regiszterben. Ha csak 1 bitet kell állítani, ajánlott előre megírt függvények használata. Ilyen függvények az

sbi (setbit) és cbi (clearbit) nevű függvények. Ezek a függvények a compat/deprecated.h fájlban vannak deklarálva. A használatukhoz `#include` direktívát kell alkalmazni az adott c fájlban. Jelen esetben a következő sort kell beírni `#include "compat/deprecated.h"` a fájlunk elejére. A függvények használata a következő: egy adott bit beállítása 1-re egy regiszterben

```
sbi(regiszter, bitpozíció);
```

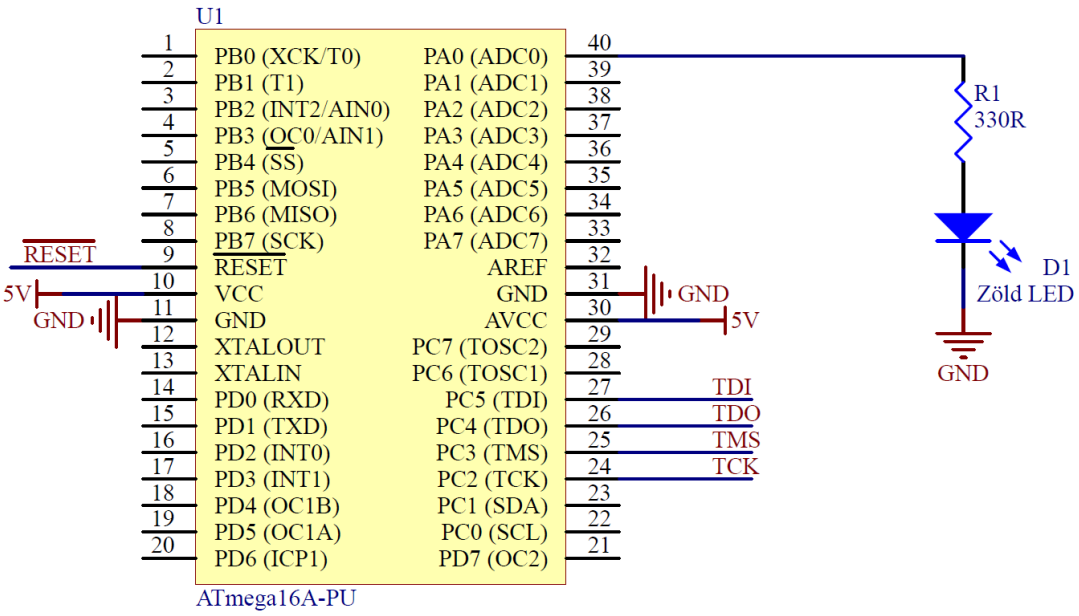
A mi példánkban a DDRA regiszter 0. bitjének a beállítása 1-re a következő utasítással történik: `sbi(DDRA,0);`. Ha 0-ba szeretnék állítani egy bitet, akkor a cbi függvényt használjuk. Példaként állítsuk vissza a DDRA 0. bitjét 0-ba, ezt a következő utasítással megtehetjük:

```
cbi(DDRA, 0);
```

Nézzük meg, még milyen regiszterek tartoznak az egyes portokhoz. A DDRx regisztereken kívül vannak még PORTx és PINx regiszterek. A PORTx regiszter kétféle funkciót lát el, attól függően, hogy milyen az adott pin iránya. Ha kimenetként van beállítva egy adott pin, akkor a PORTx regiszter megfelelő bitjének az állításával tudjuk logikai magas vagy logikai alacsony értékre állítani a hozzá rendelt lábat. Ha 1-est írunk az adott bitpozícióba, akkor a hozzá rendelt pin logikai magas értéket fog felvenni, ha pedig 0-t akkor logikai alacsonyt. Másik lehetőség, hogy az adott pin bemenetnek van állítva, ilyenkor a PORTx regiszter segítségével bekapcsolhatjuk a belső felhúzó ellenállást az adott pinhez. A PINx regiszter segítségével az adott pin állapotát tudjuk lekérdezni. Úgy tekintsük a regiszterre, mint egy változóra. Jelen esetben a PINx regiszter 8 bites. Az egyes bitpozíciók megfelelnek az adott port egy-egy pinjeinek. Például ha a PINB értéke 0b00000001, akkor a 0. Pin (PB0) logikai magas értékén van, a többi pedig logikai alacsonyon. Ha a PIND értéke 0b01001010, akkor a 6., 3. és 1. pin (PD6, PD3, PD1) logikai magas értékén van, a többi (PD7, PD5, PD4, PD2 és PD0) pedig logikai alacsonyon.

