SZAKDOLGOZAT

Koncz Tamás

Debrecen

2012

Debreceni Egyetem
Informatikai Kar

Költséghatékony, intelligens házfelügyeleti rendszer kialakítása PIC mikrovezérlővel

Témavezető:
Vitéz Attila
Tanársegéd

Készítette:
Koncz Tamás
Mérnök Informatikus

Debrecen
2012

Tartalomjegyzék

Bevezetés

Mikor ezt a témát választottam a kíváncsiság vezérelt, úgy fogtam bele, hogy képről se tudtam volna felismerni egy mikrokontrollert. Most meg minél többet tudok róluk, annál bonyolultabb rendszereket vagyok képes megalkotni. Beléptető biztonsági rendszereket és különböző termosztátokat láttam már, és mindig is kíváncsi voltam, mi folyhat az agyukban. Elektronikai alapismereteim se voltak nagyok, áramköröket, nyomtatott áramköröket sem építettem. A párhuzamos és soros kapcsolásokról voltak alapvető fizikai ismereteim, ezekre tudtam alapozni. Ahogyan szép lassan belemerülök, egyre test közelibbnek érzem a témát, olyannyira, hogy mesterképzésre a hardverprogramozási szakirányt választom ki célul.

A témavezetőm által javasolt PIC16F628-as mikrokontrollert használtam a kitűzött célok megvalósítására. Az egész történet azzal kezdődött, hogy az internetet fésültem, hogy megtudjam, hogyan néz ki egy mikrokontroller és hogyan lehet beprogramozni, de mielőtt ebbe belemerülnénk, nézzük meg, mi is az épületfelügyeleti rendszer.

Az épületfelügyelet több részegységből áll, és e részegységek együttesen, egymással összhangban működve alkotják az épületfelügyeleti rendszert:

- A tűzjelző rendszerek értesíthetik a tűzoltókat, meghatározhatják a tűz helyét, megkezdhetik az oltást, de egyben mind megegyeznek, valamilyen formában jeleznek, ha tüzet érzékelnek.
- A behatolás jelző rendszerek értesíthetik a rendőrséget, értesíthetik a tulajdonost, meghatározhatják a behatolás helyét, jeleznek, ha behatolást észlelnek.
- A videó megfigyelő rendszerek rögzíthetik a megfigyelt területeket, felismerhetik az emberi arcokat, felismerhetik az agresszív viselkedést, de mind megfigyelik a megfigyelendő területet.
- A beléptető rendszerek rengeteg formában léteznek: pin kódos beléptető, kártyabeolvasó, retinaszkenner, arcfelismerő, ujjlenyomat leolvasó stb. Mindegyik beengedi a belépő személyt, ha megfelelt a belépési kritériumoknak.
- Egyéb rendszerek:
    - Evakuációs rendszerek
    - Hőszabályzó rendszerek
    - Ügyfélfogadó rendszerek
    - Redőnyvezérlő rendszerek

- Páratartalom szabályzó rendszerek
- Lámpákat vezérlő rendszerek

Egyes cégek az összes általam említett rendszerre kínálnak megoldásokat, így megbízhatóak egy épületfelügyeleti rendszer kiépítésével. Ilyen cégek pl.: Bosch, Elcon, Schneider Electric, Alarm 2000 Kft

Az épületfelügyeleti rendszerek fő vevőköre a nagyobb intézmények, multinacionális cégek, ahol mindenre kiterjedő felügyelet szükséges. Ezek a jellemzőbb alkalmazási területek:

- Üzleti életben: bankok, kiállítótermek, konferenciaközpontok, egészségügyi központok, bevásárlóközpontok, kiskereskedelmi láncok.
- Iparban: autóipar, vegyipar, gyémántipar, gyárak, üzletek.
- Állami és Közigazgatási Szervek: börtönök, kormányhivatalok, múzeumok, egyetemi épületegyüttesek, hadiipari létesítmények.
- Szállításban és logisztikában: repülőterek, vasútállomások, raktárak és logisztikai létesítmények.

A háztartásokban nem jellemző egy teljes épületfelügyeleti rendszer kiépítése, viszont bizonyos alrendszerek elterjedtek pl.: beléptető rendszerek, behatolás jelző rendszerek, hőszabályzó rendszerek.

Építettem egy pin kódos beléptető rendszert egy próbapanel segítségével, van rajta egy ajtónyitás érzékelő. Ugyanezen a panelen építettem meg a termosztátomat is, ami képes a hőmérséklet mérésére és beállítására. Fűteni és hűteni csak szimbolikusan tud, de rá lehetne kötni egy konvektorra, hogy ténylegesen azt vezérelje. Továbbá számítógépen leszimuláltam egy RS232-es kommunikációt 2 mikrokontroller között.

Mielőtt megépítem a próbapanelre az áramköröket, leszimulálom őket számítógépen a Proteus Design Suite, és azon belül az ISIS Professional v7.6 nevű program segítségével, amit a Labcenter Electronics gyártott, így nem kell mindig behelyeznem a mikrokontrollert a programozó áramkörbe, erről a későbbiekben lesz szó, hanem elég módosítanom a programkódot és egyből látom a változást a számítógép képernyőjén.

Alkatrészeimet a feladatokhoz egy debreceni szaküzletben, a Hobby Elektronikában vásároltam. Az áramot az áramkörbe a számítógépem tápegysége szolgáltatta (12 V).

További inspirációt nyújtott a temérdek mennyiségű online videó, amiben a legkülönbözőbb felhasználásokra használták a PIC16F628-at.

PIC16F628 bemutatása

Ebben a fejezetben a PIC16F628 mikrokontrollert szeretném olyan szinten bemutatni, hogy a következő fejezetekben a vele elvégzett feladatok érthetőek legyenek. A következőekben a PIC rövidített megnevezést használom a PIC16F628-ra.

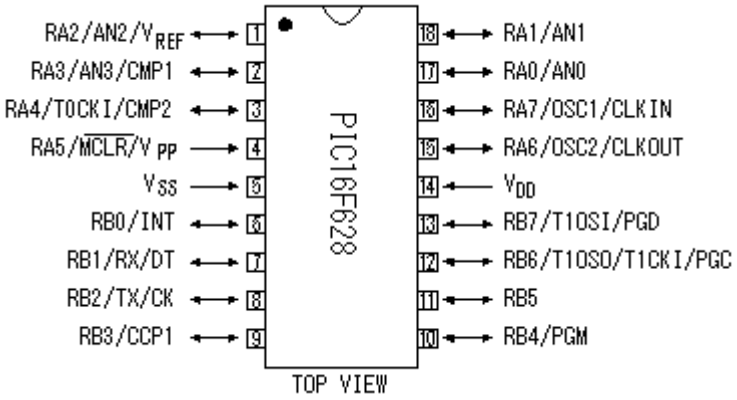
Úgy kell elképzelni, mint egy személyi számítógépet csak sokkal kisebbe. A PIC kapható 4 MHz-es változatban is. Én a 20 MHz-essel dolgoztam. A 20 MHz-es processzorban másodpercenként 20 millió processzor ciklus hajtódik végre. 1 műveleti ciklust ez a processzor 200 ns alatt képes elvégezni, tehát másodpercenként 5 millió műveleti ciklust végez el, azaz 4 processzor ciklus 1 műveleti ciklus. 35 féle művelet végrehajtására képes, amelyek közül az általam használtakat fogom ebben a fejezetben bemutatni. Egy művelet, művelettől függően, 1 vagy 2 műveleti ciklust emészt fel.

Tartalmaz egy 3,5 kByte-os FLASH ram-ot, ami a hexadecimális formátumban feltöltött programkódot tartalmazza. A ráírt programkód az áram megszűntekor nem vész el, szóval ez tekinthető a miniatürizált „személyi számítógépünk winchesterének”.

224 Byte ram található benne, ami az áramellátás megszűntekor törlődik, ez tekinthető a „személyi számítógép memóriájának”. Van még benne 128 Byte EEPROM, ami az előzővel ellentétben megmarad az áram megszűnése esetén és a PIC beállításait, valamint más megőrizendő adatokat tartalmaz. A legnagyobb különbég a PIC és egy PC között, hogy nem lehet működés közben programozni. Méretei igen csekélyek: 23 mm hosszú, 6 mm széles és lábaival együtt 7 mm magas.

Működéséhez egyenáramú 3-tól 5,5 V szükséges, én 5 voltot adtam neki. A számítógép tápegységemből érkező 12 voltot átalakítottam 5 voltra.

18 lába van 1-18-ig fizikailag számozva. Ebből 2 lábat elfoglal az áramellátás: 14-es láb a pozitív és az 5-ös láb a negatív csatlakozó. A maradék 16 láb I/O azaz be- és kimeneti csatlakozó.



Azt, hogy mikor legyen egy láb kimeneti és mikor bemeneti a programkód dönti el. Az I/O lábak eloszlanak A port és B port között. A lábak fizikai címzése szerint 1-4-ig és 15-18-ig az A port részei, 6-13-ig a B port részei. Logikai címzés szerint A0-A7-ig az A port részei, B0-B7-ig a B port részei. Arra, hogy ezeket miféleképpen lehet ki- vagy bemenetre átállítani a következőekben térek ki. A lábak logikai megnevezése eltér a fizikai megnevezéstől. A következőekben mindig logikailag fogom őket megnevezni. Az áramellátásra szolgáló lábaknak nincs logikai megnevezésük. Most nézzük meg, hogy mely fizikai lábakhoz mely logikai lábak tartoznak:

Fizikai láb	**Logikai láb**
1.	A2
2.	A3
3.	A4
4.	A5
6.	B0
7.	B1
8.	B2
9.	B3
10.	B4
11.	B5
12.	B6
13.	B7
15.	A6
16.	A7
17.	A0
18.	A1

A PIC memóriája 4 részből áll, Bank0, Bank1, Bank2 és Bank3, bankonként 128 Byte kapacitással, azaz 128 regiszterrel (1 regiszter 1 Byte). Ezek közül nincs kihasználva az összes Byte (regiszter). A kihasználatlanok írhatatlanok és 0 értéket kapnak. A kihasználtakat vagy a programkód írja fölül, vagy maga a PIC. Ezek a regiszterek fejenként 1 Byte, azaz 8 bit információt hordoznak. Minden regiszterre egy 16-os számrendszerbeli számmal lehet hivatkozni. (Például 86h, ahol a h jelenti, hogy hexadecimális számról van szó.) Hagyományosan bizonyos regisztereket külön névvel illetnek.

Az I/O portok be- vagy kimenetre való állításához a TRISA, valamint a TRISB regisztert kell átállítani, amelyek a Bank1 85h (TRISA) és 86h (TRISB) regiszterei. A regisztereken belül a bitek jobbról balra 0-tól 7-ig vannak megszámozva. A TRISA 0-dik bitjének értéke tartozik az A Port 0-ás lábához és így tovább felfelé minden bit 1 lábhoz tartozik, ugyanez vonatkozik a B Port-ra is. A 0-ás bitérték jelenti a kimenetet, az 1-es bitérték meg a bemenetet, tehát ha például TRISB értéke 00000100, akkor a B2-es láb bemenet, a többi B láb kimenet.

Az adott utasítás mindig csak egy adott bankra vonatkozik, amelyet a STATUS regiszterben tárolt érték határoz meg, így a bankok között lépkedni kell a különböző regiszterek eléréséhez. A Bank0 és Bank1 közötti lépkedéshez elegendő a Status regiszter 5. bitjét módosítani 0-ra, ha Bank0-ba lépünk és 1-re, ha Bank1-be lépünk. A Status regiszter az összes bankból szabadon elérhető.

A regisztereket memóriacím alapján lehet elérni, a szükséges memóriacímek megtalálhatók a PIC használati utasításának 14. oldalán (2. irodalomjegyzék). A memóriacímek hexadecimális számok 00h-tól 1FFh-ig.

Programozása a processzor által ismert 35 db hardverszintű paranccsal lehetséges, ezek közül csak a legfontosabbakat magyarázom el, amelyeket én is használtam. Programozásához az MPLAB szoftvert használtam, amivel létrehoztam egy ASM file-t, amit aztán lefordított az MPLAB HEX file-ra, hexadecimális formátumra, és ez került a PIC-re. Nem muszáj hardver szintű nyelven programozni, lehetséges BASIC, valamint C segítségével történő programozása, viszont ez esetben is egy ASM file majd egy HEX file jön létre. Azért választottam, hogy hardver szinten programozom, hogy jobban megértsem a PIC belső folyamatait és működését. A parancsokat ASM file esetén enter választja el. A program végét jelző szó: end.

