19. Az EEPROM nem felejtő memória használata

Írta: Veréb Szabolcs

Lektorálta: Szabó Ádám, Trádler Máté

NEM FELEJTŐ MEMÓRIATÍPUSOK

A korábbi tananyagrészekben már megismerkedhettünk a mikrovezérlőkben megtalálható felejtő, illetve nem felejtő memóriák egy részével. A felejtő memóriák közé tartozott például a RAM (Random Access Memory), amely nagy sebességű adathozzáférést biztosít, így tökéletesen megfelel a változók, műveletek eredményeinek, részeredményeinek tárolására. Azonban a tápfeszültség megszűnésekor elveszti információtartalmát, így ha a programkódunkat a RAM egységben tárolnánk, akkor minden egyes újraindítás után újra fel kéne töltenünk a kódunkat, ami a legtöbb esetben nem kivitelezhető.

Képzeljük csak el, hogy milyen abszurd lenne, ha az autónk slusszkulcsának minden egyes elfordításakor ki kéne hívni a márkaszervíztől egy embert, aki felprogramozná az összes autóban található mikrovezérlőt. Emiatt szükség van egy nem felejtő memóriára, amiben tároljuk a megőrzendő információkat, például a programkódot.

Erre a célra fejlesztették ki a ROM-ot, azaz a Read Only Memory-t. Ahogy a nevéből is következik, ez a memóriatípus csak olvasható volt, a gyártás során feltöltötték értékekkel/programkóddal a memóriát, melynek módosítására a későbbiekben semmilyen lehetőség nem volt. Érezhetjük, hogy ez óriási megkötést jelentett felhasználói oldalról, hiszen csak egy előre definiált feladatra lehet használni a memóriát. Ennél kicsit szabadabb mozgásteret biztosít a PROM (Programmable ROM), melyet már maga a felhasználó programozhatott fel egy általa kitalált feladatra. Hátránya, hogy OTP (One Time Programmable) memória volt, így csak egyetlen egyszer lehetett programozni, azaz bármilyen hiba esetén a memória használhatatlan lett. Például ha a programkódunkban a+b helyett a-b került be, azaz rossz utasítást adunk ki a processzornak, és ezt a programozás előtt nem vettük észre, akkor az egész memória mehetett a szemétbe. Illetve ha később egy új funkciót vagy teljesen új feladatot akartunk volna felprogramozni, ahhoz is egy új memóriaegységre lett volna szükség. Ez arra hasonlít, mintha új eszközt, de legalább egy új winchestert kellene vennünk ahhoz, hogy feltelepítsünk egy új szoftvert a számítógépünkre.

Ezt követte az EPROM (Erasable PROM), melynek tartalmát ultraibolya fénnyel lehetett törölni, ami már nagykörű szabadságot adott a funkciók bővítésére, azonban a memória törléséhez egy speciális fényforrásra volt szükség, ami még mindig korlátozó tényező volt. Végül az áttörést az EEPROM (Electrically Erasable PROM) hozta meg, melyet már elektromos áram segítségével lehetett programozni és törölni is. Habár ennek a memóriáknak a nevében benne van a “csak olvasható” szókapcsolat, ez már csak annak eredetére utal, hiszen tudjuk írni, olvasni és törölni is.

Kitekintés

Az írás olyan művelet, ahol a memóriában lévő adatcsoportokat (ebben az esetben bájtokat) tetszőleges értékűre állítjuk. A törlés egy olyan speciális írás, ahol az adatcsoportokat csupa egyessel töltjük fel, amely bájtoknál 0b11111111 vagy 0xFF-et jelent.

Az EEPROM-mal tehát van egy olyan memóriánk, melyben tárolhatunk olyan adatokat, programkódot, amelyekre újraindítás után is szükségünk van. Sajnos ez a tulajdonság a sebesség rovására megy, így ezen a téren hátrányt szenved a RAM-mal szemben.

Az ATmega16A esetében a RAM írása és olvasása két processzorórajel alatt végbemegy, amely 8MHz esetén 0.25 µs-ot jelent, az EEPROM esetében ugyanez 8.5 ms, ami 34000-szer lassabb. Ezt valahogy úgy képzelhetjük el mint a rövid és hosszú távú memóriát (emlékezést): gyorsan meg tudunk jegyezni dolgokat rövid időre, viszont ahhoz, hogy tartósan emlékezzünk egy-egy információra, azt hosszabban kell feldolgoznunk.

A FLASH ÉS AZ EEPROM

Egy újabb nem felejtő memória típus a Flash, ami az EEPROM egy továbbfejlesztett változata, a memória tárhelyének növelése céljából a méret megtartása mellett. Ez az adatsűrűség növelését jelenti, amit úgy érnek el, hogy megspórolják a törlésre, írásra és olvasásra szolgáló részáramkörök nagy részét, és azt blokkosítva (több bájtot összekapcsolva) valósítják meg. Így egyetlen bájtot csak úgy lehet vele törölni, hogy kimásoljuk az egész blokk tartalmát, kitöröljük a blokkot, majd az egy bájt kivételével visszaírjuk a többit. Ez érezhetően lassabb, mint egyetlen bájt törlése az EEPROM esetében.

Írás és olvasás esetén egyszerre egy egész blokkhoz hozzáférünk, így nagyobb adatcsomagok esetén gyorsabb mint a sima EEPROM. Ebből is látható, hogy a Flash memóriát nem egyes bájtok vagy értékek tárolására és hozzáféréséhez fejlesztették ki, hanem nagyobb mennyiségű adatokhoz, így például tökéletesek arra, hogy az egész programkódot a programozás során beleírjuk, majd a mikrovezérlő bekapcsolásakor onnan adatokat töltsünk az annál gyorsabb fozzáférésű RAM-ba.

