

12. Debuggolás - hibakeresés a programban

Írta: Kiss Dávid

Lektorálta: Proksa Gábor

BEVEZETÉS

A processzorok bináris számokon különféle műveleteket tudnak végrehajtani, például összeadást, összehasonlítást, logikai műveleteket. Ahhoz, hogy számunkra hasznos feladatokat tudjanak végezni, meg kell tudnunk mondani a processzornak, hogy mely számokon milyen műveleteket hajtson végre. Ezeket a parancsokat nevezzük utasításnak, az összes utasítást pedig utasításkészletnek. Minden utasításhoz hozzárendelhetünk egy-egy bináris számot. Ezzel meg is oldottuk a problémát, hiszen ha egy utasítást szeretnénk adni a processzornak, akkor elegendő egy bináris számot küldenünk neki, ami alapján tudni fogja, hogy mit kell tennie.

A mikrovezérlők processzora is így működik: a programmemóriájából egymás után olvas ki számokat és hajt végre ez alapján utasításokat.

Hogyan kerül be a programmemóriába az utasítás? A legegyszerűbb az lenne, ha az utasításhoz tartozó bináris számokat "kézzel" beírogatnánk a memóriába. Néhány utasítás esetén minden további nélkül megtehetnénk, viszont bonyolult programoknál nagyon sokáig tartana a begépelés és rengeteg hibát követhetnénk el. Ezért kitalálták az úgynevezett assembly programnyelvet.

Ebben a nyelvben tulajdonképpen az utasításokra már nem bináris számokkal hivatkozunk, hanem szöveggel (például összeadásra az "ADD" kifejezéssel), majd az elkészült szöveges leírás után egy számítógépes program "lefordítja" a szöveget bináris számokká, vagyis utasításokká.

Ezzel a módszerrel az a gond, hogy különböző processzoroknak különböző utasításai lehetnek, így minden processzoron más-más assembly kódot kellene írunk. Erre a problémára adnak megoldást a magas szintű programnyelvek, amilyen az előző tananyagrészekben megismert C nyelv is. A C kódsorokat egy számítógépes program "lefordítja" assembly nyelvre, majd a keletkezett assembly kódot fordítja tovább gépi bináris kódra.

```

int main(void)
{
    //PORTok inicializálása
    IOInit();

    //A végtelen ciklus minden egyes lefutásakor a PORTA lábak értéke az ellentettjére (negáltjára) változik.
    while (1)
    {
        PORTA = ~PORTA;
    }

    //A végtelen ciklus miatt ez a kód már nem fut le, a fordító azonban hiányolná a return-t
    return 0;
}

```

1. ábra - C programkód

Az első ábrán látható a C forráskód, ez az amit könnyedén tudunk olvasni. Természetesen nem csak egy fájlból áll a programunk, akár több *.c és *.h fájl is leírhatja a program működését. Egy ilyen *.c fájl tartalma látható a fenti képen, és ti is ilyen fájlokkal fogtok a leggyakrabban találkozni a tananyag során.

(A * jelentése a számítástechnikában jellemzően az, hogy a * helyén bármilyen karakterből állhat bármennyi. Tehát a *.c azt jelenti, hogy az összes c fájl. Ki is próbálhatjuk számítógépünk keresőjében a *.c vagy a *.jpg kifejezéseket, utóbbi csak jpg kiterjesztésű képeket fog listázni.)

```

0000005F PUSH R28      Push register on stack
00000060 PUSH R29      Push register on stack
00000061 IN R28,0x3D    In from I/O location
00000062 IN R29,0x3E    In from I/O location
IOInit();
00000063 CALL 0x00000036 Call subroutine
PORTA = ~PORTA;
00000065 LDI R24,0x3B    Load immediate
00000066 LDI R25,0x00    Load immediate
00000067 LDI R18,0x3B    Load immediate
00000068 LDI R19,0x00    Load immediate
00000069 MOVW R30,R18     Copy register pair
0000006A LDD R18,Z+0     Load indirect with displacement
0000006B COM R18        One's complement
0000006C MOVW R30,R24    Copy register pair
0000006D STD Z+0,R18    Store indirect with displacement
}
0000006E RJMP PC-0x0009 Relative jump
--- No source file -----
0000006F CLI          Global Interrupt Disable
00000070 RJMP PC-0x0000 Relative jump

```

2. ábra - Assembly programkód

A fordítás első lépésében a preprocesszor (előfeldolgozó) és fordító (compiler) végzi a munkát. A fordító a magas szintű C nyelven íródott forrást egy alacsonyabb szintű, ún. Assembly nyelvű kóddá alakítja. Ezt a kódot láthajtuk a középső lépésben. Ez a fájl még hordoz magában földi halandók számára is információt. Az assembly kód megvalósítja a C forrás által előírt funkcionalitást, a processzor limitált utasítás készletével.

Innen már csak egy lépés választ el minket a gépi kódtól, mint ahogy arról korábban már szó volt, az assembly fájl egy az egyben megfeleltethető a gépi parancsoknak, csak a szavakat a megfelelő számsorokra kell cserélni.

```
:10000000C942A000C9434000C9434000C943400AA
:10001000C9434000C9434000C9434000C94340090
:10002000C9434000C9434000C9434000C94340080
:10003000C9434000C9434000C9434000C94340070
:10004000C9434000C9434000C9434000C94340060
:10005000C94340011241FBECFE5D4E0DEBFCDBF29
:10006000E945F000C946F000C94000CF93DF930C
:10007000CDB7DEB78BE390E0FC0110828AE390E01D
:1000800021E0FC01208388E390E0FC01108287E3FB
:1000900090E0FC01108285E390E0FC01108284E393
:1000A00090E0FC01108282E390E0FC01108281E389
:1000B00090E0FC011082000DF91CF910895CF9372
:1000C000DF93CDB7DEB70E9436008BE390E02BE3E1
:1000D00030E0F90120812095FC012083F6CFF894CF
:0200E000FFCF50
:00000001FF
```

3. ábra - Gépi kód

A legutolsó lépés előlünk már teljesen rejtve marad, a számítógép megoldja helyettünk ezt a problémát is. Már a korábbi egyszerű programunkban is több forrásfájlunk volt, ezek annyi ilyen kis számsorozatba fordulnak le, ahány *.c fájlunk volt. A processzorra azonban csak egyet tudunk feltölteni, ezért utolsó lépésként az úgynevezett linker ezeket a fájlokat összefűzi egy egészé.

Természetesen e folyamat korántsem egyértelmű, ugyanazt a C kódot többféleképpen is meg lehet valósítani assemblyben. A compiler *optimalizálja* a kódot, néhány dolgot más sorrendben hajt végre, ha nem használunk egy változót, nem hozza létre feleslegesen, stb. Az optimalizációt ki tudjuk kapcsolni, és akkor pontosan az történik, mint amit leírtunk. A tananyagrészt során ki is fogjuk kapcsolni, hogy ne okozzon félreértéseket a működése.

A mai napig vannak olyan programok, melyeket assembly nyelven készítenek el. Ennek célja a futásidő és a programméret közben tartása. Minden utasításról pontosan lehet tudni, hogy mekkora helyet foglal, és hány órajel periódus alatt hajtódik végre. De sajnos azt nem lehet előre megmondani, hogy a fordító milyen utasításokkal valósítja meg a C nyelven íródott programunkat.

A másik ok az, hogy teljes kontrollt szeretnének a processzor felett. Az assemblyben leírt kódsorok egyértelműen gépi kóddá fordulnak le, a processzor pontosan azt az utasítást hajtja végre amit leírtunk neki. A C fordító is egy program, emberek írták, így előfordulhatnak benne hibák, bizonyos esetekben rosszul fordíthatja le a C nyelven írt programunkat.

Kitekintés

C programban lehetőség van arra, hogy egyes kódsorokat assemblyben írjunk, ezeket a fordító egyszerűen bemásolja az általa létrehozott assembly programba. Így kritikus helyzetekben teljes mértékben a kezünkbe vehetjük az irányítást.