Azért írok a programozásról a PIC16F628 bemutatása című fejezetben, mert a processzor által felismerhető parancsok is a PIC részei. A 35 parancs a PIC leírásának 108. oldalán összefoglalva megtalálható (2. irodalomjegyzék).

Az MPLAB-ban írt program elején rendszerint a nevekhez kötött konstansokat illik bemutatni (Pl.: TRISB equ 86h, STATUS equ 03h). Ebben a példában a fentebb említett TRISB és STATUS regiszterek memóriacímét tároljuk le, tehát mostantól nem 86h, illetve 03h-val, hanem az emberileg befogadhatóbb TRISB, illetve STATUS névvel tudunk rájuk hivatkozni. A továbbiakban a nevesített funkcionális regisztereket nagybetűvel írt NEVÜK formájában vezetem be. Úgy nevezem meg őket, ahogyan a PIC használati utasítása javasolja (2. irodalomjegyzék). Jellemzően a következő lépés az I/O portok beállítása, tehát lépjünk be a Bank1-be, hogy beállítsuk a portokat: bsf STATUS,5. A bsf, valamint a bcf a leggyakrabban használt parancsok egyike, felírási formájuk bsf/bcf REGISZTER,bitszám. A bsf 1-esre állítja az adott regiszter adott számú bitjét a bcf pedig lenullázza azt, tehát amikor visszalépünk bank0-ba: bcf STATUS,5, de maradjunk csak egyelőre a bank1-ben, hiszen az I/O portok beállítása még hátravan.

Létezik egy w regiszter, amit ideiglenesen használhatunk adattárolásra, de egyszerre csak 1 Byte fér bele, úgyhogy, ha használjuk jobb észben tartani, hogy milyen adatot tartalmaz. Ha fontos adatot tartalmaz, jobb lementeni mielőtt újból felülírnánk a w regisztert. A többi regisztert csak bitenként lehet írni. A w regiszterre viszont egyszerre rá lehet írni 1 bájtot, és az értékét átadni a többi regiszternek. Tehát a w egyfajta hidat teremt a futó programkód és a többi regiszter között.

Szeretnénk azt elérni, hogy a B2 láb bemenet legyen, a többi láb meg kimenet, tehát a TRISB-be a következő értéket kell letárolnunk: 00000100, ami decimálisan megfelel 4-nek. Az MPLAB assembly programnyelvében ezt így jelöljük: d’4’. Ezt az értéket tároljuk el w-ben: movlw d'4', látható, hogy ez a movlw paranccsal lehetséges. Ezután mozgassuk át a w értékét TRISB-be: movwf TRISB, ezzel tanultunk is 2 új parancsot: movlw és movwf. Most, hogy készen vagyunk a beállítással, visszamehetünk a Bank0-ba: bcf STATUS,5.

Tegyük fel, hogy a B5 lábra van kötve egy LED, célunk legyen az, hogy a LED világítson. A PORTA (05h) és PORTB (06h) regiszterek értékeinek módosításával lehet a kimeneteket módosítani, mindkét regiszter a Bank0-ában található. 1 érték esetén van kimenet, 0 érték esetén meg nincs, valamint ezeken regisztereken keresztül lehet a bemeneteket vizsgálni, 1

érték esetén van bemenet, 0 érték esetén meg nincs. Innentől kezdve, hogy ezeket tudjuk roppant egyszerű a dolgunk: bsf PORTB,5

Most csináljunk egy olyan programot, ami a B0-ára kötött LED-et kikapcsolva tartja mindaddig, amíg a B2-re nem érkezik bemeneti jel, amint érkezett maradjon a LED bekapcsolva. 2 új gyakori parancsot kell most megtanulnunk, de először leírnám a programkódot, hogy a későbbiekben értelmezni tudjuk:

```
bcf PORTB,0          ;LED kikapcsolása (így lehet kommentelni MPLAB-ban)
ciklus
btfss PORTB,2
goto ciklus
bsf PORTB,0          ;LED bekapcsolása
```

Kezdeném a legegyszerűbb paranccsal, ami a goto, ez odaugrasztja a programot, ahol a goto után leírt szó szerepel. A goto parancs az eddig tanult parancsoktól eltérően 2 műveleti ciklust tartalmaz.

A btfss REGISZTER,bitszám parancs megvizsgálja, hogy az adott bit értéke egyenlő-e 1-gyel, ha igen, akkor átugorja a következő parancsot, ha nem akkor végrehajtódik a következő parancs. A btfsc REGISZTER,bitszám parancs nem volt a példában, de említésre méltó, hiszen éppen az ellenkezőjét csinálja, 0-ás érték esetén ugorja át a következő parancsot. A fenti példában, ha a bemenet 1-re változik, átugródik a goto parancs, bekapcsolódik a LED és fut tovább a program. A btfss és btfsc parancsok 1 műveleti ciklust tartalmaznak, ha nem ugorják át a következő lépést, és 2 műveleti ciklust, ha átugorják. Átugrás esetén a következő parancs helyett egy NOP parancs hajtódik végre, ami egy üres parancs, ami időnyerésre jó és 1 műveleti ciklusba kerül. Ilyen nop parancsot a programozó is berakhat, ha pontosan 200 ns-ot szeretne késlelteni.

Általánosan elmondható, hogy minden parancs, ami ugrást tartalmaz, 2 műveleti ciklusba kerül.

Most nézzünk gyakori 2 műveleti ciklusos parancsokat. A call meghív egy szubrutint, és a return a szubrutin végén visszaugrik oda, ahol a call abbahagyta. Előző programkódunkat szubrutinba rakva így nézne ki:

```
bcf PORTB,0
call bemenetfigyelo
bsf PORTB,0

bemenetfigyelo
ciklus
btfss PORTB,2
goto ciklus
return
```

Ugyebár mind a call, mind a return ugrást hajtott végre, tehát 2 műveleti ciklust emésztenek fel. Van egy regiszter ami éppen azt tárolja, hogy hányas sorban jár a program, de azt szerencsére automatikusan módosítja a PIC.

Ez után legyen az a feladat, hogy a B0-ás lábon lévő LED egy bizonyos időkésleltetéssel aludjon el:

```
bsf PORTB,0
ciklus2
decfsz COUNT1,1
goto ciklus2
decfsz COUNT2,1
goto ciklus2
decfsz COUNT3,1
goto ciklus2
bcf PORTB,0
```

A COUNT1, COUNT2 és COUNT3 szabadon írható regiszterek, amíg a PIC áram alatt van 0-255-ig bármilyen érték letárolható bennük. Itt az új művelet a decfsz regiszter,d, ami csökkenti 1-gyel a regiszter értékért az eredményt meg a d értékétől függően vagy a w-be vagy a regiszterbe tárolja el. Ha d=0, akkor a w-be, ha d=1, mint a fenti példában is, akkor a regiszterben. Amint eléri a regiszter értéke a 0-át, átugorja a következő műveletet. Átugrás esetén 2 műveleti ciklusos, átugrás nélkül 1 műveleti ciklusos művelet. A fenti példában körülbelül 256*256*256 műveleti ciklus telik el, mire lekapcsol a LED, feltéve, ha mindegyik COUNT 0-ás értékről indul, mivel csökkenti eggyel, lesz belőle 255 és újra elmenti a COUNT-ba.

Ennek ellentétes testvérművelete az incfsz regiszter,d, ami 1-gyel növeli a regisztert, de minden másban ugyanúgy viselkedik, mint a decfsz.

Az incfsz-hez és decfsz-hez hasonlóan működnek az incf és a decf műveletek, csak itt simán csökkentik vagy növelik a regisztert és szintén d-től függően vagy a w-be vagy a regiszterbe menti az eredményt, de 0 esetén ugrás nincs benne.

Még gyakran használtam a clrf parancsot, ami az utána írt regisztert lenullázza pl.: clrf TRISA az A port összes lábát kimenetre állítja át.

A PIC tartalmaz egy watchdog timert is, ami méri az eltelt időt és, ha eléri a beállított határt, akkor újraindítja a PIC-et, ez megvéd attól, hogy a program bennragadjon egy esetleges végtelen ciklusban. A programkódomban ezt sokszor lenulláztam a clrwdt paranccsal, hogy újrainduljon a számlázás. Gyakorlatilag minden helyre, ahol tudtam, hogy sok időt tölt el a program, beraktam egy clrwdt parancsot. Elvileg ki lehet a watchdogot kapcsolni valahogyan, csak arra nem jöttem rá hogyan.

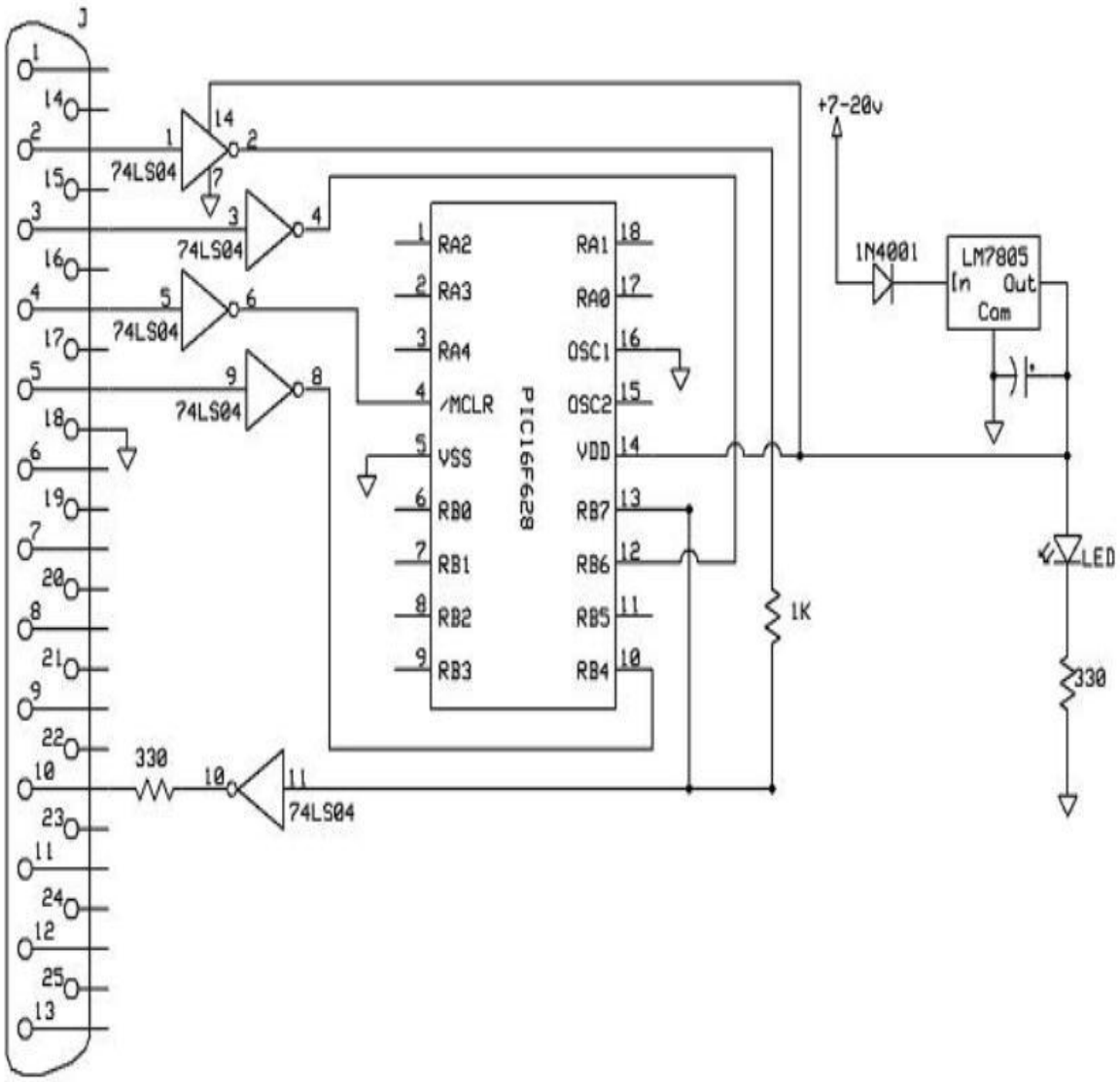
A STATUS regiszter 0. bitje a carry bit (C), ami a túlcsordulásokat tartalmazza kivonás, összeadás vagy biteltolás esetén. Az rrf parancs jobbra tolja a biteket, az rlf meg balra. Jobbra tolásnál a 7. bit megkapja az előző C értéket, a C értéke meg a korábbi 0. bit lesz. Balra tolásnál meg a 0. bit megkapja a C korábbi értékét, a C meg megkapja a 7. bit bit értékét.