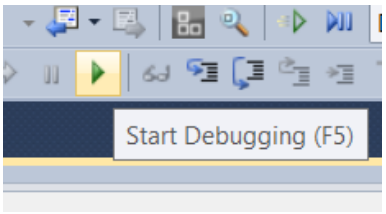
Az `IOInit()` függvényben tovább szemlélődve láthatjuk, hogy az összes porton kikapcsoljuk a felhúzó ellenállást, valamint a PA0-s pin kivételével az összes pint bemenetnek állítjuk. Ebben a függvényben tehát a portok és pinek inicializálása történik.

Nézzük tovább a *main.c* fájlt. Az inicializálás után következik egy végtelen `while(1){}` ciklus. A végtelen ciklusban lévő utasítások mindaddig végrehajtódnak, amíg újra nem indítjuk a vezérlőt vagy be nem következik valamilyen hiba. A ciklus első utasítása beállítja a PA0-s lábat logikai magas értékre, a PORTA regiszteren keresztül. A regiszter működését már ismerjük az előző bekezdésből. Az utasítás lefutása után a lábra kötött led világítani fog. A LED villogásához nem elég, ha bekapcsoljuk, ki is kell kapcsolni majd megint vissza, és így tovább. Ebből következik, hogy a bekapcsolás után a cikluson belül szükséges egy kikapcsolás is, amit, mint már tudjuk, a PORTA regiszter 0. bitjének 0-s értékre történő állításával tudunk megtenni. Építsük meg a 9-es tananyagrészben már kipróbált kapcsolást.



9. ábra *- egy LED villogtatásának kapcsolási rajza*

Ezt követően töltsük rá a példakódot változatlan formában a mikrovezérlőre a *Start Debugging* gombra kattintva.



10. ábra *- Program feltöltése (emlékeztető)*

Láthatjuk, hogy ha a ciklusban csak ez a két utasítás szerepel, nem fog villogni a LED, hanem folyamatosan világít (pontosabban mi nem fogjuk látni, hogy villog). De vajon miért? Az előző részekben már volt róla szó, hogy a mikrovezérlő minden egyes órajelre végrehajt egy utasítást. A mikrovezérlő órajele 8MHz. Ez azt jelenti, hogy másodpercenként 8 millió utasítást hajt végre. Jelen esetben a két utasítást kell végrehajtani folyamatosan, így másodpercenként 4 milliószor kapcsoljuk ki a LED-et majd vissza. Ezt a nagyon gyors változást nem tudjuk érzékelni. Valahogy késleltetni kellene a ki- és bekapcsolás utasításokat egymáshoz képest. Kicsit gondolkozzunk, mit is tudunk csinálni. Hogyan lehet bizonyos ideig várakoztatni a mikrovezérlőt? Erről egy későbbi bekezdésben majd részletesebben értekezünk. Első körben használjuk az előre megírt `_delay_ms()` függvényt, amit ha megpróbálunk magyarra fordítani akkor a *késleltetés ezredmásodpercben* kifejezést kapjuk, a paraméterként megadott ideig fogja várakoztatni a vezérlőt. Ezt a függvényt használjuk a két utasítás hívása után, így mind a bekapcsolás mind a kikapcsolás után várni fogunk egy bizonyos ideig. A példakódunkban legyen a késleltetés értéke 500, tehát fél másodpercenként kapcsoljuk ki majd be a LED-et. Írjuk be a `_delay_ms(500);` sorokat a LED bekapcsolása és kikapcsolása után:

```
	while (1)
	{
		//Bekapcsolja a LED-et
		PORTA = 0b0000001;
		//Várakozik
		_delay_ms(500);
		//Kikapcsolja a LED-et
		PORTA = 0b0000000;
		//Várakozik
		_delay_ms(500);
	}
```

Ezzel megvalósítottuk, hogy másodpercenként egyszer villogjon a LED. Töltsük rá a kódot a vezérlőre, és ellenőrizzük a működést.