MI IS AZ A “DEBUGGOLÁS”?

Most, hogy tudjuk hogyan fordult le a kódunk, hibák után fogunk benne kutatni. A fejlesztés során az első feltétel, hogy szintaktikailag helyes kódot írjunk. Ennek mikéntjéről már korábbi részekben értekeztünk. A szintaktikai helyesség azonban önmagában még korántsem jelenti azt, hogy a kód úgy is viselkedik, ahogyan mi azt elterveztük.

A processzor másodpercenként több milliószor elvégzi amit előírtunk neki, tehát szabad szemmel nem sok esélyünk van követni a működését. Másrészt nem is nagyon látjuk a dolgok folyását, hiszen igen nehéz szemügyre venni az elektronok táncát a szilícium-chipben (és nem csak azért, mert egy fekete műanyag tokban van). Ezekre a problémákra ad megoldást a debuggolás, amely alatt a hibakeresés folyamatát értjük. Lehetőségünk van lépésről lépésre futtatni a kódunkat, követni a különböző regiszterek és változók állapotát, és így ellenőrizni azt, hogy valóban helyes-e a működés.

Kitekintés

Amennyiben járatos vagy az angol nyelvben, furcsállhatod a kifejezést: debuggolás (debugging ~”kibogarazás”). A szó helyes fordítása hibakeresés, de joggal merül fel a kérdés, hogy honnan is ered ez a kifejezés. A régi időkben, amikor még a számítástechnika gyerekcipőben járt, és az a számítási kapacitás, amely most az előtted lévő kis fekete tokban található, egy tornaterem méretű helyiséget ölelt fel, a hibakeresés egy kicsit más jellegű folyamat volt. Ebben az óriási térben helyezték el a különböző alkatrészeket, és kötötték össze őket nagyon-nagyon sok vezetékkel. Ezeket a gépeket még nem is programozták, a kívánt funkciót a megfelelő bekötéssel érték el, több tízezer vezeték segítségével. Természetesen ez a komplex konstrukció sem elsőre működött, ez is tartalmazott hibákat, amiket fel kellett derítenie a mérnököknek. A hosszas kutatómunka és méricskélés eredményeképp találtak egy bogarat, mely több vezetékot összekötött, tehát kiderült, hogy nem tervezési hiba történt.

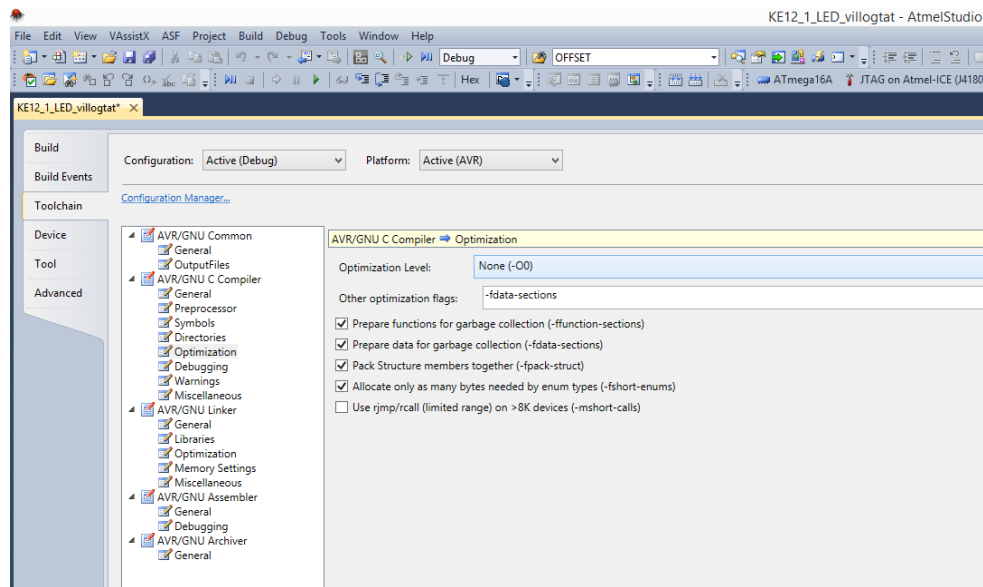
EGY EGYSZERŰ PÉLDA

Az első példa, ami reméljük rávilágít a debuggolás, mint funkció létfontosságára, egy nagyon egyszerű, egyetlen soros kis program. A látványosabb eredmény érdekében kössünk a mikrovezérlő PA0 lábára egy LED-et, ahogy azt tettük a 9-es és 11-es részekben is. Töltsük a mikrovezérlőre a `KE12_1_LED_villogtat` projektet. Ha ezt sikerrel elvégeztük, akkor azt látjuk, hogy a PA0 lábra kötött LED világít. Nézzük csak meg a feltöltött kódot!

A `KE12_1_LED_villogtat` szoftverben leírtak tanulsága alapján a LED-nek villognia kéne, és ahogy azt az előző tananyagrészen érintőlegesen tárgyaltuk, villog is. A processzor nagyon gyorsan futtatja a kódot, a LED villog ugyan, de olyan gyorsan, hogy az emberi szem képtelen érzékelni, ezért úgy látjuk, mintha folyamatosan világítana.

Először demonstrációs céllal nézzük meg, hogy mi az alapja annak, amiről a tananyagrészt hátralévő része szólni fog. Ehhez most csak kövesd pontosan a leírtakat, aztán a későbbiekben megnézzük részletesebben az egyes funkciókat. Még mielőtt lefordítanánk a projektet, néhány beállítást tisztázzunk. A bevezetés során szó volt róla, hogy az optimalizáció nem lesz a barátunk, ezért győződjünk meg róla, hogy valóban

ki van-e kapcsolva. Ezt a projekt beállítások között találjuk meg, a *Toolchain* fülön, az *AVR/GNU C Compiler/Optimization* szekció alatt. Itt válasszuk a *None (-O0)* beállítást.



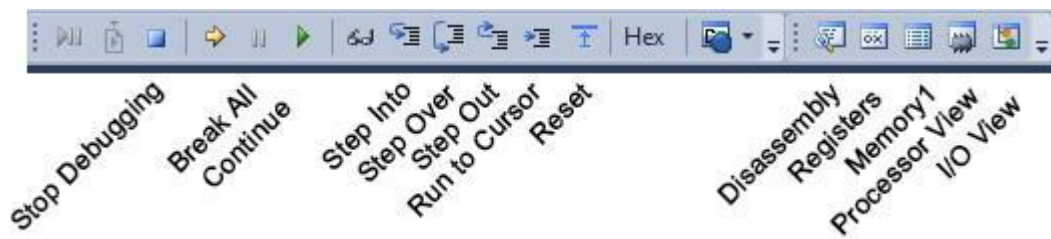
4. ábra - Optimalizáció kikapcsolása

Futtassuk most debug módban a kódot! Ehhez kattintsunk a *“Start Debugging”* gombra. Láthatjuk, hogy az Atmel Studio felülete kissé meg is változott. Ebben a módban a mikrovezérlőn ugyan úgy fut a kód mint eddig, csak a számítógép folyamatosan kommunikál vele, így befolyásolni tudjuk a kód futását, például szüneteltetni, vagy soronként léptetni.

Az utóbbit próbáljuk is ki! Első lépésben a *“Break all”* (Ctrl + F5) gomb segítségével állítsuk meg a program futását, majd a *“Step Over”* (F10) gomb segítségével soronként (függvényhívásonként) tudunk lépkedni a kódban. A zárójelekben a funkciókhoz rendelt billentyű kombinációt találod, javasoljuk most ezeknek a használatát, így nem kell az eszköztáron keresgélni.

Soronként lépkedve láthatjuk, hogy a LED valóban változtatja az állapotát. A program tehát helyesen működik, csak a szemünk nem elég gyors ahhoz, hogy a villogást észlelje. Az előző részben megírt LED villogtató program is hasonlóan működött, csak tartalmazott késleltetést. Erre éppen azért van szükség, hogy a szemünk is érzékelni tudja a villogást. Érdekes lehet kipróbálni, hogy minimum hány ms késleltetés kell ahhoz, hogy szabad szemmel meg lehessen különböztetni az egyes villanásokat.