Ezek a leggyakrabban használt parancsok, így ezek ismeretében a legtöbb ismert feladat elvégezhető. Lényegében ezzel véget is ért a PIC alapvető bemutatása, a következőekben az általam végrehajtott feladatokról írok.

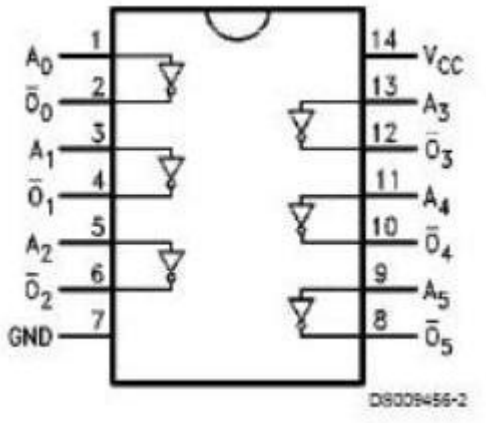
Programkód feltöltése a PIC-re

Mielőtt a programozási munkákba bele tudtam volna fogni, valamiféleképpen találnom kellett egy megoldást arra, hogy a HEX file-t a számítógépről rátöltsem a PIC 3,5 kByte-os FLASH memóriájára. Erre az Massachusetts Institute of Technology Egyetem honlapján találtam megoldást. Bevallom nem túl okosan kezdtem bele, mert megépítettem egy hardvert, anélkül, hogy előtte megbizonyosodtam volna arról, hogy van-e az adott hardverhez driver. Végül is szerencsém lett, és találtam egyet.

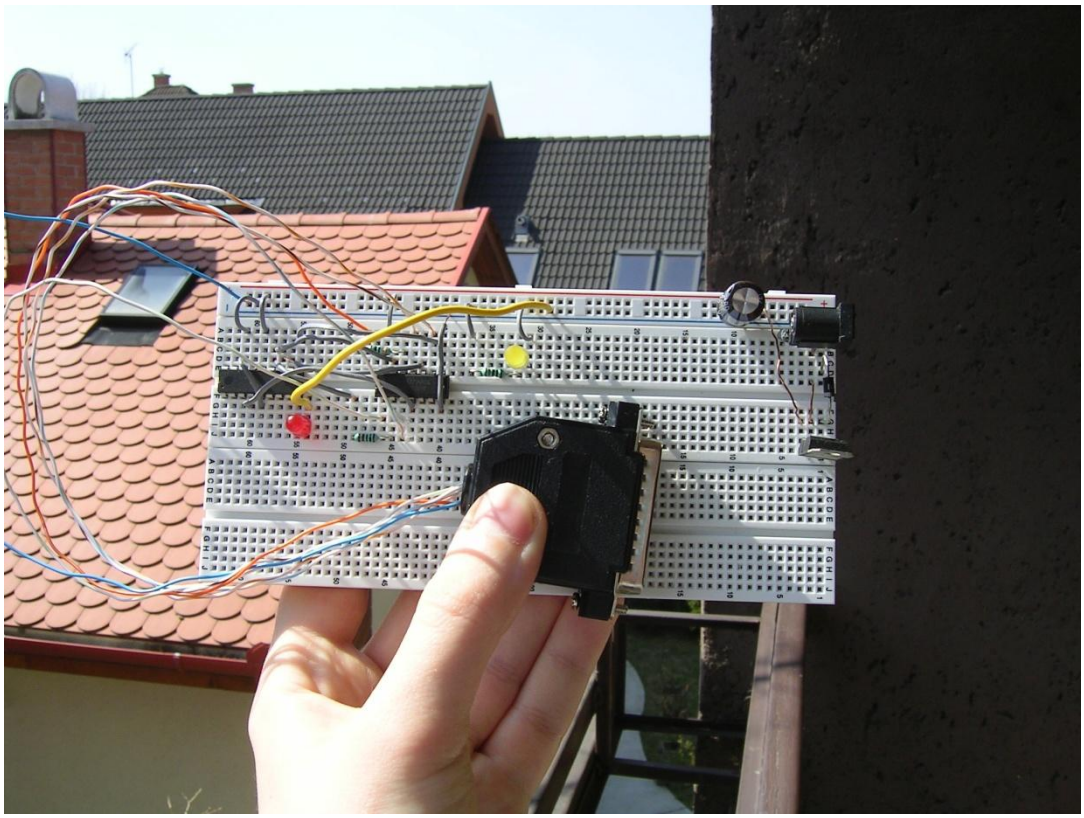
Egy olyan programozó áramkört valósítottam meg, ami a PC párhuzamos portján tölti fel a kódot a PIC-re. Ezt a portot használják a régi nyomtatók is. Itt látható a programozó:



Itt látható a programozóban található inverter:



A programozó panelt kezdetben nyomtatott áramköri lapon próbáltam megvalósítani, de a marásnál komplikációk adódtak. Nem marattam elég ideig, és amit kaptam eredményül az egy összefüggő rézlemez lett, ami minden pontot összekötve zárlatos. Miután a témavezetőm ezt meglátta, adott nekem egy műanyag áramköri próbapanelt, amin gyorsabban tudom a kapcsolásokat módosítani. Innentől kezdve nem volt szükségem marásra és forrasztásra. Itt látható a tesztpanelen megvalósított programozó áramkör:



Ez a panel 5V-osra alakítja át a feszültséget, ami a PC tápegységéből jön. A vezeték egyik végét a +12V-os sárga vezetékhez forrasztottam, a másikat meg a földhöz, így összesen 12V-ot kap a berendezés. Az LM7805-ös stabilizátor IC alakítja át az áramot 5V-ra. 3 lába van, az input láb megkapja a 12V-ot az egyenirányító diódán keresztül, így ha véletlenül valaki fordítva köti be, nem probléma, mivel nem engedi át az áramot. A common láb lesz a föld,

azaz a mínusz, az output láb pedig a +5V. Működési elvének lényege, hogy a common lábhoz igazítja az output lábat úgy, hogy 5V legyen a potenciál különbség.

Ehhez az áramkörhöz tartozik egy giveio nevű driver, amit a jimpic-1.9 programmal együtt telepítettem a PC-re. Ez a program Windows parancssorban fut, és lehetőséget biztosít a PIC azonosítására jimpic-1.9 –t paranccsal (Emlékszem milyen boldog voltam, amikor beírtam ezt a parancsot, és kiírta, hogy „Chip 16F628 is detected”.). Lehet vele a PIC memóriáját törölni a jimpic-1.9 –e paranccsal, és HEX kódot rátölteni a jimpic-1.9 –b *.hex paranccsal, ahol a * helyére kerül a fájl neve. (6. irodalomjegyzék)

Tehát az MPLAB-bal létrehozott HEX fájlt a jimpic-1.9 a megfelelő driver segítségével ráküldi a párhuzamos portra, ahonnan megkapja a programozó áramkör, és ebben rákerül a PIC-re. A PIC-et ezután kiemelhetjük a programozói áramkörből és behelyezhetjük abba az áramkörbe, ahol használjuk a ráírt programot. Ennek a végső áramkörnek a kimenetei és bemenetei pontosan illeszkednek a PIC lábaihoz.

PIN kódos beléptető rendszer

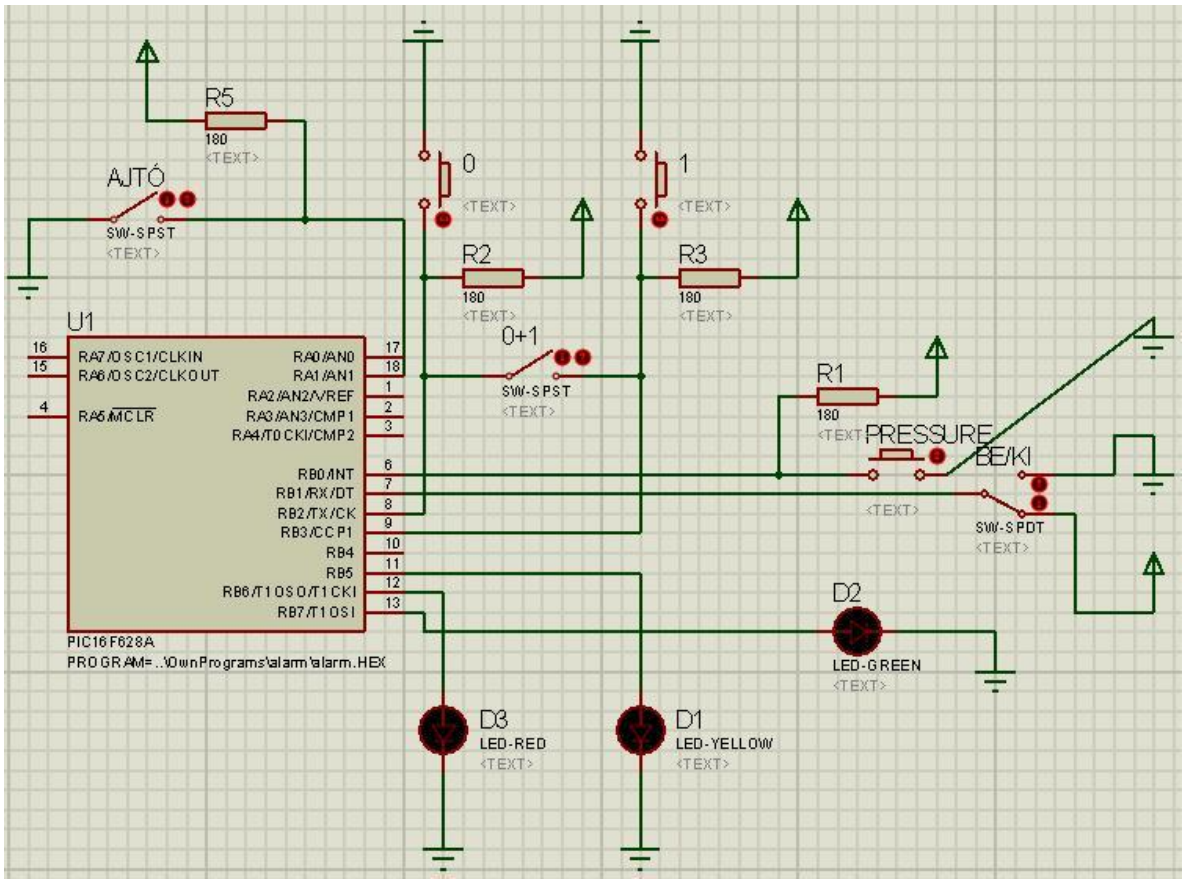
Ez a rendszer tartalmaz egy 2 gombos PIN kódos beléptetőt, amibe beprogramozható egy 4 jegyű kód, tehát összesen 16 féle kódot képes eltárolni. Van rákötve egy kapcsoló, ami a riasztót kapcsolja be/ki. 2 érzékelőt tartalmaz: egy ajtónyitás érzékelőt és egy nyomógomb érzékelőt. Van rajta 3 LED is, amelyek közül a zöld a PIN kód nyomógombjainak sikeres lenyomásáról tájékoztat, a sárga arról, hogy történt-e érzékelés, miközben be volt kapcsolva a riasztó. Ha történt, akkor 15 másodpercünk van beírni a PIN kódot vagy PIN kód független kiriasztást kezdeményezni, különben kigyullad a piros LED, ami a riasztást szimbolizálja.

A következő ki- és bemenetek vannak a következő lábakon:

- kimenetek:
    - sárga LED (B5)
    - piros LED (B6)
    - zöld LED (B7)
- bemenetek:
    - Reed mágneses relé (ajtónyitás érzékelő) (A1)
    - nyomógombos érzékelő (B0)
    - riasztó be/kikapcsoló (B1)
    - bal oldali gomb (B2)

- jobb oldali gomb (B3)

Itt látható a beléptető rendszer ISIS-ben készített tervrajza. A kész áramkörben nem található kapcsoló a 2 nyomógomb között, ide csak azért kellett raknom, hogy a 2 gomb egyszerre való lenyomását is leszimuláljam:



HASZNÁLATI UTASÍTÁS:

Áram alá helyezésnél felvillan egyszerre mindhárom LED és a zöld LED 3-at villog. Bekapcsolás után érdemes beprogramozni egy 4 jegyű PIN kódot, ha ezt nem tesszük meg az alapértelmezett PIN kód: bal, bal, bal, bal. Az áram megszűnésével a PIN kódot elfelejti és visszaállítódik az alapértelmezett PIN kód.