Az EEPROM ezzel szemben egy-egy érték hozzáférését teszi könnyen elérhetővé, ami alkalmassá teszi paraméterek, kalibrációs értékek tárolására. Ezek a memóriák a RAM-mal ellentétben elméletileg is véges élettartammal rendelkeznek, melyet a garantált írások és törlések számával adnak meg. Ez az írás-törlési ciklusszám jellemzően nagyobb az EEPROM-ok esetében a Flash-hez képest, például a használt ATmega16A esetében a Flash 10.000, míg az EEPROM 100.000 ciklust visel el garantáltan. Ez azt jelenti, hogy ennyiszer módosíthatjuk a memóriaterület minden egyes bit-jét.

AZ EEPROM FELHASZNÁLÁSI LEHETŐSÉGEI

Bizonyos feladatoknál szükség lehet arra, hogy a memóriában rendelkezésünkre álljon egy paraméter, ami megőrzi az értékét az újraindítások során. Amennyiben szükség van arra, hogy ezt az értéket a program futása közben módosítsuk, úgy nem elégséges a programkódban tárolni a paramétert. Bár egy konstans

megtartja értékét minden újraindítás esetén, az nem változtatható meg, továbbá egy változó értékét ugyan módosíthatjuk, de az az újraindítás után elvész.

Megoldást jelentene erre a problémára az, ha a program futása közben tudnánk módosítani magát a programkódot, így a benne található konstansok értékeit is, ez azonban komoly problémákhoz vezethet, hiszen ezzel megszűnne a program lefolyásának kiszámíthatósága. Akár olyan részekbe is belenyúlnánk (nem feltétlenül szándékosan), ami teljesen megváltoztatja a programunk működését. Ezt ki lehetne küszöbölni, ha az ilyen paraméterek tárolására lenne egy elkülönített memóriaterület, melyet kétféleképp valósíthatunk meg: külön hardveres kialakítsú, vagy szoftveresen emulált.

Utóbbi azt jelenti, hogy a Flash memóriának egy részéhez hozzáférhetünk futásidőben, a többi részéhez viszont nem. Előfurdulhat, hogy a mikrovezérlő gyártója megengedi a teljes Flash-hez való hozzáférést, ekkor a programozónak kell gondoskodnia arról, hogy a módosított memóraiterület elkülönüljön a programkódot tartalmazó résztől. Így tettek az ATmega16A esetében is, ahol a 16KB-os Flash teljes egészében módosítható futás közben is, melyet egy 512B-os EEPROM egészít ki. Ha a mikrovezérlőben van külön EEPROM, akkor érdemes azt használnunk, hiszen annak élettartama általában egy nagyságrenddel nagyobb.

AZ EEPROM ALKALMAZÁSI TERÜLETEI

Az EEPROM-ot általában olyan konfigurációs paraméterekre szoktuk használni, ami az üzembe helyezéshez szükséges vagy a végfelhasználó tetszőlegesen módosíthatja, esetleg a használat tevékenységét rögzíti (loggolja).

Erre lehet példa mondjuk egy légkondicionáló rendszer távirányítón beállított hőmérséklete, melyet frusztráló lenne a távirányító/légkondi minden bekapcsolásakor újra és újra beállítani, ezért a távirányító EEPROM-jában eltároljuk azt. Vagy például olyan szoftvert írunk, ahol a soros port sebessége állítható, és nem akarjuk, vagy nem tudjuk minden indítás után elvégezni a beállítást, akkor az EEPROM-ban egyszerűen eltárolhatjuk az értékét.

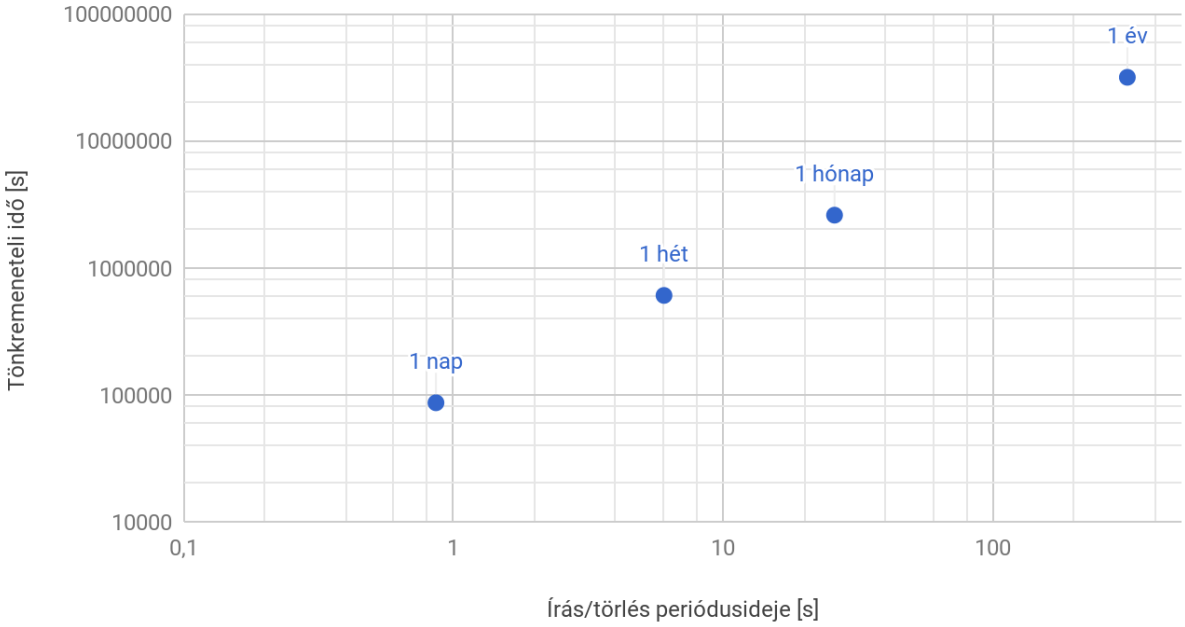
Ipari alkalmazások esetén, ahol fontos nyomonkövetni az eszközzel kapcsolatos abnormális viselkedéseket, az egyes hibakódokat eltárolják az EEPROM-ban, így a szervizben könnyen megállapítható a tönkremenetel tényleges oka (hasonlóan a repülőgépek feketedobozához). Szintén EEPROM használatos szenzorok kalibrációnak eltárolására, amire jó példa a számítógéphez kapcsolt joystick vagy kormányrendszer, hiszen üzembehelyezéskor - amikor először csatlakoztatjuk a számítógéphez az eszközt - be kell tanítanunk a vezérlő szervek maximális kitérését, ezekre az értékekre a későbbi bekapcsolások során emlékezik a rendszer, és tetszés szerint a kalibráció újbóli végrehajtásával felülírhatóak. Egy időjárás figyelő berendezés esetén fontos lehet elmenteni minden nap a legalacsonyabb és legmagasabb hőmérsékletet, legnagyobb szélsebességet, és nem akarjuk, hogy ezek az értékek áramkimaradás esetén elvesszenek.