A DEBUG FELÜLET ÁTTEKINTÉSE ÉS FUNKCIÓI



5. ábra - Debuggolás során használatos menüsor

Most, hogy az első lépéseken túl vagyunk, ismerjük a meg többi gombot ezen az eszköztáron, melyeket használni fogunk a továbbiakban. Az első három gomb a kód futását befolyásolja.

Stop Debugging

A debuggolás leállítására használatos gomb. A debug folyamatot valamint a mikrovezérlő és a számítógépünk közötti kapcsolatot állítja le. Fontos, hogy miután végeztünk a debuggolással, és módosítani szeretnénk a kódon, mindig állítsuk le a debuggolást. Ha ezt nem tesszük meg, akkor nem fogjuk tudni a módosított programot feltölteni a mikrokontrollerre, mert foglalt lesz a programozó.

Break All

Amikor erre a gombra kattintunk, pillanatnyilag felfüggesztjük a program futását. Ezt úgy kell elképzelni, mintha meg tudnánk állítani a mikrovezérlőben az órajelet. A valóságban persze nem ez történik, hiszen akkor kommunikálni sem tudnánk vele.

Continue

Ha megállítjuk a program futást, valahogyan folytatni is kell később, erre szolgál ez a gomb.

A következő gombok szintén a kód futásával kapcsolatosak, ezek segítségével tudjuk „léptetni” a kódot, azaz olyan tempóban futtatni, ahogyan szeretnénk.

Step Into

A következő kódsort hajtja végre, méghozzá úgy, hogy ha az egy függvényhívás, vagy kapcsos zárójelekkel határolt blokk, akkor abba belép, és annak az első sorát hajtja végre.

Step Over

Szintén a következő sort hajtja végre. Amennyiben az egy függvényhívás, vagy kapcsos zárójelekkel határolt blokk, akkor nem lép be, viszont a blokk tartalma végrehajthatódik.

Tegyük fel, hogy a LED-et nem egyszerű bitmanipulációval kapcsoljuk be és ki, hanem van egy SetLED() függvényünk. Ha a “Step Over” gombbal lépünk erre a függvényre, akkor csak azt látjuk, hogy bekapcsolt a LED, azonban ha a “Step Into”-val lépünk tovább, a függvény belsejébe jutunk, ahol is a LED bekapcsolásának lépéseit követhetjük.

Kitekintés

Kódolás technikailag elegánsabb a SetLED() függvény használata, hiszen így elrejtjük az olvasó elől az első ránézésre semmitmondó bitműveleteket, helyette pedig egy olyan függvényt hívunk, aminek a neve kommentálás nélkül elárulja a funkcióját, és a háttérben megvalósítja a szükséges műveleteket.

Step Out

Abból a függvényből tudunk kilépni, amelyben éppen benne vagyunk. A program futása a következő függvény hívásnál áll meg. Természetesen a függvény hátralévő része is lefut, csak nem áll meg a futás a következő függvény elejéig.

Run to cursor

A kurzor pozíciójáig futtatja a kódot.

Reset

A processzor újraindítására szolgál. Gyakorlatilag azzal ekvivalens, hogy kihúzzuk és újra bedugjuk a processzor tápját, attól a kellemetlenségtől eltekintve, hogy így nem kell újra felépíteni a debug kapcsolatot.

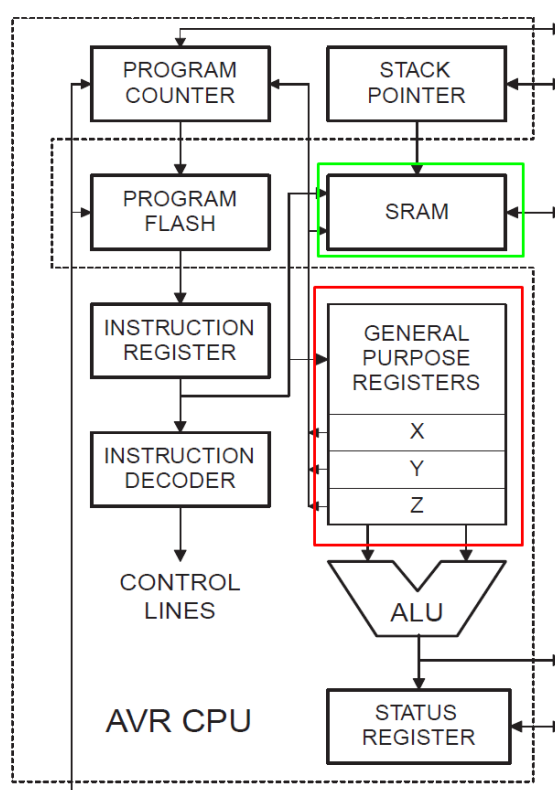
A hátralévő gombokkal különböző nézeteket nyithatunk meg, melyekből informálódhatunk a processzor belső állapotáról. Ezeknek a nézeteknek közös tulajdonsága, hogy nem csak olvasni tudjuk a különböző regisztereket, hanem mi magunk is be tudunk avatkozni „kézzel” az állapotukba, azaz át tudjuk írni az értéküket.

Disassembly

Ha ide kattintunk, akkor megjelenik az Assembly kód, amit a fordító készített a mi C fájlunkból. Akkor lehet erre a nézetre szükségünk, ha pontosan látni akarjuk azt, hogy mi történik a processzorban.

Registers

Ennek a nézetnek a segítségével a CPU regisztereinek a tartalmát figyelhetjük. 32 ilyen 8 bites regiszter van a mikrokontrollerben, ezek ideiglenes tárolók azoknak az adatoknak, amikkel dolgozunk. Például ha össze akarunk adni két számot, akkor ez a két szám először bekerül két ilyen regiszterbe, és csak utána tudja őket összeadni a processzor. Ha egy ciklust hajtunk végre, akkor a ciklusszámláló is egy ilyen regiszterben tárolódik.



6. ábra - ATmega16A mikrovezérlő regiszterei

Az ábrán GENERAL PURPOSE REGISTERS néven láthatóak. Nem összekeverendő a RAM fogalmával, amely egy nagyobb tömbként kezelendő memória terület. Ebből másolódnak az éppen szükséges változók a regiszterekbe.

Memory1

A processzor memóriatérképét tekinthetjük meg itt. Az összes elérhető memóriaterület láthatjuk, nyers formában, hexadecimálisan kódolva, a sorok végén pedig ASCII karakterekként.

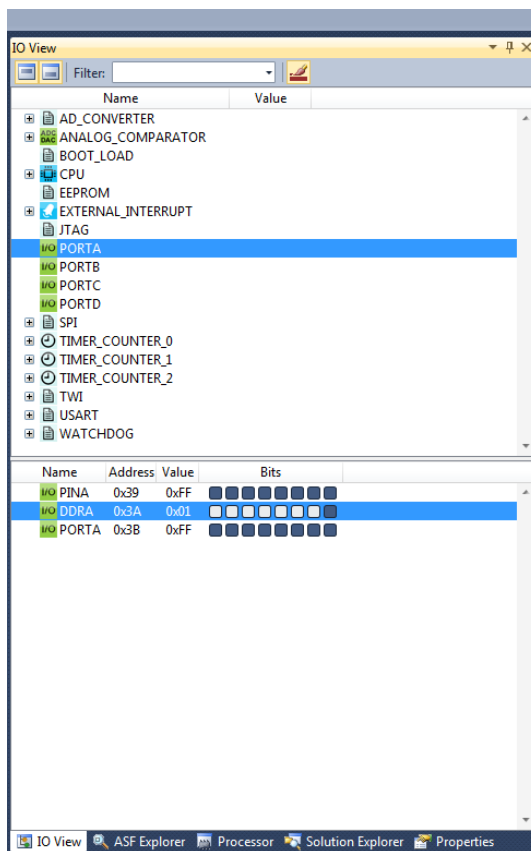
Processzor View

Ebben a nézetben a processzor végrehajtó egységével szorosan összefüggő regiszterek tartalmához férünk hozzá. Ezek a következők:

- Program Counter – programszámláló
- Stack pointer
- X regiszter
- Y regiszter
- Z regiszter
- Status regiszter
- és végül a 32 általános célú regiszter.