PIN kód beprogramozása: A riasztó élesítetlen állapotában lehetséges. Nyomjuk le egyszerre a bal és jobb nyomógombot, a zöld LED hosszan tartó világítása jelzi, hogy kezdhetjük a programozást. A bal és jobb gombok tetszőleges kombinációban való lenyomásával megadhatjuk neki a PIN kódot, ami kizárólag 4 jegyű lehet. Miközben írjuk a kódot, a zöld LED felvillanása tájékoztat a sikeres gombnyomásokról. Miután megadtuk a teljes kódot, a zöld LED 3-szor gyorsan felvillan, jelezve, hogy a kódot eltárolta.

A riasztó be/kikapcsolót megnyomva élesíthetjük a riasztót. (Ez programozó módban nem lehetséges, de, ha e közben nyomjuk be a kapcsolót, a programozó módból kilépve a riasztó egyből élesítődik.)

Élesített állapotban, ha a nyomógombos érzékelő gombját lenyomjuk, vagy, ha az ajtónyitás érzékelő mágneseit eltávolítjuk egymástól, bekapcsol a sárga LED, és, ha nem riasztjuk ki, akkor 15 másodpercre rá bekapcsol a piros LED. Miután a sárga LED bekapcsolt, el kell kezdeni a kiriasztási fázist. (Kiriasztás nem szükséges, ha a sárga LED nem kapcsol be. Ekkor a riasztó be/kikapcsolóval egyszerűen megszüntethető az élesítés.)

Kiriasztás 2 fajta módon lehetséges:

- felhasználó független kiriasztás, ha elfelejtenénk a kódot: Nyomjuk le egyszerre a bal és jobboldali gombokat, ha ez sikerült a zöld LED folyamatos világítása jelzi, majd nyomjuk meg riasztó be/kikapcsolót, ha ez sikerül a zöld LED 3-szori gyors villogása jelzi, és az összes LED kialszik.
- PIN kódos kiriasztás: Nyomjuk meg a riasztó be/kikapcsolóját. A zöld LED felvillanása jelzi, hogy kezdhetjük a PIN kód beírását. A gombok lenyomásának érzékelését a felvillanó zöld LED igazolja vissza. Ha a beírt 4 karakter hibás, 3-szor felvillan a zöld LED, és nem történek semmi, a sárga LED továbbra is bekapcsolva marad. Ha helyes kódot írunk be, akkor 6-szor villan fel a zöld LED, és az összes LED kialszik.

A PROGRAMKÓD:

A nyomógombok lenyomás hatására leföldelik a vezetéket, így a PIC adott lábán 0 jelenik meg. Egyébként 1 jelenik meg, hiszen áramot kap a láb. Ez eltér az emberi logika által diktált formától, itt a megnyomás 0, ha nem nyomjuk meg, akkor meg 1.

A program elején beállítottam a ki- és bemeneti lábakat, valamint kikapcsoltam a komparátort. A komparátor kikapcsolása úgy lehetséges, hogy a CMCON regiszternek decimális 7-es értéket adunk. A komparátor az A port lábain hasonlítja össze a bemeneti áramokat, és átállítódó bitek jelzik, hogy melyik áram a nagyobb. Mivel most nincs szükségünk áram összehasonlításra, ráadásul az A1-re van kötve az ajtónyitás érzékelő, szükséges volt a komparátor kikapcsolása:

```
bcf STATUS,5                ;BANK0
movlw d'7'
movwf CMCON                 ;komparátor kikapcsolása
bsf STATUS,5                ;BANK1
movlw d'15'
movwf TRISB
movlw d'2'
movwf TRISA
bcf STATUS,5                ;BANK0
```

A fenti kódrész az A port A1-es lábát bemenetre állítja, a B port B0, B1, B2, B3 lábait is bemenetre állítja, a többi láb kimenet lesz.

A következő rész a LED-eket felvillantja egyszer, utána jön a reszet rész, ahova visszaugrik a program a programozás befejeztével, a PIN kódos kiriasztás után vagy a felhasználó független kiriasztás után. Ez tartalmazza a zöld LED 3-szori gyors felvillanását.

Programozás előtt ez a ciklus fut le:

```
isiton?
btfss PORTB,1
goto sensor                 ;riasztó élesítése esetén megy ebbe bele
btfss PORTB,2
goto checkBbutton           ;programozás módba rakás
clrwdt
goto isiton?
```

Tehát látszik, hogy 2 dolog lehetséges ekkor: vagy a riasztót élesíthetjük vagy beprogramozhatunk egy PIN kódot. Programozáshoz ugyebár egyszerre kell lenyomni mindkét gombot, így, ha a bal oldali le van nyomva, rögtön meg kell nézni, hogy a jobb oldali is le van-e nyomva:

```
checkBbutton
btfss PORTB,3
goto progLED
goto isiton?
```

Kezdjük akkor a PIN kód beprogramozását, ami az általam erre a célra lefoglalt PIN regiszterbe kerül letárolásra. A regiszter páros bitjei tartalmazzák a kódszámjegyeket, a páratlan bitjei meg azt, hogy megtörtént-e az előtte lévő páros bit letárolása. Ha 1-es értéket kap a páratlan bit, akkor megtörtént az előtte lévő páros biten az eltárolás, ha 0-s értéket kap, akkor még nem történt meg. Fontos eltárolni, hogy megtörtént-e az adott biten a kódszámjegy eltárolása. Így például tudjuk, hogy a 3. kódszámjegy eltárolódott és várhatjuk a 4. kódszámjegyet a felhasználótól. A kódban a 0 megfelel a bal gombnak, az 1 pedig a jobb gombnak. Így a PIN regiszter pl.: bal, jobb, jobb, jobb kódot letárolva így néz ki: 11111110.

Miután megtörtént a letárolás, a páratlan helyeken lévő 1-es visszaigazoló biteket töröljük, helyet adva egy következő programozásnak. A végén így fog kinézni a PIN regiszter: 01010100. Kiszűrve a visszaigazoló biteket, megmaradnak a páros bitek 1110, jobbról balra olvasva bal, jobb, jobb, jobb.

A zöld LED-et villogtató részt (…) kihagyva így néz ki a PIN programozás főciklusa:

```
progLED
…
clrwdt
btfss PIN,1
goto prog1
btfss PIN,3
goto prog2
btfss PIN,5
goto prog3
btfss PIN,7
goto prog4
goto reszet
```

Tehát, ha az összes páratlan bit 1-es, visszaugrik a fentebb említett reszet részre. Így néz ki például a 2. PIN kódszámjegyet beprogramozó rész:

```
prog2
btfss PORTB,2          ;bal gomb
bsf PIN,3
btfss PORTB,3          ;jobb gomb
bsf PIN,3
btfss PORTB,2          ;bal gomb
bcf PIN,2
btfss PORTB,3          ;jobb gomb
bsf PIN,2
btfsc PIN,3
goto progLED
clrwdt
goto prog2
```

Látható, hogy mindegy melyik gombot nyomjuk le, a 3-as bit 1-es lesz. A bal gombot lenyomva a 2-es bit 0, a jobb gombot lenyomva pedig a 2-es bit 1 lesz.

Most nézzük meg mi lesz, ha az előző oldalon említett „isiton?” ciklusból nem a programozást választjuk, hanem a riasztó élesítő kapcsoló lenyomását. Ekkor bekerülünk a „sensor” nevű ciklusba, ami figyeli, hogy kapott-e bármelyik érzékelő jelet. Ebből a ciklusból a kapcsoló újbóli lenyomásával bármikor kiléphetünk, de, ha már világít a sárga LED, és belépett az alarm ciklusba, visszalépés csak kiriasztással lehetséges.

```
sensor
clrwdt
btfss PORTB,0                ;nyomásérzékelő
goto Alarm
btfsc PORTA,1                ;ajtózár érzékelő
goto Alarm
goto isiton?
```

Meglepő, hogy nem a „sensor”-ban marad, hanem visszalép az „isiton?”-ba, de nyugalom, ott egyből megnézi, hogy le van-e nyomva a kapcsoló és, ha le van, jön is vissza, még programozni se lehet ezen állapotában. Lényegében a „sensor” fut ciklikusan. Na, de mi van, ha az egyik érzékelő érzékel valamit? Jöhet egy másik fontos ciklus, az alarm:

```
Alarm
clrwdt
clrf DISALARM
bcf  PORTB,7                 ;zöld LED kikapcs
bsf  PORTB,5                 ;sárga LED bekapcs
call inccounter              ;a 15 másodpercet számolja
btfsc PORTB,1
goto kiriasztLED             ;PIN kódos kiriasztás
btfss PORTB,2
goto checkAbutton            ;felhasználó független kiriasztás
goto Alarm
```

Itt megjelenik egy eddig még ismeretlen DISALARM regiszter, ami a kiriasztáshoz beütött kódot tartalmazza. Ugyanúgy épül fel, mint a PIN regiszter. Minden sikertelen próbálkozás után le kell 0-áznunk, hogy a páratlan bitjei is 0-ák legyenek. Továbbá az „alarm” ciklus elágazik vagy a PIN kódos vagy a felhasználó független kiriasztás irányába, továbbá beindul a 15 másodperces számláló, a call inccounter-rel gyakran fogunk mostantól találkozni, hiszen bármit is csinálunk, számolni kell. Lehet, hogy éppen akkor telik le a 15 másodperc, és gyullad be a piros LED, amikor a kiprogramozás kellős közepén vagyunk, így néz ki az említett szubrutin:

```
inccounter
btfss COUNTER,7
incf COUNTER
btfsc COUNTER,7
incf  COUNTER2
call shortdelay              ;időt vár
btfsc COUNTER2,7
bsf  PORTB,6                 ;piros LED bekapcs
clrwdt
return
```

Ha a COUNTER2 utolsó bitje is 1 lett, akkor kigyullad a piros LED. Visszatérve az „alarm” ciklusra, nézzük meg a felhasználó független kiriasztást. Miután egyszerre lenyomtuk a 2 gombot belépünk ide:

```
checkkapcsolo
clrwdt
bsf PORTB,7                    ;zöld LED bekapcs
btfsc PORTB,1
goto reszet
call inccounter
goto checkkapcsolo
```

Látható, hogy innentől már nincs visszaút, a kiriasztást be kell fejeznünk a riasztó be/ki kapcsoló kikapcsolásával. A bonyolultabb rész a PIN kódos kiriasztás, ahol a PIN és a DISALARM regiszterek összehasonlításra kerülnek, miután a DISALARM regiszter feltöltődött, azaz beírtuk a kódot. A DISALARM regiszter feltöltésének a programkódja ugyanúgy néz ki, mint a PIN regiszter feltöltésének a programkódja, hiszen a művelet is ugyanaz. Beírjuk a kódot, csak annyi különbséggel, hogy máshol tárolódik. A bonyodalom akkor keletkezett, amikor egyenlőség jel hiányában akartam 2 regisztert összehasonlítani, hogy vajon helyes-e a beírt PIN kód. Az assemblyben ugyebár nincs egyenlőségjel. Ezt a problémát btfsc és a btfss bitvizsgáló műveletekkel oldottam meg. Például, ha a 0. bit 1 a PIN regiszterben, akkor nézzük meg, hogy a DISALARM-ban is 1-e. Ha igen, akkor mehetünk tovább a 2., 4. és 6. bitre. Ha mindegyik megegyezik, mehetünk vissza a „reszet”-hez, ha az egyik különbözik, akkor meg mehetünk vissza az „alarm” ciklusba.

```
checkPIN
… (zöld LED villogtatás)
btfss PIN,0
goto egyenlo0?1
goto egyenlo1?1
check2
btfss PIN,2
goto egyenlo0?2
goto egyenlo1?2
check3
btfss PIN,4
goto egyenlo0?3
goto egyenlo1?3
check4
btfss PIN,6
goto egyenlo0?4
goto egyenlo1?4
```

Most nézzük meg azt az esetet, amikor a PIN 0. bitje 1, ilyenkor meg kell nézni, hogy a DISALARM 0. bitje 1-e, ehhez viszont be kell lépni az „egyenlo1?1”-be.

```
egyenlo1?1
btfss DISALARM,0
goto Alarm
goto check2
```

Ha 1 itt is, akkor folytatja és a check2-re ugrik, ha nem, akkor visszamegy az „alarm”-ba. Ha a 6. bit, azaz 4. kódszámjegy is megegyezik, akkor meg a „reszet”-re ugrik, és a program visszalép az elejére. Most nézzük meg azt az esetet, amikor a PIN 6. bitje 0:

```
egyenlo0?4
btfsc DISALARM,6
goto Alarm
goto reszet
```

Ha a DISALARM 6. bitje is 0, akkor belép a reszetbe, és újraindul a programkód, magyarul megtörtént a PIN kódos kiriasztás.