Tárolhatunk benne felhasználói beállításokat, például egy motorvezérlő áramkörben az éppen rákötött motor paramétereit: tekercsellenállás, induktivitás, maximális áram, -fordulatszám, -teljesítmény, szabályozási paraméterek. Így nem kell minden bekapcsoláskor ezeket az értékeket újra és újra beírnunk, de egy másik motor csatlakoztatásakor azokat könnyen módosíthatjuk, akár az általunk tervezett motorvezérlő által mért értékekkel.

Egy beléptető rendszer esetében az azonosító eszköz EEPROM-jában eltárolhatjuk a belépési kódunkat, míg a beléptető oldalon eltárolhatjuk az adott azonosítóval végrehajtott belépések számát és az utolsó belépés idejét, esetleg a hibás azonosítások számát is.

A MEMÓRIA ÉLETTARTAMA

Remélem sikerült érzékeltetni, hogy az EEPROM memória mennyi minden dolog tárolására alkalmas, igazából csak a saját képzelőerőnk szab ennek határt. Azonban az EEPROM használata során észben kell tartanunk néhány dolgot. A legfontosabb, hogy a memóriánk véges írási és törlési számmal rendelkezik, tehát ha nem akarjuk rövid időn belül felélni az élettartamát, akkor csak szükséges esetben végezzünk írási műveletet. Azt, hogy mi számít gyakori írásnak, a memóriánk ciklusszáma határozza meg, az ATmega16A esetében ez 100.000 írási és törlési ciklust jelent. Ha másodpercenként tárolnánk egy változó értéket, akkor alig több mint 1 nap alatt elhasználnánk a memóriánkat, egy napban ugyanis 86.400 másodperc van. Ha csak 315 másodpercenként írunk bele, akkor 1 évet is garantáltan kibír a memória.



***1. ábra** - ATmega16A EEPROM-jának tönkremenetele az írás gyakoriságának függvényében*

AZ ÉLETTARTAM NÖVELÉSÉNEK MÓDSZEREI

Tehát az EEPROM nem való gyorsan változó értékek eltárolására, és nincs is értelme folyamatosan írni vagy törölni, hiszen elsődleges célja a kikapcsolt állapotok közötti adattárolás. A memória élettartamának növelésére többféle lehetőségünk van.

Az egyik legkézenfekvőbb az, ha megpróbáljuk elkerülni a felesleges írásokat, azaz csak akkor írjuk felül az adott bájt tartalmát, ha az eltér a mostanitól. Ez azt jelenti, hogy minden írás előtt kiolvassuk a memória tartalmát, összehasonlítjuk a beírandó értékkel, és szükség esetén elvégezzük az írást. Ez a módszer növeli az íráshoz szükséges időt abban az esetben ha az tényleg szükséges, viszont csökkenti ha azonos értéket

akarunk tárolni. Emiatt nem érdemes olyan esetekben használni, ahol biztosított az adatok különbözősége (ilyen például a számlálás).

További élettartam növelési módszer lehet, ha tudjuk, hogy az áramkörünk mikor kapcsol ki, és még azelőtt a fontos megtartandó értékeket, amiket eddig a RAM-ban tároltunk, kiírnánk az EEPROM-ba, majd a következő bekapcsoláskor onnan visszatöltenénk őket a RAM-ba. Ilyen tápkimaradást érzékelő áramkör építhető, viszont arra különösképp ügyelni kell, hogy a teljes írási folyamat alatt a mikrovezérlő és az EEPROM tápfeszültsége az adatlapjukban leírt minimális érték fölött maradjon, ugyanis ez alatt a feszültségszint alatt nem meghatározott a memória és a vezérlő működése, így hibás adat kerülhet a memóriába, vagy meg sem történik az írás.

A másik lehetséges élettartam növelés feltétele, hogy a memória tárhelyének csak egy kis részére legyen szükségünk, ami általában igaz is. Tegyük fel, hogy az 512B-os EEPROM-ból csak 30 bájtra van szükségünk. Ekkor alap esetben 100.000 irási ciklus után lenne 30 bájtnyi területünk, amely az élettartama végénél jár, és lenne 482 bájtnyi memóriánk, ami eddig érintetlen volt és még 100.000 írási ciklust vígan elviselne. Botorság lenne nem kihasználni ezt a maradék területet.