I/O View

Ez a nézet lesz számunkra a leginkább hasznos. Itt az összes periféria állapotát megvizsgálhatjuk és módosíthatjuk. Könnyűszerrel megállapíthatjuk például, ha valamit nem megfelelő módon konfiguráltunk.



7. ábra - I/O View panel

Ha kiválasztjuk a PORTA-t, akkor alul megjelennek az A porthoz tartozó regiszterek, ezek a PINA, DDRA és a PORTA.

A PINA regiszter tárolja azt, hogy a kontroller milyen logikai értéket olvasott be az adott lábakról. A DDRA azt, hogy kimenetnek, vagy bemenetnek van konfigurálva az adott láb. A PORTA regiszterbe pedig mi írhatjuk bele, hogy milyen értéket szeretnénk látni az adott lábon, ha azt kimenetként konfiguráltuk.

Ha a PORTA legelső bitjére (jobbról az első négyzet) kattintunk, akkor láthatjuk is, ahogy a LED-et tudjuk kapcsolgatni a próbapanelen. Legyünk türelmesek, amíg a beállításunk érvényre jut, meg kell várni amíg a programozó eszköz kiírja a regisztertartalmat, majd visszaolvassa az újakat. Érdekes játék lehet kipróbálni azt is, hogy ezeket a lábakat lefogjuk az ujjunkkal és léptetünk egyet a programon. Ha szerencsénk van, akkor tapasztalhatunk némi változást a bemenetek értékén. Ennek oka a processzor elektronikai felépítésében keresendő, illetve abban, hogy ezeket a lábak nem kötöttük sehova, tehát "lebegnek".

AMIKOR BONYOLÓDIK A HELYZET

Egészítsük ki most néhány sorral a kis programunkat, hogy még egy debug funkciót megnézhessünk közelebbről. Ehhez nyissuk meg a *KE12_2_LED_szamlalo* projektet. A módosításunk célja, hogy ne minden ciklusban változzon meg a LED állapota. Ehhez szükségünk lesz egy változóra. Ezt a változót nevezzük mondjuk *counter*-nek, ami angolul számlálót jelent. E változó értékét 0 és 9 között fogjuk változtatni.

Lehet, hogy már sejted milyen trükkre készülünk. Az új programunk a fenti counter változót minden ciklusban megnöveli, és amikor kisebb az értéke mint egy előre definiált szám, a példa esetében 4, akkor bekapcsolja a LED-et, egyébként kikapcsolja.

Kitekintés

Gyakorlatilag egy szoftveres PWM-et fogunk megvalósítani. De mi is az a PWM? A három betűs mágikus rövidítést a Pulse Width Modulation szavak oldják fel helyesen. Magyarul Impulzus Szélesség Modulációt jelent, használhatnánk tehát az ISzM rövidítést is, de inkább ne tegyük, nem ez az elnevezés terjedt el.

A PWM-mel a tananyag folyamán még többször fogsz találkozni, fogunk építeni analóg PWM fokozatot, illetve azt is meg fogjuk tanulni, hogy a mikrovezérlőbe beépített digitális perifériák segítségével hogyan tudunk PWM jelet előállítani.

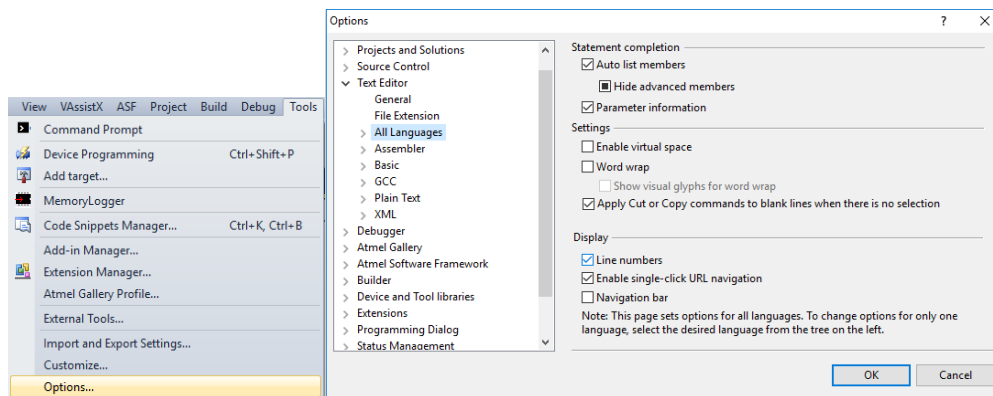
Impulzus szélesség modulációval befolyásolni tudjuk például LED-ek fényerejét, motorok nyomatékát (és ezáltal sebességét). Azt használjuk ki, hogy a valós fizikai rendszerek, eszközök (mint például a szemünk vagy egy villanymotor) nem tudják a fizikai mennyiségek változását olyan gyorsan követni, mint amilyen gyorsan azokat egy mikrokontrollerrel módosítani tudjuk.

Pontosan ez történt az első példa esetében is, a szemed kisimította a változást. Ezért látsz például a monitorodon mozgó képet, pedig valójában állóképek sorozatát jeleníti csak meg.

A PWM-et széles körben alkalmazzák. A legközelebbi gyakorlati példa ott lapul a zsebedben, a mobiltelefon kijelzőjének háttérvilágításának fényerejét is így lehet beállítani.

A programban szándékosan szerepel egy hiba, ezt fogjuk most a debuggolás eszközének segítségével felderíteni. Első lépésben nézzük meg mit látunk a próbapanelen. A LED nem világít, pedig mi olyan programot írtunk, ami legjobb tudomásunk szerint villogtatja.

Kapcsoljuk be a sorok számozását a *Tools/Options* ablak *Text Editor/All Languages* füle alatt a *Line Numbers* jelölőnégyzet bekattintásával.



8. ábra - Sorok számozásának bekapcsolása

1. Indítsuk el a programot Debug módban, *“Start Debugging”* (F5) gombbal. Ekkor elindul a program futása. A LED most sem világít.
2. A következő lépésben reseteljük a processzort, *“Reset”* (Shift+F5) gombbal. Erre azért van szükség, mert bármikor is állítjuk meg a processzort, már nagyon sok programsoron túl lesz, mivel sokkal gyorsabb mint mi. Ezt a lépést akár el is tudjuk kerülni, ha a *“Start debugging and break”* (Alt+F5) gomb segítségével indítjuk a debuggolást. Lépünk tovább a *“Step Over”* (F10) gomb segítségével. Látható, hogy most ugrottunk a `counter`-t létrehozó 14-es sorra.

```

10 int main(void)
11 {
12
13     //A folyamatosan futó számláló
14     int counter = 0;
15     //PORTok inicializálása
16     IOInit();

```

9. ábra - Counter változó létrehozása

3. A következő lépés a `IOInit()` függvényhívást tartalmazó 16. sor.
4. Ha még egyet lépünk, akkor a 22. sorban találjuk magunkat. Ez a sor vizsgálja meg, hogy a `counter` változó értéke elérte-e már a 10-et, ha nem akkor majd megnöveli azt. Azért kötjük ki, hogy 10-nél kisebb legyen az értéke, hogy ne kelljen a végtelenségig növelnünk. Lépünk tovább!