Ezzel befejeztem a PIN kódos beléptető rendszerem bemutatását, a működéséről videó megtekinthető itt: http://youtu.be/zQSO3uPLSYY

A DS18B20 digitális hőmérő

Érdemes erről a hőmérőről külön szót ejteni, hiszen a hőszabályzó áramkörömnek a része. Ez küldi a PIC felé az aktuális hőmérséklet adatokat. Ez látszólag egyszerű, de, ha közelebbről megnézzük, ez is egy apró számítógép, egy kicsit kevesebb funkcióval ellátva, mint a PIC, de meg kell értenünk a működését ahhoz, hogy a következő fejezetben ismertetett hőmérő programom működését is felfogjuk. A működése a 3. irodalomjegyzékben is le van írva.

Itt látható egy DS18B20-as hőmérő:

Fél Celsius fok pontossággal képes meghatározni a hőmérsékletet -10 és +85 fok között. Egyébként -55 és +125 fok között működőképes. Kódolástól függően 1 hőmérsékletmérési eredmény tárolásához 9-12 bit kapacitásra van szükség. Mindegyik terméknek van egy 64 bites ROM azonosítója, ami lehetővé teszi, hogy egyszerre több termék legyen egy közös

vezetéken, mégis megkülönböztessük őket. A 64 bites azonosító első 8 bitje a CRC kód, ami a maradék 56 bitből van legenerálva, és leellenőrizhető vele az adatátküldés pontossága. Ezután következik az egyedi 48 bites azonosító, végül a 8 bites termékcsalád kód, ami 28h (hexadecimálisban).

3 és 5,5V között működik, tehát tökéletesen megfelelő neki az 5V-os áramköröm. 2 fajta módon lehet árammal ellátni vagy az áramfogadó lábán keresztül vagy az adatátviteli lábán keresztül. Nálam a VDD, azaz áramfogadó lábán keresztül kapja a feszültséget. 3 lába van, egy VDD egy GND (föld) és középen a DQ, amin keresztül az adat áramlik. 750 ms alatt konvertálja át digitálissá a hőmérsékletet. A DQ láb úgy közöl adatot, hogy leföldeli az áramot, tehát, ha a chip nem csinál semmit, akkor 1 a jel alapértelmezetten, ha leföldeli, akkor meg 0-ára vált.

Nézzük, hogyan épül fel a hőmérő memóriája. 9 bájt fér bele, ebből a 0. és az 1. bájt tartalmazzák 9-12 biten a hőmérsékletet. A 2. bájt a felső riasztó hőmérsékletet (TH), e hőmérsékleti érték felett aktiválódik a hőmérsékletérzékelő riasztója. A 3. bájt az alsó riasztó hőmérséklet (TL), ha ez alá esik a hőmérséklet, szintén aktiválódik a hőmérsékletérzékelő riasztója. A 4. bájt a konfigurációs regiszter. A konfigurációs regiszterben lehet beállítani, milyen pontosan digitalizálja a hőmérsékletet 9-12 bit pontosságig, ahol a bitkülönbség a tizedes törtekben jelentkezik. A 2., 3. és 4. bájt programozhatóak és az EEPROM-ban tárolódnak, tehát az áramellátás megszűnése esetén se vesznek el. 5-7. bájtig a hőmérsékletérzékelő saját célokra használja a regisztereket, e regiszterek nem írhatóak. A 8. bájt pedig a CRC bájt, ami az előző 8 bájt alapján generált kód, ez arra jó, hogy, ha leolvassuk 0-7-ig a bájtokat, ellenőrizni tudjuk a 8. bájt segítségével, hogy adatveszteség nélkül ment-e az adatátvitel.

Nézzük akkor, hogyan tárolódik le a hőmérséklet. 2-es számrendszerbeli alakban tárolódik, ahol 0. bit a 2^{-4}-t és egyre növekedve a 4. bit a 2^{0}-t, az 5. bit a 2-őt, a 6. bit a 4-et, egészen a 10. bitig, ami a 64-et jelképezi. A 11-15. bit értéke megegyezik, 0, ha pozitív számról van szó, és 1, ha negatívról. Attól függően, hogy hány bites hőmérséklet pontosságra van állítva a hőmérő, az összes bit ki van használva 12 bites pontosságnál, 11 bitesnél a 0. bit nincs kihasználva, 10 bitesnél a 0. és az 1. bit és végül 9 bitesnél a 0., az 1. és a 2. bit. Pl.: +10,5 így van ábrázolva: 0000 0000 1010 1000 (8+2+0,5).

Mivel a hőmérő is egy mini számítógép, parancsokat kell neki adni, hogy megtudjuk tőle mennyi a hőmérséklet. Rengeteg parancs létezik én mégis azokra fókuszálok, amelyeket a programomban használtam.

3 lépésből áll a DS18B20-nak való utasításadás:

1. Inicializálás
2. ROM parancs
3. Funkció parancs

Az inicializálás abból áll, hogy a vezérlő, én esetemben a PIC, kiküld egy reszet impulzust, ami egy minimum 480 µs-ig tartó 0 jel, és válaszul az eszköztől kap 15-60µs-ra rá egy jelenléti impulzust, ami egy 60-240µs-ig tartó 0 jel lesz. Az eszköz ezzel jelzi, hogy rajta van a csatornán és várja a parancsokat.

A ROM parancsok arra szolgálnak, hogy az eszközt beazonosítsuk a 64 bites ROM azonosítójukkal, ez kihagyható minden esetben, ha csak 1 eszköz van a vezetéken, mint például az én esetemben. Egyszerűen ROM parancsként kiküldöm a SKIP ROM parancsot, ami azt jelenti, hogy a következő funkció parancs a vezetéken lévő összes eszköznek szól. A SKIP ROM parancs azonosítója CCh.

Ezután küldhetjük át a hőmérőnek a funkció parancsokat, ezek közül 2-őt használtam. A CONVERT T [44h] parancsot, ahol 44h a parancs 1 bájtos azonosítója. Ez a parancs arra szolgál, hogy az aktuálisan mért hőmérsékletet digitalizálja, és tárolja el a hőmérséklet memóriában, ami a 0. és az 1. bájt. A másik parancs, amit használtam az a READ SCRATCHPAD [BEh] parancs. Ez kéri, hogy küldje át a hőmérő a PIC-nek a memóriában letárolt adatokat. Ezt használtam a hőmérséklet lekérdezéséhez. Ha nem akarjuk az összes 9 bájtot fogadni, hanem például elég az első 2 bájt, ami a hőmérséklet, akkor küldhetünk neki egy reszet impulzust. Reszet impulzus bármikor küldhető, ha újból akarunk neki parancsolni.

Most nézzük, hogyan tudunk írni rá adatokat. Lényegében, amikor parancsot adunk ki, akkor is írunk rá adatokat, így írásra mindenképpen szükségünk van a program elkészítéséhez. Írásra időréseket használunk, 2 fajta időrés létezik, a 0 író és az 1 író, attól függ, hogy 0-ást vagy 1-est akarunk átküldeni. Egy időrés 60 és 120µs tartományban van, és ebből a DS18B20 15-60µs-ig vesz mintát. Ha ekkor a jel alacsony, akkor 0-ást kap, ha magas, akkor meg 1-est. Mintavétel előtt mindig 0-ára kell állítani a jelet, hogy a hőmérő tudja, mikor kezdődik az időrés. Tehát, ha minimum 60µs-ig 0V-on tartjuk a vezetéket, akkor 0-ást küldünk át, ha csak

nagyon rövid ideig (kábé 1µs-ig) 0V-on tartjuk a vezetéket, utána 60µs-ig 5V-on, akkor 1-est küldünk át.

Bit fogadásnál, amikor például a hőmérséklet értékét küldi át a DS18B20 a PIC-nek, a fogadó egységnek kell egy minimum 1µs-ig 0V-ra állítani a vezetéket, ezzel jelezve a hőmérőnek, hogy küldheti a bitet. A hőmérő 0-ás bit esetén 60µs-ig 0V-ra lehúzza a vezetéket, 1-es bit esetén pedig nem csinál semmit, hagyja, hogy az áramforrás küldje az 5V-ot a vezetékbe. A PIC-nek 15µs-ig kell mintavételeznie a vezetékből miután elküldte a 0V-os jelet. A mintavételezés eredménye lesz a kapott bit.

Ennyit elég tudni a DS18B20-as működéséről és parancsairól annak érdekében, hogy megértsük a következő fejezetben lévő programkódot.

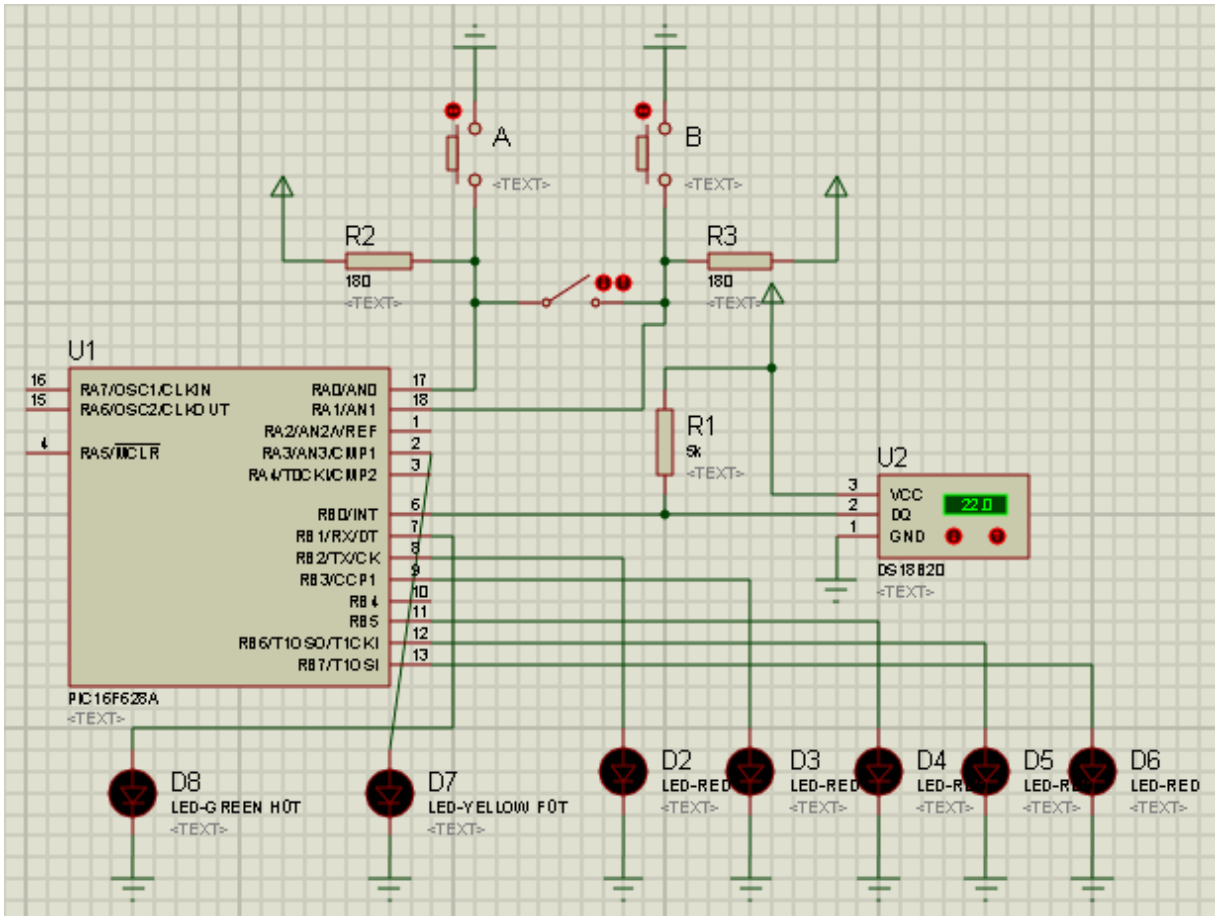
Hőmérsékletmérő és szabályozó egység

Ez az áramkör a levegő hőmérsékletét jelzi ki bináris formátumban 5 piros LED segítségével 0-31 Celsius fokig. Be lehet rajta állítani a kívánt hőmérsékletet, és, ha az alatt van a hőmérséklet, akkor bekapcsol a sárga LED, és jelzi, hogy fűt, ellenkező esetben meg bekapcsol a zöld LED, ami jelzi, hogy hűt, ha nem világít egyik se, akkor pontosan megegyezik a kívánt és a mért hőmérséklet. Természetesen nem fűt és nem hűt semmit, a LED-ek csak szimbolizálják a fűtést és a hűtést.