DELAY VS ÓRAJEL SZÁMLÁLÁS

Az előző bekezdésben volt szó arról, hogy tudjuk várakoztatni a mikrovezérlőt a `_delay_ms` függvény segítségével. Van-e lehetőség máshogy elérni amit szeretnénk? Nézzük meg mit is tanultunk eddig. A mikrovezérlő minden egyes órajel ciklusban végrehajt egy utasítást. Ezen kívül ismerjük az órajel frekvenciát is. Ezekből már ki lehet számolni, hogy egy adott idő alatt, hány utasítás hajtódik végre. Tehát ha fogunk egy változót és egy ciklusban folyamatosan növeljük, akkor kapunk egy számlálót, ami minden órajel ciklusra lép egyet. Nincs más dolgunk, mint addig léptetni a számlálót míg el nem ér egy általunk előre kiszámolt értéket.

Egy példa: A mikrovezérlő órajele 8MHz, tehát 8 millió utasítás 1 másodperc alatt. Legyen egy változónk aminek az értékét addig növeljük egy ciklusban, míg el nem éri a 8 milliót. Ebben az esetben, ha az értéknövelés egy utasítás alatt végrehajtódik, akkor eltelt 1 másodperc. Ezzel a módszerrel mi is megvalósítottunk egy késleltető algoritmust. Ez teljesen tökéletes amikor nem törekszünk nagyon pontos időzítésre, csak úgy közelítőleg szeretnék valamilyen késleltetést beállítani a programban. A `_delay_ms` függvény ezzel szemben nagyon pontos időzítést biztosít. Egy példafüggvény:

```
void _sajat_delay_ms(unsigned int delay)
{
	//belső változók
	static unsigned int counter;
	static unsigned int time;

	//nullázzuk ki az eltelt időt
	time = 0;
	//addig amíg nem érjük el a kért időt
	while(delay > time)
	{
		//minden ciklus elején nullázzuk a számlálót
		counter = 0;
		//növeljük a számláló értékét 8000-ig
		//ezzel elvileg 1 ezredmásodpercet várunk
```

```
            while(counter <= 8000)
            {
                  counter++;
            }
            //növeljük az eltelt idő értékét
            time++;
      }
}
```

Másoljuk be ezt a függvényt az `int main(void)` függvény elé, majd cseréljük ki a `_delay_ms(500)` függvényhívást a `_sajat_delay_ms(500)` függvényünkre.

```
      while (1)
      {
            //Bekapcsolja a LED-et
            PORTA = 0b0000001;
            //Várakozik
            _sajat_delay_ms(500);
            //Kikapcsolja a LED-et
            PORTA = 0b0000000;
            //Várakozik
            _sajat_delay_ms(500);
      }
```

A kód feltöltése után láthatjuk, hogy a LED villog, de sokkal lassabban, mint ahogy vártuk, egy periódus nagyjából 6-7 másodperc alatt zajlik le 1 helyett. Ennek az az oka, hogy nem csak a `counter++` számít műveletnek a processzor számára, a műveletek sorrendje sokkal inkább így néz ki:

1. `counter` változó értékének beolvasása a memóriából a műveleti regiszterbe
2. `8000`, mint érték beírása a műveleti regiszterbe
3. összehasonlító (`<=`) művelet végrehajtása
4. előző művelet eredménye szerint ugrás a `counter++` vagy a `time++` programsorra
5. `counter` vagy `time` változó értékének növelése (inkrementálás)
6. `counter` vagy `time` változó értékének írása a memóriába
7. ugrás a ciklus elejére

Nem is olyan egyszerű kiszámolni, hogy pontosan mennyi ideig is fog tartani mire lefut egy ilyen késleltető ciklus, ezért precíz időzítésre, mint már említettük, alkalmatlan. Ha a 8000 helyére 1195-öt írunk, akkor egész elfogadható lesz a késleltetés, de még mindig nem pontos.