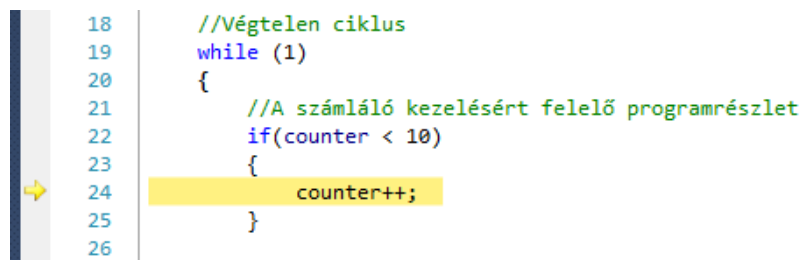
```

18 //Végtelen ciklus
19 while (1)
20 {
21     //A számláló kezelésért felelő programrészlet
22     if(counter < 10)
23     {
24         counter++;
25     }
26

```

10. ábra - Counter változó feltételvizsgálata

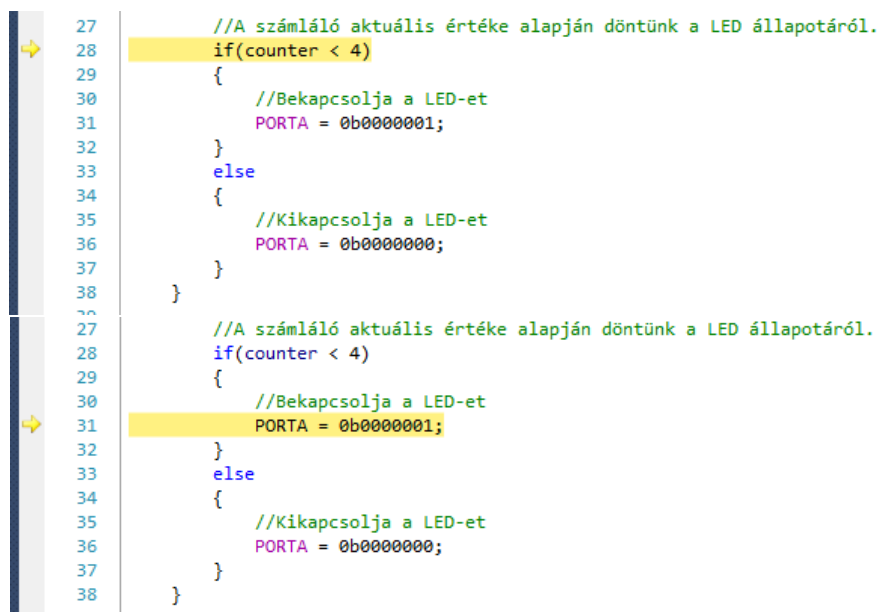
5. Mivel a `counter` értéke kisebb volt mint 10, ezért megnöveljük, ez látható a 24. sorban. Lépünk tovább!



```
18 //Végtelen ciklus
19 while (1)
20 {
21     //A számláló kezelésért felelő programrészlet
22     if(counter < 10)
23     {
24         counter++;
25     }
26 }
```

11. ábra - Counter változó növelése

6. Most megvizsgáljuk, hogy a `counter` változó kisebb-e mint 4. Ha kisebb, akkor bekapcsoljuk a LED-et, ha nagyobb, akkor pedig ki. Lépünk tovább!

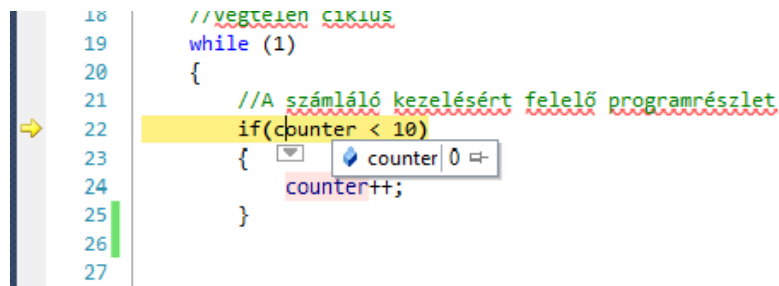


```
27 //A számláló aktuális értéke alapján döntünk a LED állapotáról.
28 if(counter < 4)
29 {
30     //Bekapcsolja a LED-et
31     PORTA = 0b0000001;
32 }
33 else
34 {
35     //Kikapcsolja a LED-et
36     PORTA = 0b0000000;
37 }
38 }
39 }
40 }
41 }
42 }
43 }
44 }
45 }
46 }
47 }
48 }
49 }
50 }
51 }
52 }
53 }
54 }
55 }
56 }
57 }
58 }
59 }
60 }
61 }
62 }
63 }
64 }
65 }
66 }
67 }
68 }
69 }
70 }
71 }
72 }
73 }
74 }
75 }
76 }
77 }
78 }
79 }
80 }
81 }
82 }
83 }
84 }
85 }
86 }
87 }
88 }
89 }
90 }
91 }
92 }
93 }
94 }
95 }
96 }
97 }
98 }
99 }
100 }
```

12. ábra - Counter változó feltételvizsgálata, majd LED bekapcsolása

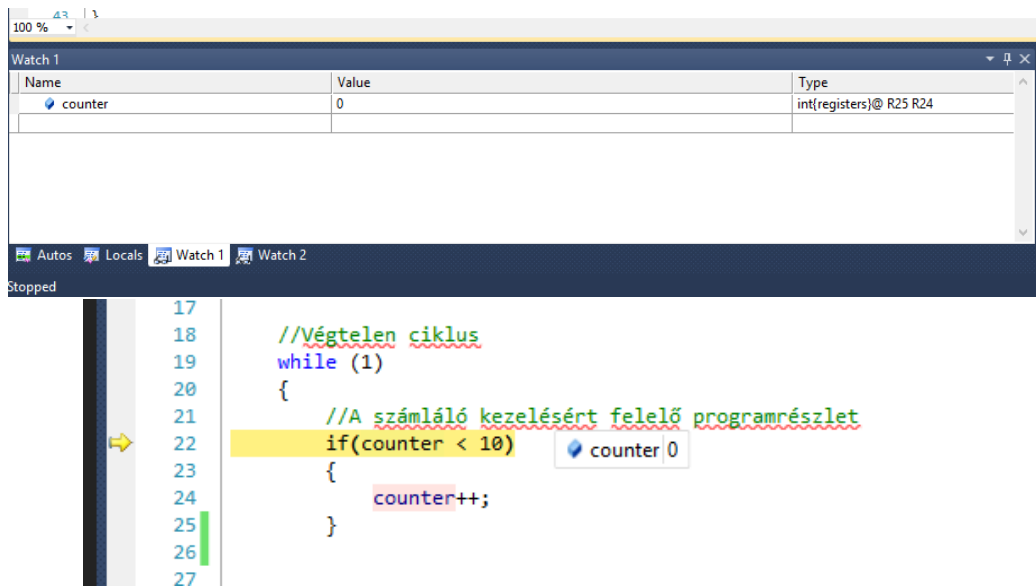
7. A szoftver megáll még a 38. soron, de ez csak a végtelen ciklus végét jelzi, amiben fut a programunk, lépünk tovább ezen is!

8. A LED világít, mi pedig a 22. sorban állunk, a `counter` kiértékelésénél. Ahhoz, hogy meg tudjuk nézni, hogy hol történik más, mint amit elterveztünk, jó lenne látni a változók aktuális értékét. Ehhez vigyük az egeret a `counter` egy példánya fölé.



13. ábra - változó értékének megjelenítése

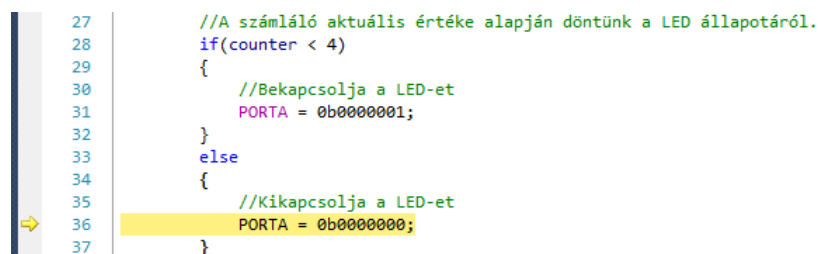
Ilyenkor láthatjuk a változó aktuális értékét. Ha rákattintunk a kis rajzszög szimbólumra, akkor ott is marad a kis ablak.