Ki- és bemenetek, valamint a hozzájuk tartozó lábak:

- Kimenetek:
  - 5 db piros LED (B2, B3, B5, B6, B7)
  - zöld LED (B1)
  - sárga LED (A3)
- Bemenetek:
  - bal oldali nyomógomb (A0)
  - jobb oldali nyomógomb (A1)
- Ki- és bemenet:
  - DS18B20-as digitális hőmérő (B0)

Itt látható a hőmérséklet szabályozóm ISIS-ben készített tervrajza. Ezen szimuláltam le a működését mielőtt megépítettem volna:



HASZNÁLATI UTASÍTÁS:

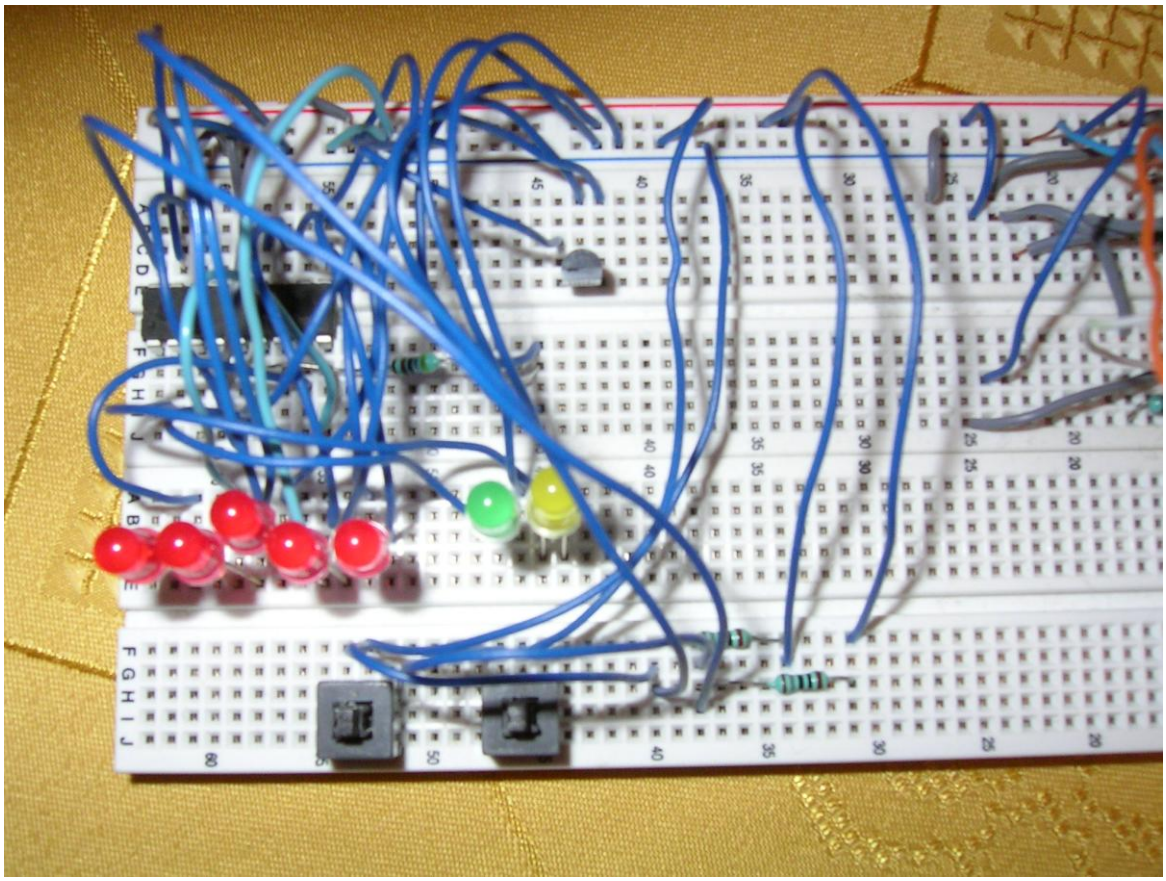
Áram alá helyezéskor az összes LED felvillan jelezve a készülék működését, és kijelzi az 5 db piros LED-del binárisan a hőmérsékletet. Megszámozzuk a LED-eket balról-jobbra 1-5-ig. Az 1-es a 16-os, a 2-es a 8-as, a 3-as a 4-es, a 4-es a 2-es, és az 5-ös pedig az 1-es számot jelképezi. A világító LED-ek jelképezik a számokat és ezeket a számokat kell összeadni, hogy megkapjuk a kijelzett hőmérsékletet. Tehát, ha pl.: 21 fok van, akkor világít az 1-es, a 3-as és az 5-ös LED (16+4+1).

Az eszköz folyamatosan mutatja az aktuális hőmérsékletet, ha nem világít a sárga és a zöld LED egyszerre. Mindkét LED egyszerre világít hőmérsékletszabályozó módban. Hőmérsékletszabályozó módba a bal és jobb gomb egyszerre történő lenyomásával léphetünk be, valamint egyszerre történő lenyomásával léphetünk ki. Ebben a módban a piros LED-ek az aktuálisan beállított hőmérsékletet mutatják. Ezt a hőmérsékletet a jobb gomb lenyomásával növelhetjük egyesével, a bal gomb lenyomásával meg csökkenthetjük

egyesével. A termosztát alapból 21 fokra van állítva, így minden áram alá helyezésnél 21 fok lesz a kívánt hőmérséklet.

Miután visszaléptünk hőmérő üzemmódba a sárga LED jelzi, hogy fűteni kell, a zöld LED meg azt, hogy hűteni kell, ha egyik sem világít, akkor megegyezik a beállítani kívánt hőmérséklet a mért hőmérséklettel, ha meg mindkettő világít, akkor hőmérsékletszabályozó módban vagyunk.

Itt látható az áramköri tesztpanelen megépített hőmérsékletszabályozó egység:



A PROGRAMKÓD:

Az kód a komparátor kikapcsolásával, valamint ki- és bemenetek beállításával kezdődik:

```
bcf STATUS,5            ;BANK0
movlw d'7'
movwf CMCON
bsf STATUS,5            ;BANK1
movlw d'1'
movwf TRISB
movlw d'3'
movwf TRISA
bcf STATUS,5            ;BANK0
```

A B0-ást bemenetre állítjuk, erre lesz kötve a hőmérő, így ez ki- és bemenetként is funkcionál majd. Az A0-át és A1-et bemenetre állítjuk. A0 a bal gomb, A1 pedig a jobb gomb, a többi láb marad kimeneti.

A termosztátot alapbeállításként 21 fokra állítjuk. A TERM változóban tárolódik el a kívánt hőmérséklet:

```
movlw 15h
movwf TERM
```

Az aktuális hőmérséklet a későbbiekben a TEMP változóban kerül eltárolásra.

Bekapcsolásnál az összes LED felvillan, majd kialszik, és utána elindul a GET_TEMP ciklus, ami addig fut, amíg át nem lépünk hőmérsékletszabályozó üzemmódba. Ebben a ciklusban olvassuk le a hőmérőről a hőmérsékletet, és elmentjük a TEMP változóba, valamint összehasonlítjuk a TERM és TEMP változókat, hogy fűteni vagy hűteni kell, továbbá figyeljük a gombnyomást.

Így indul a ciklus:

```
GET_TEMP
    CALL   DS18B20_INITIALIZATION        ; Reset impulzus
    MOVLW  H'CC'                         ; Skiprom parancs
    CALL   DS18B20_WRITE_BYTE
    MOVLW  H'44'                         ; Convert T parancs
    CALL   DS18B20_WRITE_BYTE
```

Tehát leadjuk a DS18B20-asnak a Convert T parancsot, hogy digitalizálja a hőmérsékletet, és mentse el a 0. és az 1. regiszterébe. A Skiprom parancs meg ugyebár azt jelenti, hogy a következő parancs a vezetéken lévő összes eszköznek szól.

Hogy megfelelő legyen a kommunikáció be kellett állítanom mennyi legyen 1µs. Ezt a DELAYuS szubrutin hajtja végre. W értékétől függ hány µs-ot késleltet. W értékének 5-szörösét késlelteti, tehát pl.:

```
MOVLW (D'480')/5
CALL DELAYuS
```

Ez például 480µs-ot késleltet.

Ugyebár mondtam, hogy 3 részből áll a hőmérővel való kommunikáció, a GET_TEMP ciklus elején jól látszódik ez a 3 rész elkülönítve: inicializálás, ROM parancs, funkció parancs. Nézzük meg az inicializálást:

```
DS18B20_INITIALIZATION
clrwdt
CALL    setout                  ;kimenetre állítja a B0-át és 0-s jelet ad ki
MOVLW   (D'480')/5              ; 0-át küld 480µs-on át.
CALL    DELAYuS
CALL    setin                   ;bemenetre állítja a B0-át
MOVLW   (D'60')/5
CALL    DELAYuS
BTFSC   PORTB, 0                ; Van-e válasz 0?
goto    DS18B20_INITIALIZATION          ; Ha nincs, újra indul az 1. lépés
MOVLW   (D'420')/5              ; 420µs várakozás, hogy a bemenet újra 1 legyen.
CALL    DELAYuS
return
```

Ebben látszik az, amit a DS18B20 bemutatásánál elmondtam, kiküldünk egy 480µs-os 0-át majd kapunk rögtön utána egy 60-240µs-os 0-át, Ez a ciklus addig küldi a 0-át, amíg nem kap válasz 0-át rá. A setout nemcsak kimenetre állítja azt a lábat, amin a hőmérő van, hanem 0-át küld ki rajta:

```
setout
bsf STATUS,5                    ;BANK1
bcf TRISB,0
bcf STATUS,5                    ;BANK0
bcf PORTB,0
return
```

A setin meg egyszerűen bemenetre állítja:

```
setin
bsf STATUS,5                    ;BANK1
bsf TRISB,0
bcf STATUS,5                    ;BANK0
return
```

Most, hogy megtörtént az inicializálás, a következő kérdés, ami felvetődik, hogy hogyan is küldhetnénk el 1 bájtot (parancsot) a hőmérőnek. A következő szubrutin a w értékét küldi át a hőmérőnek:

```
DS18B20_WRITE_BYTE              ;W értékét ráírja a DS18B20-asra
      MOVWF   TMP0
      MOVLW   D'08'
      MOVWF   TMP1
      RRF     TMP0,1            ; jobbra shiftelés, a kimaradt érték belemegy a C-ébe
      CALL    DS18B20_WRITE_BIT
      DECFSZ  TMP1,1
      GOTO    $-D'3'            ;ez 3 sorral visszaugrik
      RETURN
```

A TMP0-ában eltárolódik az átküldendő adat. A TMP1-be pedig a decimális 8 tárolódik el, ez számolja, hogy mind a 8 bit átment-e. A DS18B20_WRITE_BIT szubrutin a C bitet (carry bitet) küldi át a hőmérsékletérzékelőnek. A bitküldés folyamata ciklikusan ismétlődik 8-szor, mindaddig, amíg a TMP1 el nem éri a 0-át. A TMP0 bitjei pedig jobbról balra küldődnek át, tekintve, hogy az RRF jobbra shiftel. Amikor biteket kapunk majd a későbbiekben a hőmérőtől azok is jobbról balra érkeznek, megszokhattuk, hogyha bitekről van szó, akkor jobbról balra szokás őket olvasni. Az RRF parancs végén az 1-es jelzi, hogy a biteltolás eredményét vissza kell rakni a TMP1-be.

Most nézzük hogyan küldünk át egy bitet:

```
DS18B20_WRITE_BIT       ;A Carry, azaz C értékét küldi ki a lábán
    CALL    setout
    BTFSS   STATUS, C
    GOTO    $+D'2'      ;2 sorral előre ugrik
    CALL    setin
    MOVLW   (D'60')/5   ;60µs várás
    CALL    DELAYuS
    CALL    setin
    RETURN
```

Ugyebár tanultuk, hogy le kell húzni 0V-ra a lábat mielőtt bármilyen bitet küldünk, ezzel jelzünk a hőmérőnek, ezt a „setout” szubrutin megteszi. Ha 0-ás bitet küldünk, akkor 60µs-ig így marad a jel, ha meg 1-est, akkor rögtön bemenetre állítjuk és a 4,7 kOhm-os ellenálláson keresztül visszaáll magától 5V-ra.