A gyakorlatban az az eljárás, hogy megpróbáljuk az elhasználódás mértékét kiegyenlíteni a teljes memóraterületen, tehát minden egyes íráskor más-más memóriacímre írunk. Felbontjuk a memóriánkat jelen példa esetén 30 bájtos blokkokra, összesen 17-re, és minden egyes írást a soron következő blokkon végzünk el (az utolsó után az első következik).

Ezzel sajnos még nem vagyunk kész, ugyanis egy kikapcsolást követően tudnunk kell, hogy melyik blokkba, milyen memóriacímre kell írni a soron következő adatot. Ezt eltárolhatjuk egy pointer-be, aminek szintén az EEPROM-ban kell lennie. Kézenfekvő lenne azt mondani, hogy ez a pointer kerüljön egy fix helyre a memóriába, ahonnan minden bekapcsoláskor ki tudjuk olvasni az aktuális blokknak a címét, ezzel azonban nem jutnánk semerre. A pointer-t minden egyes blokk írásakor frissíteni kell a következő blokk címére, és ha azt egy fix helyen tároljuk, akkor a pointer memóriaterületét ugyanolyan nagy ütemben használjuk, mint egyszerű esetben.

Erre az a megoldás, hogy nem egy pointer-t tárolunk, hanem a memóriában egy speciális mintával jelezzük a soron következő blokkot. Mivel az EEPROM olvasás közben nem használódik el, így írhatunk egy algoritmust, ami bekapcsoláskor végigolvassa az egész EEPROM-ot a meghatározott mintát keresve, ha megtalálta, akkor onnan kell folytatnia az írást, különben pedig a memória elejéről (valószínűleg ez az első adat felvitele). Ez a meghatározott minta pedig célszerűen a 0xFF-fel feltöltött (azaz törölt) blokk. Ez a kód jelenti az üres bájtot az EEPROM-ok esetében, így érdemes olyan módon tárolni az adatokat, hogy a 0xFF ne jelentsen más értelmes értéket.

Tehát egy írási folyamat úgy néz ki, hogy az aktuális blokkot feltöltjük a fontos értékekkel, és a következő blokkot pedig töröljük (feltöltjük 0xFF-ekkel). Így minden írás során két blokkot írunk, így 30 bájtos blokkok esetében a 512 bájtos EEPROM élettartamát nem 17-szerezzük, hanem “csak” 8,5-szerezzük. Ha mind a két módszert alkalmazzuk, akkor jelentős mértékben növelhetjük az EEPROM élettartamát. Természetesen, ha olyan értéket tárolunk, amely nagyon ritkán változik (naponta 1-2-szer), például kalibrációs értékek, szabályozó paraméterek, akkor nincs feltétlenül szükség ilyen élettartam bővítő funkciók implementálására.

Kitekintés

Elméleti példa lépésszámlálóra

A lépésszámláló feladata, hogy megszámolja viselőjének lépéseit, és addig meg is jegyezze a lépésszámot, míg a felhasználó azt le nem nulláza. A lépésszámlálónak a gyártás óta megtett összes lépést is tárolnia kell. A számláló akkumulátorról működik, egy 256B-os EEPROM található benne 100.000 írási ciklusszámmal, az aktuális lépésszámot 24 bit szélességen tároljuk, az összesített lépésszámot pedig 32 biten. A lépéseket napközben ébrenlét alatt akarjuk számolni, melynek időtartama 14 óra. Számoljuk ki a garantált élettartamot alapesetben, kikapcsoláskori mentéssel, illetve a két módszer együttes alkalmazásával! Az egyszerűség kedvéért két lépés között legalább 0,5 másodperc telik el.

Megoldás:

A reset gomb megnyomásakor az EEPROM-ban található összesített lépésszámhoz hozzá kell adni az aktuális értéket, majd nullázhatjuk azt. Az átlag felhasználó arra kíváncsi, hogy egy nap alatt mennyit lépett, így feltételezhetjük, hogy naponta egyszer nullázza a számlálót, azaz napi egyszer írjuk az összesített értéket. Ez olyan nagy periódusidő, hogy érdemes teljesen függetlenül kezelnünk az aktuális lépésszámtól, 4 bájtnyi helyet elkülönítve neki. Ennek a 32 bitnek az élettartama így:

$$T = 24[h]/1 \cdot 100.000 \; = \; 2.400.000\,[h] = 274\,[év]$$

Alapeset:

Minden változást az EEPROM memóriában tárolunk. Ez azt jelenti, hogy minden lépéskor, azaz fél másodpercenként növeljük az EEPROM-ban található értéket, azaz ennyi időközönként írjuk a memóriát. Ez azt jelenti, hogy az EEPROM élettartama

$$T = 0.5[s] \cdot 100.000 \; = \; 50.000\,[s] = 13.89\,[h]$$

Azaz felfelé kerekítve mindössze 14 óra, ami mondjuk napi 2 óra aktivitást feltételezve nem egészen 7 nap. Nem biztos, hogy ezt az élettartamot akarnánk az eszköz dobozán látni.

Mentés kikapcsoláskor:

A lépésszámláló érzékeli a tétlenséget követő lépéseket mely hatására bekapcsol, és hosszú tétlenség esetén kikapcsol az akkumulátoros üzemidő növelése céljából. Bekapcsoláskor kiolvassa az EEPROM aktuális értékét, működés közben a RAM-ban számolja magának a lépéseket, majd tétlen állapotba lépés esetén kikapcsolás előtt beleírja az aktuálisat az EEPROM-ba. Tegyük fel, hogy a felhasználó az egészségére gondolva minden negyedórában mozog 5 percet, azaz a 14 óra ébrenlét alatt összesen 56-szor kapcsol be a lépésszámláló 5 perces időtartamokra (utóbbi igazából lényegtelen, lehetne több vagy kevesebb is). Ebből az következik, hogy naponta 56-szor írja felül az EEPROM értékét, így a teljes élettartam

$$T = 24[h]/56 \cdot 100.000 \; = \; 42.857\,[h] = 4.89\,[év]$$

Ezzel a megoldással már majdnem 5 évet képes üzemelni a lépésszámlálónk, ami impozánsabb az 1 hétnél.