14. ábra - Watch Window, változók folytonos nyomonkövetése

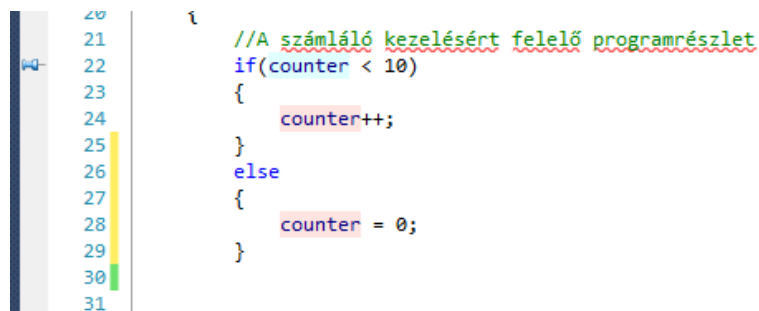
További lehetőségként a *Watch Window*-ban is megjeleníthetjük, ha a *Name* oszlop egy szabad mezőjébe begépeljük a változó nevét.

9. Léptessük most a kódot az F10 gomb nyomogatásával (kényelmesebb, mint a “*Step over*” gombra kattintani). Azt figyelhetjük meg, hogy a *counter* értéke folyamatosan nő, és amikor eléri a négyet, a LED kikapcsol, hiszen a 28. sorban található kifejezés hamissá válik, így a 36. sorra fogunk ugrani.



15. ábra - LED kikapcsolása

10. Amikor a számláló értéke eléri a 10-et, érdekes jelenséget figyelhetünk meg. A `counter` értéke nem változik többé. Ez azért van így, mert csak akkor növeljük az értékét, ha 10-nél kisebb a számláló. Igen ám, de így soha többet nem lesz 4-nél kisebb. Ezzel meg is találtuk a hibánkat!
11. Javítsuk ki a kódot!
12. Ne nézd meg a megoldást, írd meg a javítást!
13. Kész?
14. Kipróbáltad?
15. Működik?
16. Ha igen, akkor jó, ha nem sikerült rájönni, íme a jó megoldás:



```
20 {  
21 //A számláló kezelésért felelő programrészlet  
22 if(counter < 10)  
23 {  
24     counter++;  
25 }  
26 else  
27 {  
28     counter = 0;  
29 }  
30 }  
31 }
```

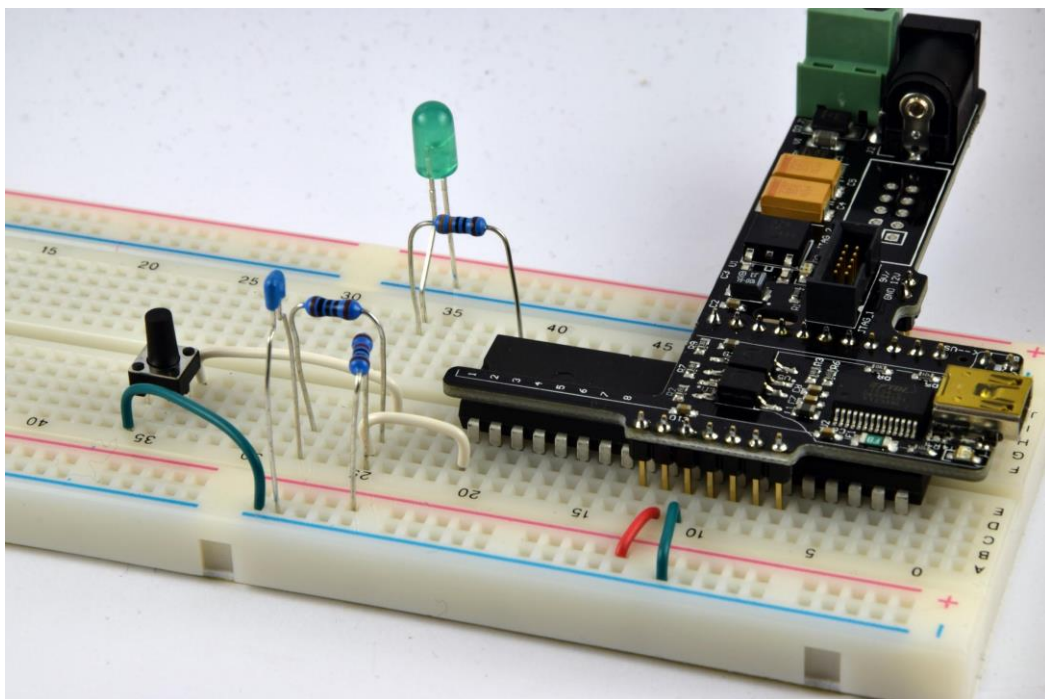
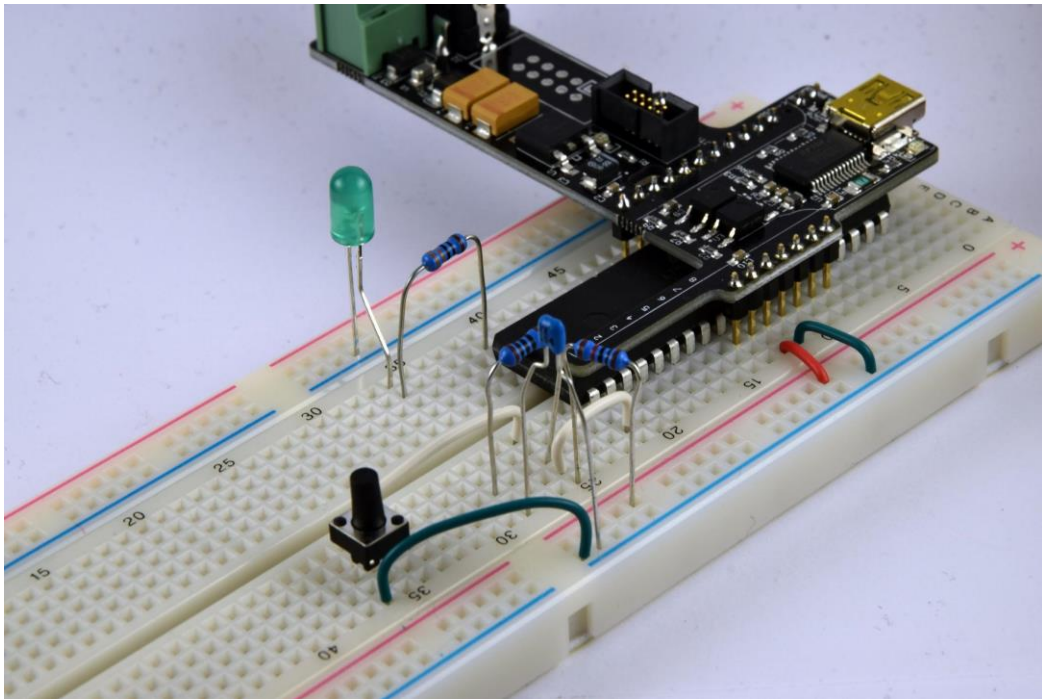
16. ábra - Szükséges javítás

Bizony, a számlálót le kell nullázni, ha már elérte a maximális kívánt értékét.

17. Próbáljuk ki a javított verziót (ha eddig nem ment), majd kísérletezzünk a 4 helyett más 0 és 9 közötti értékekkel. Láthatjuk, hogy minél nagyobb ez az érték, annál fényesebben világít a LED.