Ha már itt járunk nézzük meg milyen egy bit olvasás:

```
DS18B20_READ_BIT        ;A kapott értéket felveszi a C (carry)
    CALL    setout
    CALL    setin
    BSF     STATUS, C   ;1-re állítom alapból a carry flaget
    BTFSS   PORTB, 0
    BCF     STATUS, C
    MOVLW   (D'60')/5   ;60µs várás
    CALL    DELAYuS
    RETURN
```

„Setout”-tal kiküldünk egy 0-át jelezve, hogy készen állunk a bit fogadására. Rögtön utána bemenetre állítjuk és olvashatjuk is lefele a vezetéket, hiszen vagy visszaállt 5V-ra (1-est kaptunk) vagy a hőmérő lehúzta 0V-ra (0-ást kaptunk). A C (carry) értékében letárolódik a bit.

Most nézzük meg, hogyan hasznosítja a C-ben eltárolt bitet a program, és írja be a w regiszterbe:

```
DS18B20_READ_BYTE  ;Megkapja a DS18B20 által küldött biteket és W-ben tárolja
    MOVLW   H'08'
    MOVWF   TMP1
    CALL    DS18B20_READ_BIT
    RRF     TMP0, 1             ;jobbra shiftelés, a C-ből átkerül a bit a TMP0 7. bitjére
    DECFSZ  TMP1, 1
    GOTO    $-D'3'              ;3 lépéssel hátra ugrás (bitolvasás)
    MOVF    TMP0, W
    RETURN
```

A TMP1 értéke 8 lesz, ez számolja vissza, hogy megkaptuk-e mind a 8 bitet. A TMP0 pedig mindig a bal oldalába (7. bitjébe) megkapja a C bitet, és mivel jobbra shiftelés történik az egész sor mindig jobbra mozog, így a folyamat végén az elsőnek küldött bit a 0. bit értékén lesz, az utoljára kapott bit, pedig a 7. bit értéken. Mikorra a megvan 8 bit a TMP1 0 lesz, így a DECFSZ átugrik a GOTO-n. Végül a w megkapja hőmérő által küldött 8 bitet.

Innentől már tudjuk pontosan, hogyan történik a hőmérővel a kommunikáció, így visszatérve a GET_TEMP ciklusunkhoz megérthetjük azt, hogy hogyan történt a convert T parancs kiadása. A következő parancs, amit ki kell adnunk az a READ SCRATCHPAD parancs:

```
CALL    DS18B20_INITIALIZATION    ;Reset
MOVLW   H'CC'                     ;Skiprom parancs
CALL    DS18B20_WRITE_BYTE
MOVLW   H'BE'                     ;Read Scratchpad parancs
CALL    DS18B20_WRITE_BYTE
```

Miután ezt megkapta a hőmérő elkezdi küldeni a memóriaregisztereinek értékét, kezdve az első 2 regiszterrel, ami a hőmérsékletet tartalmazza:

```
CALL    DS18B20_READ_BYTE         ;A scrachpad 0. Byte-ja a W-be kerül
MOVWF   TEMP1
CALL    DS18B20_READ_BYTE         ;A scrachpad 1. Byte-ja a W-be kerül
MOVWF   TEMP2
call show_temp
```

Utána a show_temp szubrutin a piros LED-ek segítségével kijelzi az hőmérsékletet. Ez a TEMP1 4-es, 5-ös, 6-os, 7-es, valamint a TEMP2 0-s bitjét figyeli, magyarul az egész számokat 0-31-ig.

Utána a TEMP regiszterben tárolódik a hőmérséklet, hogy ne legyen 2 regiszterben szétszórva. A TEMP-ben az egész értékek tárolódnak. A TEMP1 4-es, 5-ös, 6-ös, 7-es bitjei lesznek a TEMP 0-s, 1-es, 2-es, 3-as bitjei. A TEMP2 0-s, 1-es, 2-es, 3-as bitjei lesznek a TEMP 4-es, 5-ös, 6-ös, 7-es bitjei.

A GET_TEMP ciklus végén következik a TEMP (mért érték) és a TERM (kívánt érték) összehasonlítása, magyarul fűteni vagy hűteni kell-e? Ha TEMP>TERM hűteni kell (zöld LED), ha TEMP<TERM, akkor fűteni kell (sárga LED). A PIC processzora nem ismeri a relációs jelet, csak a használható 35 parancsból gazdálkodhattam. A XOR műveletet használtam a TEMP és TERM regiszterek bitjei között. A XOR ugyebár kizáró vagy, tehát eredménye akkor 1, ha a 2 bemeneti érték különbözik. Ha a bemeneti értékek megegyeznek, akkor kizárják egymást, és az eredmény 0 lesz. Tehát 0 XOR 1, 1 XOR 0 1-et, 0 XOR 0, 1 XOR 1 0-át ad eredményül. 2 db 8 jegyű bináris szám közül az lesz a nagyobb, amelyiknek 1 lesz azon a helyi értéken az értéke, ahol elsőnek eltérnek egymástól. Pl.: 1010**1**100 és 1010**0**111 közül az első lesz a nagyobb, mert a 3. bitértéken térnek el egymástól elsőnek, és ott az első számnak 1, a második számnak 0 az értéke. Ebben a példában, ha számjegyenként XOR műveletet hajtunk végre a 2 szám között, ezt a számot kapjuk: 0000**1**011. Balról a legelső 1-es lesz a legnagyobb helyi értéknél való eltérés. Ennél a helyi értéknél megvizsgáljuk a bitet, és amelyiknek 1 lesz az lesz a nagyobb szám.

Ezen algoritmus alapján döntöttem el, hogy melyik a nagyobb. Kezdem a XOR művelettel:

```
movlw d'0'
addwf TEMP,0                ;TEMP eltárolása W-be
xorwf TERM,0
movwf TEMP3
goto checkbit7
goto GET_TEMP               ;ez visszaugrik a GET_TEMP ciklus elejére
```

Az elején a w-nek átadtam a 0 értéket, és hozzáadtam TEMP-et és ezt eltároltam w-be. Röviden a hőmérséklet értékét w-be tároltam. Majd XOR műveletet hajtottam végre w és TERM között majd az eredményt eltároltam w-be. Végezetül a w értékét megkapta a TEMP3, magyarul a XOR művelet eredményét megkapta TEMP3. Most vizsgálnunk kell a TEMP3 értékét a 7. bittől kezdve egyesével haladva a 0. bit felé, ahol elsőnek 1 lesz azon a ponton kell a TEMP és TERM bitjeit összehasonlítani.

7. bit vizsgálata:

```
checkbit7
BTFSC TEMP3,7
goto temportermbit7
goto checkbit6
```

Tegyük fel, hogy itt még 0 az érték, haladunk tovább, jön a 6. bit vizsgálata:

```
checkbit6
BTFSC TEMP3,6
goto temportermbit6
goto checkbit5
```

Tegyük fel, hogy itt 1 az érték tehát megtaláltuk a legelső eltérést a TEMP és TERM között, nézzük meg melyik lesz 1, azaz melyik lesz a nagyobb?

```
temportermbit6
btfsc TEMP,6
bsf PORTB,1                ;hűtés bekapcs
btfsc TEMP,6
bcf PORTA,3                ;fűtés kikapcs
btfss TEMP,6
bsf PORTA,3                ;fűtés bekapcs
btfss TEMP,6
bcf PORTB,1                ;hűtés kikapcs
goto GET_TEMP
```

Itt a TERM-mel már nem is foglalkozunk, elég a TEMP-pel, hiszen, ha TEMP 6. bitjén az érték 0, akkor kizárásos alapon a TERM 6. bitje 1 lesz. Ha TEMP értéke itt 1, akkor ő lesz a nagyobb, így hűtést bekapcsolhatjuk a fűtést meg kikapcsolhatjuk, ha a TEMP értéke itt 0 lesz, akkor ő lesz a kisebb, így a fűtést bekapcsolhatjuk, a hűtést meg kikapcsolhatjuk. A végén meg visszaugrik a GET_TEMP-be és kezdődik az egész folyamat elölről, tehát kezdetbe megkérjük a hőmérőt, hogy digitalizáljon, majd lekérjük tőle a hőmérsékletet, majd kijelezzük a hőmérsékletet, majd elmentjük a TEMP-be a hőmérsékletet, és a végén összevetjük a TEMP és TERM regisztereket, és az alapján döntünk, hogy fűtsünk vagy hűtsünk. A GET_TEMP-en belül, amíg várunk a hőmérőre, hogy a hőmérsékletet digitalizálja, addig figyeljük a gombnyomást is, hogy nem akar-e a felhasználó hőmérsékletszabályozó módba ugrani.

Ennek érdekében, hogy várás közben gombnyomást figyeljen a program belecsempésztem a gombnyomás érzékelést az időkésleltetésbe:

```
delaycheckA
clrf COUNT3
bsf  COUNT3,2
loop1    decfsz COUNT1,1
goto loop1
         btfss PORTA,0
         goto checkB           ;A gomb lenyomva, B le van-e nyomva?
         decfsz COUNT2,1
goto loop1
         decfsz COUNT3,1
goto loop1
```

```
return
```

Ha A gomb le van nyomva, nézzük meg a B gombot is, ugyanis egyszerre két gomb lenyomásával lehet hőmérsékletszabályozó módba kapcsolni:

```
checkB
btfss PORTA,1
goto wait2
goto GET_TEMP
```

Hogy rögtön a két gomb lenyomása után, ne kezdje el a hőszabályzót beállítani, egy kis késleltetést raktam be. (Ez megfelel szoftveres pergésmentesítésnek.) Ugyebár, ha nincs késleltetés, akkor tovább lép a program, és a két gomb lenyomását hőmérsékletcsökkentés vagy növelés parancsnak vélheti, amit nem akarunk:

```
wait2
call show_term
clrwdt
bsf PORTB,1                ;zöld LED bekapcs
bsf PORTA,3                ;sárga LED bekapcs
call delay
call delay
call delay
goto set_termosztat
```

Itt nem csak a késleltetés történik meg, és a zöld és sárga LED bekapcsolása, hanem a hőszabályzóval eddig beállított hőmérséklet kijelzése a „show_term” subrutinnal:

```
show_term
bsf PORTB,2
bsf PORTB,3
bsf PORTB,5
bsf PORTB,6
bsf PORTB,7
btfss TERM,0
bcf PORTB,7
btfss TERM,1
bcf PORTB,6
btfss TERM,2
bcf PORTB,5
btfss TERM,3
bcf PORTB,3
btfss TERM,4
bcf PORTB,2
return
```

Ez mind az 5 piros LED-et bekapcsolja, majd a 0-ás TERM biteknél kikapcsolja. Ilyenkor a 0-ás bitek kijelzésénél olyan rövid ideig van bekapcsolva a LED, hogy kikapcsoltnak tekinthető. Most nézzük azt a részt, ahol a hőmérsékletet lehet beállítani:

```
set_termosztat
        call show_term
        btfss PORTA,0          ;bal gomb benyomása
        call sub_term          ;csökkenti 1-gyel a hőmérsékletet
        call delay
        clrwdt
        btfss PORTA,1          ;jobb gomb benyomása
        goto checkA            ;le van-e a bal gomb is nyomva?
goto set_termosztat
```

A bal gomb benyomásával csökkentjük 1-gyel a hőmérsékletet:

```
sub_term
DECF TERM,1
call delay
call delay
goto set_termosztat
```

Ez a csökkentet értéket visszamenti a TERM-be, hiszen a DECF parancs 2. bemenete 1, 0-ás esetén a w-be mentette volna. Gombnyomás után ismét szükség van időkésleltetésre, hogy a lenyomást 1 lenyomásnak nézze, ne sok lenyomásnak. Ugyanis bármennyire rövid ideig tart a lenyomás, a PIC processzora gyorsan fut és értelmezne több 100 lenyomást, ha nem késleltetnénk.