Mentés kikapcsoláskor és adat cirkulálás:

A lépésszámot 3 bájton tároljuk, így a 252 bájtos memóriaterületet (ugyebár 4 bájtot külön kezelünk az összesített értéknek) 252/3=84 blokkra oszthatjuk fel. Mivel egy körbefordulás alatt minden bájtot egyszer írtunk és egyszer töröltünk, tehát két ciklust viszünk bele, így 84/2=42-szer nagyobb élettartamot érünk el, mint az előző megvalósításban, azaz nagyjából 205 évet.

Ebből a példából látható, hogy az EEPROM írásával óvatosan kell bánni a hosszú élettartam érdekében, továbbá a rendszert néhány funkcióval kiegészítve drasztikusan lehet növelni azt.

AZ EEPROM BEMUTATÁSA ATMEGA16A VEZÉRLŐVEL

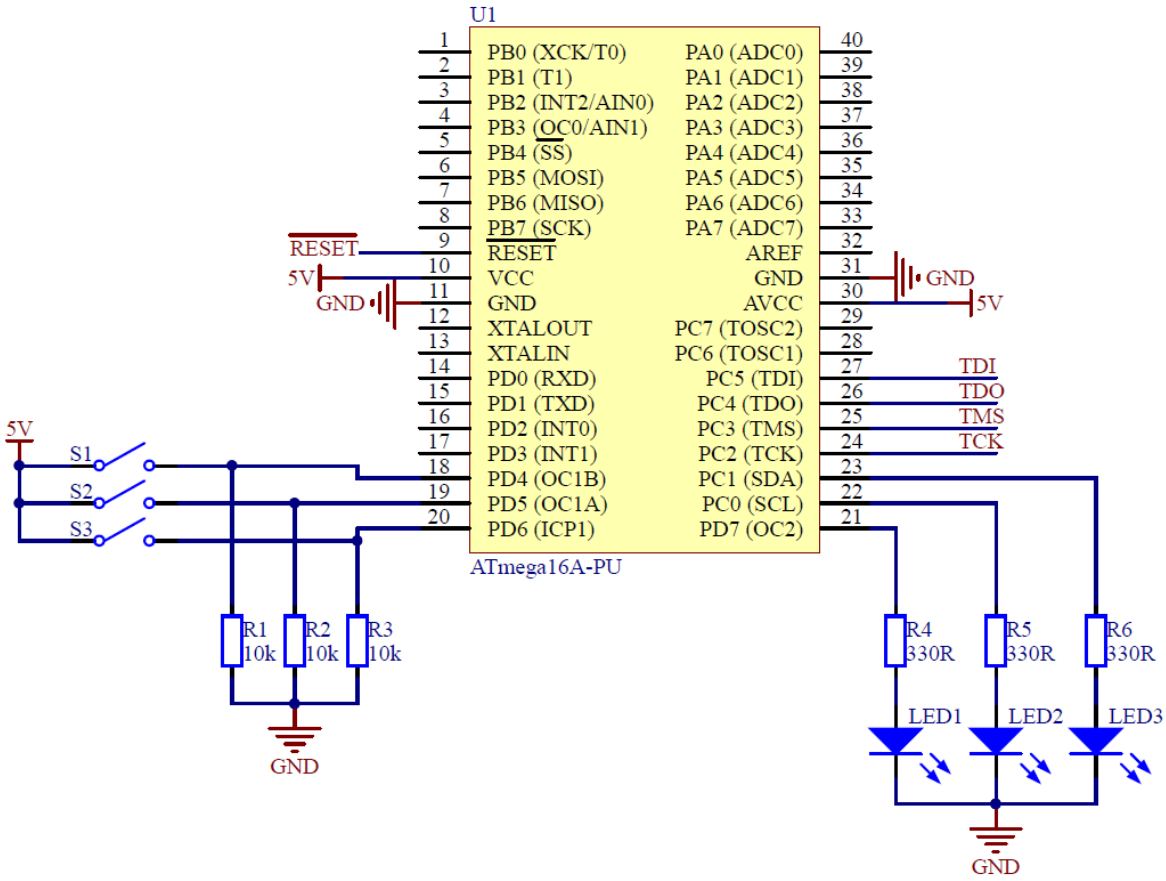
A következőkben az EEPROM gyakorlati használatát fogom ismertetni egy egyszerű példa segítségével. Az ATmega16A adatlapjában a 18-23. oldalon találunk információt és útmutatást az EEPROM használatához. Itt részletesen leírják, hogy a mikrovezérlőnek pontosan mely regisztereit és hogyan kell vezérelni az EEPROM írás és olvasása céljából. Ez három regisztert jelent, úgy mint EECR (EEprom Control Register), EEAR (EEprom Address Register) és EEDR (EEprom Data Resiter).

Az első tárolja az EEPROM beállításait, illetve azt, hogy milyen milyen művelet (írás, olvasás) meg végbe épp az EEPROM-on, a második egy cím regiszter, ami az írandó/olvasandó memóriaterület címe, a harmadik pedig maga az adat, amit beleírunk, vagy épp kiolvastunk az EEPROM-ból. Ezek használatát a továbbiakban nem fejteném ki, ugyanis az Atmel Studio biztosít egy EEPROM könyvtárat a programozók számára, melyben kész függvények segítségével egyszerűen kezelhetjük a memóriát.