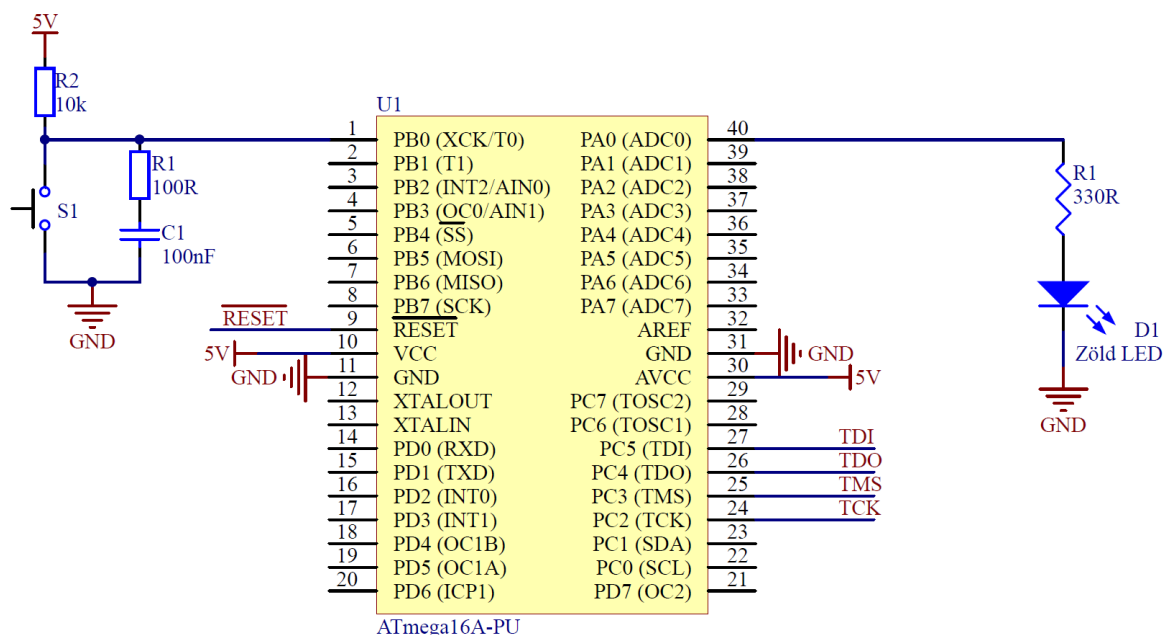
MÉG EGY KIS BONYOLÍTÁS

Még egy csavart vigyünk bele a programunkba. A PB0 lábra kötött nyomógomb segítségével állítsuk a LED fényerejét. Ehhez inicializálni kell a B port 0-ik lábát bemenetként, illetve egy nyomógombot is rá kell kötnünk erre a lábra. Az áramkörünk a breadboard-on így néz ki:



17. ábra - Nyomógombbal ellátott kapcsolás megvalósítása breadboard-on

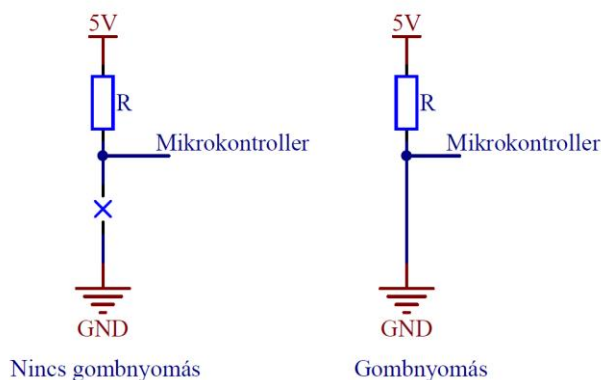
Nézzük most meg, hogyan is működik ez a kapcsolás, hisz nyomógombra akár később is szükségünk lehet. Kapcsolási rajz formában az alábbi ábrán látható.



18. ábra - Nyomógommbal kiegészített LED-es kapcsolás

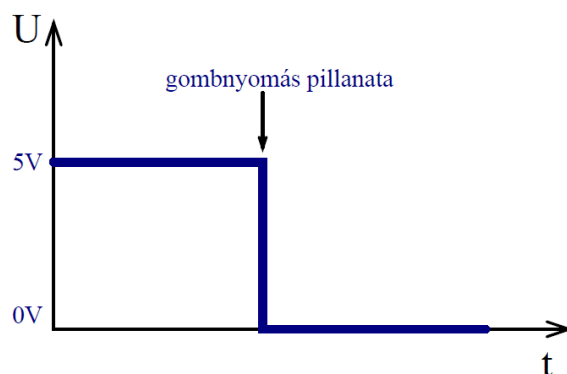
A nyomógombhoz tartozó kapcsolás (baloldalt) áll egy felhúzó ellenállásból, egy kapcsolóból, illetve láthatunk egy sorba kapcsolt ellenállást és kondenzátort is, ami egy alul áteresztő RC szűrőt valósít meg (hogy ez mi és mire is szolgál azt kicsit lejjebb fogjuk látni).

A legfontosabb ezek közül a felhúzó ellenállás és a kapcsoló, nézzük meg külön most ezt a részt:



19. ábra - Nyomógomb állapotai

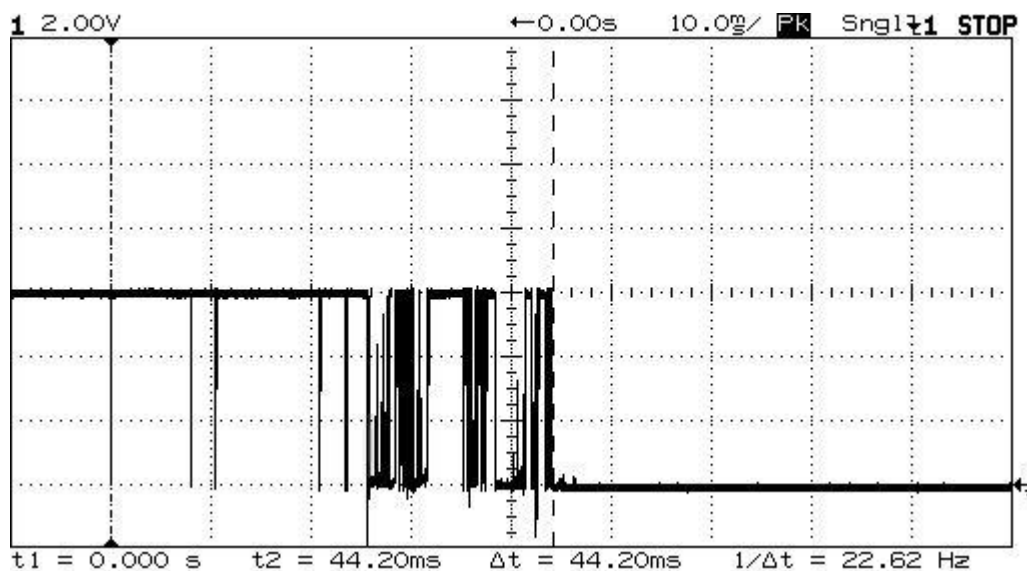
A baloldali kapcsolás mutatja azt az állapotot, amikor nem nyomjuk a gombot, míg a jobboldali kapcsoláson a lenyomott állapot figyelhető meg. A következő ábra a mikrovezérlő lábán mutatja a feszültség alakulását.



20. ábra - Gombnyomás pillanata a mikrovezérlő lábán

Amikor nem nyomjuk a gombot, gyakorlatilag nem folyik áram az ellenálláson, így feszültség sem esik rajta, a tápfeszültség megjelenik a mikrovezérlő lábán, amikor lenyomjuk a gombot, akkor már folyik áram, ezért a feszültség leesik 0 V-hoz közeli értékre. (Elgondolkoztató feladat: hogy kell úgy bekötni a gombot, hogy akkor jelenjen meg a feszültség, amikor megnyomjuk a gombot?)

A maradék két alkatrész alkotja az alul áteresztő szűrőt. Ez azt jelenti, hogy ez az ellenállás és kondenzátor minden gyorsan változó jel szempontjából rövidzár, de egyenfeszültségű jelek szempontjából szakadás, vagyis mint egy szűrő viselkedik, a lassan változó jeleket átengedi, de a gyorsan változó jeleket már nem. Erre azért van szükség, mert a nyomógomb belső mechanikája a gombnyomások során hajlamos rezegni. Ez számunkra elég kedvezőtlen viselkedés, mert a kontaktusok egymás után többször elengednek és összezárnak, ez pedig a mikrovezérlő szempontjából több gombnyomást jelentene, holott mi csak egyszer szeretnénk volna megnyomni a gombot. A jelenséget az alábbi ábrán láthatjuk.