Érdemes megjegyezni a példa kapcsán, hogy a változók esetlegesen túlcsordulhatnak. Tudjuk, hogy a változóinkat véges méretű memóriahelyen tároljuk, így nem lehetnek tetszőlegesen nagyok. Egy 8 bites előjel nélküli változó (`char`) 0-tól 255-ig képes értéket felvenni. Miért jelenthet ez gondot? Nézzük meg a kettes számrendszerben ábrázolva, hogy mi is történik túlcsorduláskor. Azért ez a legszemléletesebb, mert a processzor is ebben a számrendszerben dolgozik. Vegyük a 8 biten tárolt 178 decimális számot, ennek a 0b10110010 bináris szám felel meg. Adjunk hozzá a 8 biten tárolt 150-et (0b10010110).

```
  10110010
+ 10010110
-----------
 101001000
```

A végeredmény már nem fér el 8 bit szélességen, viszont az eredményt is 8 biten tároljuk, így a legnagyobb helyiértékű bit nem fér bele a kimeneti változóba, szaknyelven túlcsordul (mint a víz a pohárban, ha többet akarunk beletölteni, mint amekkora a pohár). Ekkor az processzor által kiszámolt eredmény a várt 328 (0b**101001000**) helyett 72 (0b**01001000**) lesz. Ez óriási gondot tud jelenteni bármilyen matematikai műveletnél, ezért a változók értéktartományára nagyon oda kell figyelnünk.

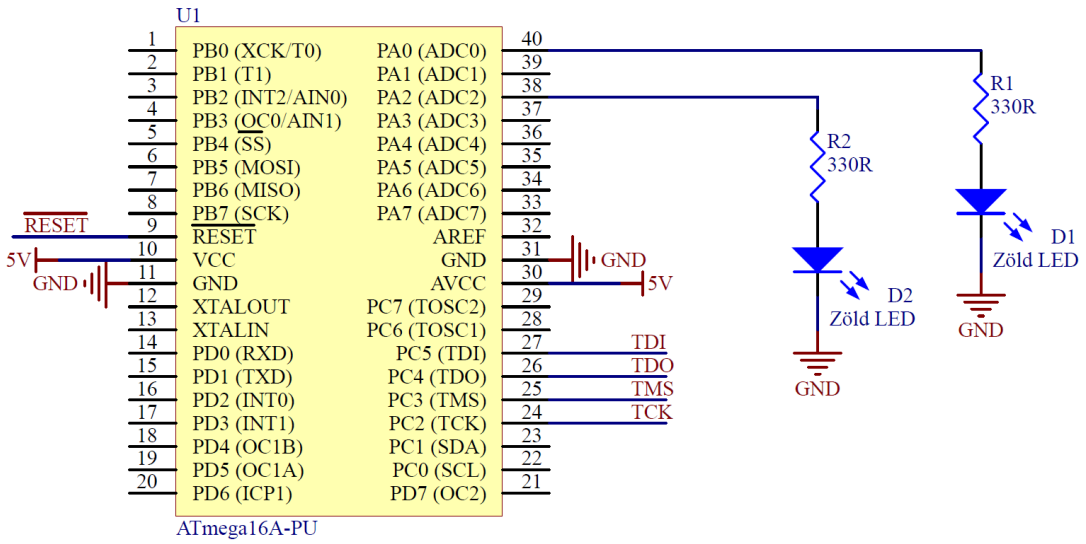
Változó típus	Méret	Értéktartomány
`char`	8 bit	**0**-tól **255**-ig vagy **-128**-tól **127**-ig
`unsigned char`	8 bit	**0**-tól **255**-ig
`signed char`	8 bit	**-128**-tól **127**-ig
`int`	16 bit	**-32768**-tól **32767**-ig
`unsigned int`	16 bit	**0**-tól **65535**-ig
`long`	32 bit	**-2147483648**-tól **2147483647**-ig
`unsigned long`	32 bit	**0**-tól **4294967295**-ig

Gondoljunk bele, hogy a példában addig növeljük a változót, míg el nem éri a 8000-t (vagy az 1195-öt). Ha a számlálóm egy 8 bites változó lenne, akkor egészen 255-ig semmi probléma nem adódna, minden ciklusban nőne az értéke. Azonban a 255-höz hozzáadva 1-et nem 256-ot, hanem újra 0-t kapnánk. Így soha nem érné el a számláló a 256 vagy annál nagyobb értéket, ezzel végtelen ciklusba kerülne a program.