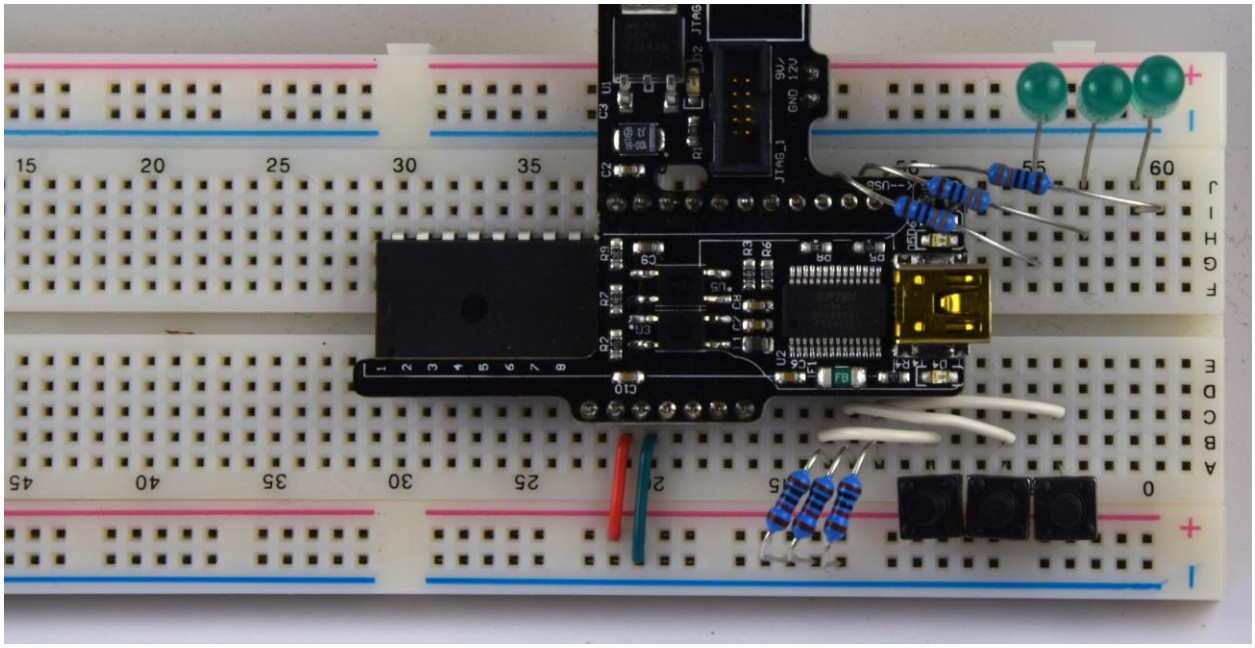
Az `eeprom_read_byte()` és `eeprom_write_byte()` függvények szolgálnak egy-egy bájt írására és olvasására, de ezeknek létezik *word* (16bit), *dword* (32bit), *float* és *block* (n db bájt) változata is, továbbá a könyvtárban megtalálható az `eeprom_update_byte()` függvény (és fentebb felsorolt variációi), ami csak eltérés esetén írja felül a bájt tartalmát. Az `eeprom_update_block()` függvénynek három argumentuma van: tömb címe, EEPROM kezdő memóriacím, és tömb hossza. A metódus egyesével átmásolja a bájtokból álló tömb első n darab elemét a megadott kezdeti EEPROM címtől kiindulva. Csak akkor módosítja az EEPROM egyes bájtjait, ha az ott lévőtől eltérő értéket akarunk beleírni, azaz szükségszerű az írás.