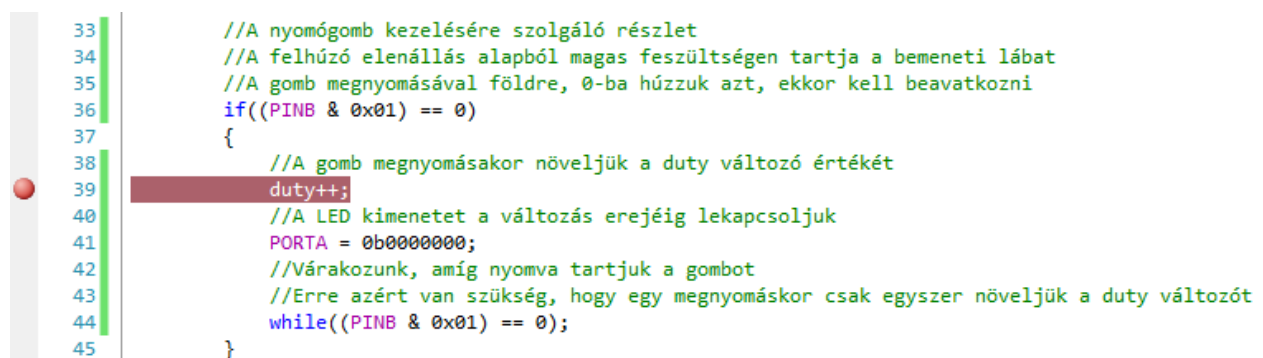


21. ábra - Rezgő nyomógomb

Ezeket, az ún. nagyfrekvenciás komponenseket tudjuk kiszűrni az alul áteresztő RC szűrőnkkel, mivel a szűrő nem fogja átengedi őket.

Térjünk vissza a programunkra, nyissuk meg a *KE12_3_LED_nyomogomb* projektet. A fentiekben azt ígértem, hogy ennek a szoftvernek a használatával, a LED fényerejét a nyomógommbal tudjuk majd állítani. A gyakorlat azonban most mégsem ezt mutatja.

Mit rontottunk el? Miért nem történik semmi a gombnyomásra? A debug környezet egyik leghasznosabb funkciójáról nem beszéltünk még. Ki tudunk jelölni bizonyos sorokat, melyeket ha elér a program, akkor megáll, és meg tudjuk nézni a változóink és regisztereink aktuális értékét. Az ilyen jelöléseket hívjuk breakpoint-oknak. Elhelyezni úgy tudjuk őket, hogy a program sor előtti, bal szélén található szürkés sávra kattintunk. Ha sikerült, megjelenik egy piros kör, jelezve, hogy az adott sort elérve le fog állni a program futása. De hol állítsuk meg a programot? Nézzük meg észreveszi-e a gombnyomást, tegyünk egy breakpoint-ot az ezt vizsgáló részbe, azaz a 36. sorba.



```
33 //A nyomógomb kezelésére szolgáló részlet
34 //A felhúzó elenállás alaphő magas feszültségen tartja a bemeneti lábat
35 //A gomb megnyomásával földre, 0-ba húzzuk azt, ekkor kell beavatkozni
36 if((PINB & 0x01) == 0)
37 {
38     //A gomb megnyomásakor növeljük a duty változó értékét
39     duty++;
40     //A LED kimenetet a változás erejéig lekapcsoljuk
41     PORTA = 0b0000000;
42     //Várakozunk, amíg nyomva tartjuk a gombot
43     //Erre azért van szükség, hogy egy megnyomáskor csak egyszer növeljük a duty változót
44     while((PINB & 0x01) == 0);
45 }
```

22. ábra - Breakpoint elhelyezése

Amikor megnyomjuk a nyomógombot, akkor itt meg kellene állnia a programnak, azonban sajnos nem ez történik. Ilyenkor az első amire gyanakszunk, az a hardveres hiba. Jelen esetben, ha jól építetted meg a fentebb bemutatott kapcsolást, akkor ilyen hibáról nem beszélhetünk, mégis érdemes ellenőrizni, hogy minden megfelelően érintkezik-e a breadboard-on. Ezt legegyszerűbben úgy tudjuk megtenni, hogy multiméterrel megmérjük a mikrovezérlő azon lábának feszültségét, amelyre a nyomógombot kötöttük. Amikor nem nyomjuk a gombot, akkor 5V-hoz közeli értéket kellene látni a multiméteren, a gomb lenyomott állapota mellett pedig közel 0 V-t. Mivel a mérési eredmények nem a vártak megfelelően alakulnak, következtethetünk arra, hogy a láb máshogy viselkedik, mint ahogy azt szeretnénk, például kimenetként hajtja a mért pontot.

De ott hol lehet a gond? Gondoljuk végig, mi történik a portlábbal a szoftverben. Első lépésben inicializáljuk, utána pedig minden programciklusban megnézzük az értékét. Jelen esetben nem olyan bonyolult rájönni, hogy az elmélet melyik része nem működik megfelelően a gyakorlatban. Nézzük meg az inicializáló kódrészletet még egyszer. A `DDRB = 0xFF` soron megakadhat a szemünk. Ez a `DDRB` regiszter minden bitjét 1-be állítja, ami azt jelenti, hogy minden láb

kimenetként van konfigurálva. Itt lesz a probléma, írjuk át `DDRB = 0x00;`-ra. Ezzel a B port összes lábát bemenetként konfiguráltuk.

Indítsuk el ismét a programot, úgy, hogy a breakpoint-ot a helyén hagyjuk. Láthatjuk, hogy ha most megnyomjuk a gombot, akkor meg is áll a megfelelő soron a program. A `duty` változó aktuális értékét megtekinthetjük, ha egy tetszőleges kódbéli előfordulása fölé húzzuk az egeret, vagy "Watch window"-ban beállítjuk a figyelését. Ily módon meggyőződhetünk a program helyes működéséről: amikor a `duty` változó értéke kisebb, mint a `counter` értéke, a LED világít. További megfigyelésként megjegyezhetjük, hogy minden gombnyomásra valóban eggyel nagyobb lesz a `duty` változó értéke. Az érdekes, elsőre talán hibásnak tűnő rész 9 után következik, ugyanis a változó értéke 10 lesz. Ez nem baj, a következő kódrészlet ilyenkor érvényre jut, és lenullázza a változó értékét, ezt a program léptetésével ellenőrizhetjük.

```
40 //A nyomógomb számlálójának nullázásért felelős programrészlet
41 if(duty > 10)
42 {
43     duty = 0;
44 }
```

23. ábra - Nyomógomb számlálójának nullázása

ÖSSZEFOGLALÁS

A fenti néhány egyszerű példa remélhetőleg sikeresen rávilágított a szoftveres hibakeresési folyamatokra, illetve annak eszközeire. A későbbiek során ennél jóval komplexebb szoftvereket is meg fogunk valósítani, ezek az eszközök pedig hasznos társakká válnak a fejlesztés, programozás, hibakeresés során.

Megjegyzések

A debuggolás könnyen működésképtelenné válhat a fordítás során keletkező különböző fájlok inkonzisztenciája miatt. Ne dolgozzunk olyan mappában amit szinkronizál a számítógépünk valamilyen felhő alapú szolgáltatáshoz (Google Drive, Dropbox, stb.). Ha esetleg ezek után is kétes hibaüzeneteket kapunk, akkor a Build/Clean Solution paranccsal távolítsuk el a fordítás során keletkező fájlokat. Ilyenkor a fordítási folyamat teljesen nulláról indul újra következő alkalommal és így újraépülnek a különböző referenciák.

Probléma lehet még az optimalizációval, ezt a Project->Properties ablakban állíthatjuk át. Számunkra az egyetlen nem elfogadható lehetőség az "Optimize for size -Os".