LED VILLOGTATÁS 2 LED-EL

Az előző bekezdésekben megnéztük, hogyan is tudunk 1 darab LED-et villogtatni a mikrovezérlőnk segítségével. Ennek kapcsán megismertük, hogyan tudjuk kimenetként konfigurálni az egyes lábakat, majd hogyan tudjuk bekapcsolni és kikapcsolni a lábra kötött LED-et. Ezen kívül megvizsgáltuk, hogyan tudjuk időzíteni a kapcsolásokat egymáshoz képest.

Ebben a részben kicsit bonyolítjuk a dolgokat, már nem egy darab LED-et szeretnénk villogtatni, hanem kettőt. Először a kapcsolást kell összeállítani. Most a korábbi kapcsolást kell kicsit módosítani, nem csak a PA0-ás lábra (ide van bekötve az első LED), hanem a PA2-es lábra is kössünk be egy LED-et.



11. ábra *- két LED villogtatásának kapcsolási rajza*

Ezek után módosítanunk kell a korábbi kódunkat. Ehhez először nyissuk meg a *KE11_2_LED_villogtat_delay* projektet, hogy megmaradjon a korábbi példánk is. Emlékezzünk vissza mit is csináltunk akkor:

1. Az adott lábat kimenetként kell konfigurálni
2. Az adott lábon a felhúzó ellenállást kikapcsolni
3. Megfelelő időpontban változtatni a láb állapotát

A lábak konfigurációja, azaz az `IOInit()` függvény az *io.c* fájlban található meg, így először itt végezzük el a módosítást. Már tudjuk, hogy a DDR regiszterek segítségével lehet az pinek irányát beállítani. Jelen esetben a PA2-es lábat szeretnénk kimenetként konfigurálni. A példakódunkban a teljes DDRA regisztert egy utasítással állítjuk, így célszerű ezt módosítani, vagy minden további bit állítást a `cbi()` és `sbi()` függvények segítségével, az utasítás után rakni. Mivel binárisan adtuk meg a regiszter értékét, így egyszerűen csak a 2. bit pozícióját kell átírnunk 1-re. Tehát az utasítás a következő lesz:

```
DDRA = 0b00000101;
```

További módosításokra ebben a fájlban nincs is szükségünk. Amennyiben másik port egyik lábára kötöttük volna a LED-et, mondjuk a PB5-re, akkor értelemszerűen a DDRB regiszter 5. bitjét kellett volna átállítanunk.

Ezzel végeztünk a konfigurálással, jöhet az adott láb vezérlése. Ezt már a *main.c* fáljban fogjuk megtenni. Ugyanúgy, mint egy LED villogtatásakor, itt is a végtelen ciklusban fogjuk módosítani az állapotokat. Jelen pillanatban a teljes portot állítjuk az állapotváltások során, de ha már több lábat szeretnénk függetlenül vezérelni, nem ez a legjobb megoldás. Használjuk a már megismert `sbi()` és `cbi()` függvényeket. Így garantált, hogy mindig csak az adott láb állapotát fogjuk változtatni.

Első körben villogtassuk a LED-eket felváltva, amíg az egyik világít, addig a másik nem. Az egyes állapotváltások között eltelt idő pedig legyen 1 másodperc, tehát minden másodpercben másik led fog világítani. A kiindulási állapot legyen az, hogy először a PA0-s lábra kötött led világít, majd 1 másodperc