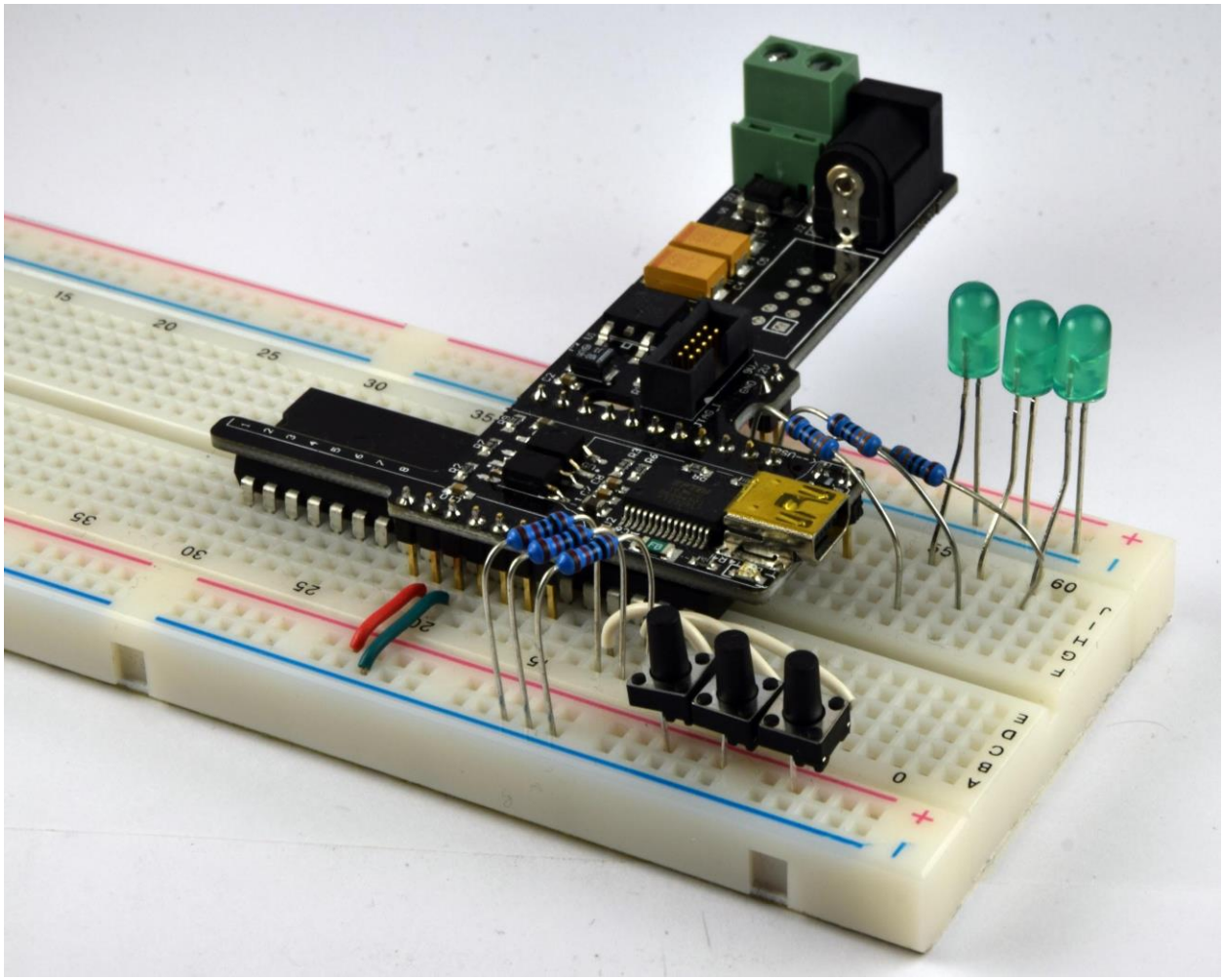
GYAKORLATI PÉLDA

Egy egyszerű példán keresztül nézzük meg, hogyan lehet pár - esetünkben három - konfigurációs beállítást eltárolni egy EEPROM segítségével. A három paramétert egy-egy nyomógomb segítségével állíthatjuk, kijelzésüket pedig ugyancsak három darab LED segíti. A példában nem adtunk a három konfigurációs értéknek külön nevet, de lehetnek például egy soros kommunikáció (UART) beállításai, mondjuk 1: paritás bit engedélyezve, 2: páros vagy páratlan paritásbit, 3: egy vagy két STOP bit. Természetesen bármi mást is mögé képzelhetünk. Töltsük be az Atmel Studio-ba a *18_1_EEPROM* projektet, nyissuk meg a *main.c* fájlt és vizsgáljuk meg a kódot. Az adatlapban található lábkiosztás alapján végezzük el a fejlécben leírt bekötéseket.



2. ábra *- Az EEPROM példakódhoz szükséges kapcsolási rajz*





***3. ábra** - EEPROM példakódhoz szükséges bekötés breadboard-on*

FONTOS: mielőtt feltöltjük a kódot a mikrovezérlőre, kommenteljük ki az összes `eeprom_write_byte()` parancsot. Ha valamit rosszul kötöttünk be, akkor előfordulhat, hogy "lebegni" fog valamelyik bemenet, ami a LED(ek) villogásához és ezzel együtt az EEPROM sokszori írásához vezet. Ez mindenféleképp elkerülendő. Ha a LED-ek működése az elvártat produkálja, akkor nyugodtan írhatjuk az EEPROM-ot is.

Kitekintés

A "lebegés" azt jelenti, hogy a port nincs egy meghatározott állapotban, azaz logiaki 0-ban vagy 1-ben, hanem a környezetből származó elektromágneses zajok hatására a két érték között folyamatosan, nagy frekvenciával változik.

```c
/*
 * 18. Fejezet
 *
 * A PD4,5,6 pineken levő kapcsolókkal változtatható meg a PD7,PC0, PC1 lábak állapota.
 * Az állapot elmentésre kerül az EEPROMba, így újraindítás után sem veszik el az információ.
 */

#include "../Headers/main.h"

//Az egyes LEDekhez tartozó EEPROM memóriabájtok címei
const uint8_t* led1_ptr = 0x00;
const uint8_t* led2_ptr = 0x01;
const uint8_t* led3_ptr = 0x02;
```

A cél, hogy egy gomb megnyomása megváltoztassa a hozzá tartozó LED állapotát, tehát ha eddig világított, akkor kapcsoljon ki, illetve fordítva. Első lépés gyanánt konstans értékként deklaráljuk az egyes LED-eket vezérlő EEPROM memóriacímeket, azaz pointer-eket. A `led1_ptr` memóriacímen található értéktől fog függeni az 1-es LED vezérlése (azaz hogy világít-e vagy sem), a `led2_ptr` a 2-es LED-nél, a `led3_ptr` pedig a 3-as LED-nél ugyanígy. Az ATmega16A EEPROM-ja 512 bájt hosszú, így a memóriacímek 0 és 511 vagy hexadecimális számokkal 0x000 és 0x1FF közötti értéket vehetnek fel. A példában az egyszerűség kedvéért az első három bájtot címezzük meg.

```c
int main(void)
{
    //PORT-ok inicializálása
    IOInit();

    //led1_ptr címen levő eeprom adat vizsgálata
    if (eeprom_read_byte(led1_ptr) > 0)
    {
        sbi(PORTC, 1);
    }

    //led2_ptr címen levő eeprom adat vizsgálata
    if (eeprom_read_byte(led2_ptr) > 0)
    {
        sbi(PORTC, 0);
    }

    //led3_ptr címen levő eeprom adat vizsgálata
    if (eeprom_read_byte(led3_ptr) > 0)
    {
        sbi(PORTD, 7);
    }
```

A programunk elején elvégezzük a ki és bemenetek inicializálását, melyre az `IOInit()` függvény szolgál. Az EEPROM-ban tárolt 0 érték jelzi, hogy a LED-et ki kell kapcsolni, ettől eltérő érték esetén bekapcsoljuk azt. Az EEPROM-ok törölt állapotban szoktak lenni az első használatkor, ez azt jelenti, hogy minden bájtja 0xFF-et tartalmaz. A portok inicializálása után kiolvassuk az EEPROM-ban tárolt három számot, és a bennük talált értékek alapján beállítjuk a LED-eket. A kiolvasásra az `eeprom_read_byte()` függvényt használjuk melynek egyetlen argumentuma az olvasandó memóriacímet tartalmazza, visszatérési értéke pedig a megadott memóriacímen lévő bájt értéke. Ezzel el is érkeztünk a programkódunk fejlécének végére, és következik a ciklikusan futtatandó rész, a gombok ellenőrzése. A `tbi()` visszaadja az adott

bemenet értékét, az `sbi()` logikai magas, míg a `cbi()` logikai alacsony értékre állítja az adott kimenetet.

```
//Végtelen ciklus
while (1)
{
    //Első kapcsoló kezelése, ha le van nyomva
    if (tbi(PIND, 4) != 0)
    {
        //Tranziensek lezajlanak ennyi idő alatt, pergésmentesítés
        _delay_ms(BTN_DELAY);

        //Várakozás a felengedésre
        while (tbi(PIND, 4) != 0);

        //Tranziensek lezajlanak ennyi idő alatt, pergésmentesítés
        _delay_ms(BTN_DELAY);

        //Ha a PC1 lábon levő LED világít
        if (tbi(PINC, 1))
        {
            //Az új állapot a "nem világít", ezt elmentem az EEPROMba
            eeprom_write_byte(led1_ptr, 0x00);
            //Kikapcsolom a LED-et
            cbi(PORTC, 1);
        }

        //Ha a PC1 lábon levő LED nem világít
        else
        {
            //Az új állapot a "világít", ezt elmentem az EEPROMba
            eeprom_write_byte(led1_ptr, 0xFF);
            //Bekapcsolom a LED-et
            sbi(PORTC, 1);
        }
    }

    if (tbi(PIND, 5) != 0)
    {
        _delay_ms(BTN_DELAY);
```

Mind a 3 nyomógombra külön-külön reagál a rendszer, de lényegében azonos módon, csak a módosított EEPROM cím és a kimeneti port különbözik. Az első *if*-be akkor lépünk bele, ha a nyomógomb megnyomott állapotba kerül, azaz a PD4-es lábat logikai magas értékbe húzza.

Kitekintés

A nyomógombok esetében nem valósítottunk meg hardveres pergésmentesítést, viszont el akarjuk kerülni a jelenség negatív hatásait (például, hogy a pergés alatt minden fel és lefutó él esetén feleslegesen írjuk az EEPROM-ot, ezzel csökkentve annak élettartamát). Ekkor meg kell oldanunk a jel szoftveres pergésmentesítését. Ennek egyik módja, hogy lemérjük, hogy mekkora az a T idő, amely alatt a pergés már biztosan lezajlik, majd az első élváltástól ekkora T időt várakoztatjuk a programunkat a jel további

kiértékeléséig, azaz késleltetést rakunk az első észlelt élváltás után. Ezt valósítja meg a `_delay_ms()` metódus, ami az argumentumába írt számnak megfelelő ezredmásodpercet várakozik.

A nyomógomb megnyomása után meg kell várnunk annak felengedését, ugyanis ekkor fejeződik be egy teljes gombnyomás. Majd visszaolvassuk a digitális kimenetünk értékét, és negáltjára változtatjuk az EEPROM írása mellett. Az `eeprom_write_byte()` függvénynek két argumentuma van, az első azt adja meg, hogy mely memóriacímre akarunk írni, míg a második a beírandó értéket tartalmazza, visszatérési értéke nincs.

HÁZI FELADAT

Ez a példakód egyszerre csak egy gomb megnyomását érzékeli, próbáld megvalósítani azt, hogy mind a három nyomógombot akár egyszerre is érzékelje. Tartsd észben, hogy az EEPROM írása relatíve lassú folyamat, és egyszerre csak egy memóriaterülethez férhetsz hozzá.

Hasznos tippek

Amíg nem győződtél meg az összeállított kapcsolásod, illetve a kódod helyes működéséről érdemes kikommentezni az EEPROM műveleteket, és azokat szokványos változókkal helyettesíteni.

Ha nem fontos, hogy az EEPROM írása minél hamarabb végbemenjen, vagy nem biztosított szoftveresen, hogy különböző érték kerüljön az EEPROM-ba, akkor érdemes az `eeprom_write_xxxx()` helyett az `eeprom_update_xxxx()` függvényeket használnod.

A kapcsolók a nyomógombokkal ellentétben mechanikailag megőrzik állapotukat, így az esetükben nincs szükség az állapot elektronikus tárolására. Hátrányuk viszont, hogy szoftveresen, például egy kommunikációs csatornán küldött üzenettel nem tudjuk módosítani az általuk beállított értéket, míg az EEPROM-ban tárolt értékeket igen. Ezen felül mechanikai kapcsolókkal a számok tárolása is nehézkes (a szám bináris helyiértékeinek egy-egy kapcsoló felel meg, tehát egy 16 bites számhoz 16 kapcsoló kell).

ÖSSZEFOGLALÁS

Ebben a részben megtanulhattuk az EEPROM alapvető használatát, illetve számos példa segítségével körbejártuk annak lehetséges alkalmazási területeit. Az EEPROM-ban lévő lehetőségek további kiaknázását a kapott ismeretek alapján már te is el tudod végezni.