múlva a PA2-es lábra kötött. Ehhez a végtelen ciklus legelső utasítása a `sbi(PORTA,0);` míg a második utasítás a `cbi(PORTA,2);` Ezután, mint már tudjuk, várakoznunk kell egy adott ideig, jelen esetben 1 másodpercet. Tehát a következő utasítás a már ismert `_delay_ms(1000);` Következő lépésben megint változtatni kell az állapotokon. A PA0-s lábat logikai alacsony értékre kell állítani, míg a PA2-es lábat logikai magasra. Ehhez a következő két utasítás megfelelő: `cbi(PORTA,0); sbi(PORTA,2);` Ezzel végrehajtottunk egy invertálást a két lábon. Már csak egy utolsó, de nagyon fontos utasítás maradt hátra, mégpedig a várakozás ismét 1 másodpercig. Ezt már mindenki fejből tudja, hogy a `_delay_ms(1000);` utasítással tudja megtenni.

```
while (1)
{
	//Bekapcsolja a LED1-et, kikapcsolja a LED2-t
	sbi(PORTA,0);
	cbi(PORTA,2);
	//Várakozik
	_delay_ms(1000);
	//Kikapcsolja a LED1-et, bekapcsolja a LED2-t
	cbi(PORTA,0);
	sbi(PORTA,2);
	//Várakozik
	_delay_ms(1000);
}
```

A kódunkat elmentjük, majd lefordítjuk és letöltjük a mikrovezérlőre. Ha mindent jól csináltunk akkor hiba nélküli fordítást kapunk és a ledek az elvárt módon villognak.

Kitekintés

Egy kis programozási trükk az invertáláshoz:

Egy kis trükköt szeretnénk bemutatni, amivel a villogtatás, vagy egyszerűen csak egy adott láb állapotának kapcsolgatása egyszerűbben megvalósítható. Abban az esetben ha egy adott láb állapotát invertálni szeretnénk, használhatunk bitmódosító operátorokat.

Maradjunk a két ledes villogtató kódnál. A PA0 és PA2 lábak állapotát invertáljuk megadott időközönként. Ami szükséges, az egy kezdeti érték. Ezt a végtelen ciklus elé rakjuk be, hiszen ezt csak egyszer szeretnénk beállítani. Jelen esetben legyen a `PORTA = 0b00000001;` Ezzel beállítottunk a PA0-s lábat logikai magas értékre, és a PA2-es lábat logikai alacsonyra.

Most jöhet a trükk. Megvan a kiindulási állapot és a két láb állapotát szeretnénk invertálni. Erre használjuk a következő utasítást `PORTA ^= 0b00000101;` Ezzel egy sorban megvalósítottuk a két láb állapotváltását.

Mit is csináltunk pontosan? Azt tudjuk, hogy a PORTA regiszter két bitjét kell módosítanunk a többit pedig változatlanul hagyni. A fenti kódrészlet tartalmaz nyelvi rövidítéseket. Például a `^=` operátorok nélkül a következőt írhattuk volna:

```
PORTA = PORTA ^ 0b00000101;
```

A két kód teljesen ugyanazt fogja végrehajtani. A hosszabb verzió segítségével magyarázom el a működést. Bitmódosító operátorokról már volt szó előző részekben, így arról nem ejtenék szót. Itt a kizáró vagy operátort használjuk. Tudjuk róla, hogy abban az esetben lesz a végeredmény 1-es, ha az operátor jobb és bal oldalon lévő bitek különböznek egymástól. Ezt a tulajdonságát használjuk ki. Minden bitpozícióba, ahol nem szeretnénk módosítani az állapotot 0-t írunk, a módosítandó pozícióba pedig 1-t. Nézzük meg, miért is működik ez a megoldás? A következő táblázat kicsit talán segít a megértésben:

Eredeti állapot	Módosító bit	Új állapot
0	0	0
0	1	1
1	0	1
1	1	0

Ez a táblázat pontosan a kizáró vagy operátor igazságtáblája. Látjuk, hogy a 0-ás módosító bit megtartja az eredeti állapotot, míg az 1-es megváltoztatja. Ezzel megvalósítottuk az invertálást, tehát csak egy várakozás kell már a kódba, ami eredetileg is benne volt. A többi rész a végtelen ciklusból törölhető. Tehát a következő kód megvalósítja a felváltva való villogást:

```
PORTA = 0b00000001;
while (1)
{
      PORTA ^= 0b00000101;
      _delay_ms(1000);
}
```