Jobb gomb lenyomásával növelhetjük a hőmérsékletet vagy, ha mellé a bal gomb is le van nyomva, visszaléphetünk az eredeti GET_TEMP ciklusba. Ezért kell megnézni, hogy a bal gomb le van-e nyomva:

```
checkA
btfss PORTA,0
goto wait
INCF TERM,1
call delay
call delay
goto set_termosztat
```

Ha le van nyomva, továbblépünk a „wait”-be, ha nincsen, akkor meg növeljük 1-gyel a hőmérsékletet, és elmentjük a TERM-be. Utána az előbbi okok miatt ismét késleltetünk. A „wait” azért szükséges, hogy ne az legyen, hogy lenyomtuk egyszerre a két gombot kilépünk a hőszabályzó módból, és újból ott találjuk magunkat a hőszabályzó módban, hanem időt kell hagyni a gombok felengedésére.

```
wait
call show_temp
call delay
call delay
call delay
goto GET_TEMP
```

Mostantól újból kijelezhetjük a hőmérsékletet, ami nem más, mint az a hőmérsékleti érték, ami a hőszabályzó módba belépés előtt volt, de századmásodpercek töredéke alatt új mérést végez a hőmérő, és a friss hőmérséklet kerül a kijelzőre.

Dióhéjban ennyit érdemes tudni a DS18B20-as digitális hőmérővel összekötött PIC16F628-ról. Videofelvétel megtekinthető róla itt: http://youtu.be/MY5bxW4-PrA

RS232 kommunikáció USART segítségével

Az RS232 egy elterjedt kommunikációs forma, a távközlésben alkalmazott szabvány. Ezen szabvány szerint soros összeköttetés valósítható meg bármilyen 2 berendezés között. A PIC-et például ezzel a szabvánnyal össze lehetne kötni egy PC-vel, egy LCD képernyővel vagy 2 PIC-et is össze lehet kötni egymással, mint ahogyan ezt én is megvalósítottam, és a későbbiekben be is mutatom.

Az RS232-ben a +3V-tól +15V-ig felel meg a 0-ának, -3V-tól -15V-ig pedig az 1-nek. 8 bites vagy 9 bites csomagokat lehet küldeni vele szinkron vagy aszinkron. Ha szinkronban küldjük, az azt jelenti, hogy a küldő és fogadó összhangban vannak egymással, és nem kell jelezni minden csomag előtt, hogy indul a csomag. Aszinkronnál viszont jelezni kell. Minden aszinkron csomag egy START bittel kezdődik és egy STOP bittel zárul le. A START bit 0, a STOP bit pedig 1. Ha nincs csomagküldés a vezetéken, akkor -3V és -15V közötti feszültség van rajta, ez azért jó, hogy mindig van feszültség a vezetéken, hogy meg lehet állapítani, él-e még a vonal. Tehát például ez a Byte: 10010101 így küldődik el aszinkron módban: 0101010011. Az elejére került egy 0 és a végére egy 1-es, a bitek meg jobbról balra küldődnek, pontosan úgy, amikor a hőmérőnek küldtem adatokat. A STOP bit után azonnal következhet START bit is, ha sűrűn vannak küldve az adatcsomagok.

A küldő és fogadó egység bitsebességének meg kell egyeznie, hogy a jel jól menjen át, és ne legyen elcsúszás. Ennek mértékegysége a baud. Megmutatja, hány jel továbbítódik 1 másodperc alatt. Ha például 200 baud a jeltovábbítási sebesség, és 1 jel 5 bit-et tartalmaz, akkor 1000 bit/másodperc az adat sebessége. Az RS232 esetén 1 jel 1 bitet tartalmaz, így 200

baud az 200 bps-nek felel meg (A bps az bit/szekundum). Azért tartalmaz 1 jel 1 bitet, mert 2 fajta jelszint létezik, a 0-ás és 1-es.

Nézzük hogyan tud a PIC kommunikálni a USART segítségével. A USART rövidíti a Universal Synchronous/Asynchronous Receiver/Transmitter-t, ami a nevében foglalja, hogy univerzális szabvány, tud szinkron és aszinkron módban kommunikálni, és tud küldeni, valamint fogadni is. A PIC B1-es lába a fogadó (RX), a B0-ás lába pedig a küldő (TX). Az USART részletes leírása a 7. irodalomjegyzékben található.

Szinkron módban az órajel és adatforgalom külön vezetéket használ, mivel közös órajelet használ a küldő és fogadó, ezért összehangolni, szinkronizálni tudják a kommunikációt. Aszinkron módban a start bit jelzi a csomag elejét, így nincs szükség csak 1 vezetékre, ami összeköti a küldő és fogadó lábakat. Két PIC összekötése esetén az egyik PIC TX lábát összekötjük a másik PIC RX lábával. Az USART-ot leggyakrabban aszinkron módban használják, így most én csak ezzel a móddal fogok foglalkozni.

8 bites adatküldésnél a TXREG regiszter tartalmazza a küldendő információt, de mivel 1 regiszterbe csak 1 Byte fér, 9 bites adatküldésnél a 9. bitet a TXSTA regiszter TX9D bitjébe kell helyezni, ami a 0. bitje. Az adatküldés elindul amint feltöltöttük a TXREG regiszter, az első küldött bit a START bit után a TXREG 0. bitje. A bittovábbítás pontosan olyan sorrendben történik, mint amikor a hőmérővel kommunikáltam. Mivel a TXREG feltöltésével indul el az adattovábbítás a TX9D bitet előtte kell feltölteni. Amikor indul az adattovábbítás a 8 vagy 9 bit átmozgatódik a Transmit Shift Regiszterbe, ahonnan a TX lábon át kijut a fogadó felé. Azért van külön Transmit Shift Regiszter, hogy adatküldés közben már tölthetjük is fel újból a TXREG regisztert, hogy elérjük a maximális sebességet

A fogadó PIC az RX lábán megkapja a biteket, amik rákerülnek Receive Shift Regiszterre folyamatosan, érkezési sorrendben. Amint az egész csomag átjutott rákerül a pufferre. A puffer továbbadja a csomagot az RCREG regiszternek, de csak akkor, ha onnan már megtörtént az adat kiolvasása a PIC szoftvere által. 9 bites adatcsomag esetén a 9. bit az RCSTA regiszter 6. bitjén tárolódik, ami az RX9-es bit. Az RCREG-ben lévő Byte kiolvasás után törlődik belőle. A puffer és az RCREG egy FIFO rendszert alkotnak, aminek rövidítése First In First Out, magyarul az elsőnek beérkező csomagot kapja meg a szoftver. Mindig az előbb érkezett csomag van az RCREG-ben és az utólag érkezett a pufferben. Eközben egy 3. csomag éppen töltődhet befelé a Receive Shift Regiszterbe. Ha a PIC processzora nem

dolgozik eléggé gyorsan, és az adat még a késleltetések ellenére sincsen időben beolvasva, egy Overrun Error, magyarul túlcsordulási hiba keletkezik.

Vannak RS-232 kódolások, amelyek két STOP bitet igényelnek, ekkor beállítunk egy 9. bitet 1-esnek, így a STOP bittel együtt két STOP bitet alkotnak.

Nézzük a USART által használt regisztereket. Az SPBRG regiszterrel tudjuk beállítani a baud szintet, tehát hány bps legyen az adatátviteli sebesség. A TXSTA és az RCSTA regiszterekkel lehet beállításokat végezni az USART módon, valamint ezekkel lehet küldeni és fogadni a 9. bitet. Az TXREG és RCREG regiszterekről már minden fontosat elmondtam. A PIR1 regiszter 5. bitje, az RCIF bit, ami 1-es, ha az USART fogadó puffere tele van és 0, ha üres. A PIR1 regiszter 4. bitje, a TXIF bit, ami 1-es, ha az USART küldő puffere üres és 0, ha tele van. A PIE1 regiszter 5. bitje (RCIE) az USART egység fogadó részének az interruptját tudja be és kikapcsolni (1 bekapcs, 0 kikapcs), a 4. bitje (TXIE) pedig a küldő egységének az interruptját tudja be és kikapcsolni (1 bekapcs, 0 kikapcs). A TXSTA, SPBRG és PIE1 regiszterek a BANK1-ben vannak, a többiek a BANK0-ban.

Most beszéljünk a baud generátor beállításáról. Szinkronos slave mód kivételével mindig be kell állítani a bitsebességet. Szinkronos slave módba azért nem kell, mert a másik eszköz mutatja az ütemet. A TXSTA regiszter SYNC bitjével (4. bit) lehet beállítani a szinkron módot, ha a bit 1, és aszinkron módot, ha a bit 0. TXSTA regiszter BRGH bitjével (2. bit) lehet beállítani aszinkron mód esetén, hogy magas sebességű, ha a bit 1, vagy alacsony sebességű, ha a bit 0, módot szeretnénk választani.

Tegyük fel, hogy az eszköz, amivel a PIC-et összekötjük 9600 bps baud értékű, és éppen ezért a PIC-et is erre kell beállítanunk. Az oszcillátorunk 20MHz-es. Ha a BRGH magas sebességre van állítva, akkor ez a képlet érvényes: SPBRG=(Fosc/(16*Baud érték))-1. BRGH 0 esetén pedig SPBRG=(Fosc/(64*Baud érték))-1. Meg kell állapítanunk, hogy az SPBRG regiszterbe milyen értéket kell beírnunk. Magas sebességű esetben (20000000/(16*9600))-1=129,21. Alacsony sebességű esetben (20000000/(64*9600))-1=31,55. Mivel csak egész számokat tudunk 0-tól 255-ig beírni a regiszterbe, a 129-et érdemes, és a BRGH-át pedig 1-re érdemes állítani, mert ott kisebb a kerekítés, mint a 32 esetében. A baud értéket számoljuk ki mindkét esetben, hogy lássuk az eltérést. BRGH=1 esetén baud érték=Fosc/(16*(SPBRG+1)), BRGH=0 esetén meg baud érték=Fosc/(64*(SPBRG+1)). Behelyettesítve az értékeket 20000000/(16*(129+1))=9615 bps és 20000000/(64*(32+1))=9470 bps. Látható, hogy

tényleg az SPBRG 129-es értéke és a BRGH 1-es értéke lesz a 9600 bps-t legjobban megközelítő választás.

Nézzük át részletesen a USART beállításához szükséges TXSTA és RCSTA regisztereket. Kezdjük a TXSTA-val. Főleg a küldés beállítására szolgál, de vannak beállítási lehetőségek, amik a küldésben is és fogadásban is jelentős szerepet játszanak, például a szinkron/aszinkron beállítás. Kezdjük a 0. bitjével, ami a TX9D, ez tárolja a küldeni kívánt 9. bitet. 1. bitje a TRMT, ami 1-est vesz fel, ha a Transmit Shift Regiszter üres, és 0-át vesz fel, ha tele van. Ezzel lehet figyelni, hogy van-e küldés folyamatban:

```
Transmit
btfss TXSTA,TRMT
goto Transmit
```

Ezt fontos figyelni, hogy ne töltsünk a TXREG küldő regiszterbe adatokat, amíg az előző küldést be nem fejeztük.

A TXSTA 2. bitje a BRGH, erről már volt szó. 3. bitje 0, nincs semmire használva. 4. bitje a SYNC, ami 1 szinkron módban, 0 aszinkron módban. 5. bitje a TXEN, ami bekapcsolja a küldő üzemmódot, ha 1, és kikapcsolja, ha 0. Kikapcsolt állapotban hiába írunk adatot a TXREG-be, nem történik semmi. A 6. bitje a TX9, ami 1, ha bekapcsoljuk a 9 bit küldése üzemmódot, 0, ha 8 bitet szeretnénk küldeni. 7. bitje CSRC, aminek csak szinkron módban van jelentősége. 1 esetén mi adjuk az órajelet, tehát master módban vagyunk, 0 esetén pedig a másik eszköz adja az órajelet, amivel össze vagyunk kötve, tehát slave módban vagyunk.

Most nézzük az RXSTA bitjeit. 0. bitje az RX9D, ami a 9. fogadott bitet tartalmazza, ha van. 1. bitje az OERR 1-essel jelzi, ha a 2 bájtos FIFO rendszer megtelt és egy 3. bájt elveszett, mert nem olvastuk ki időben az RCREG-et. Ilyen esetben a 4. bit, a CREN lenullázódik, és ezzel kikapcsolódik az adatfogadás, ezt a CREN bitet szoftveresen kell a PIC-nek 1-esre állítani, hogy újra folyamatos legyen az adatfogadás. 2. bitje a FERR, ami keretezési hibát jelöl, ez esetben nem érkezett meg a STOP bit valószínűleg a keretek elcsúszása miatt, valószínűleg rossz baud érték beállítás miatt. 1-es, ha keletkezett ilyen hiba, 0, ha nem. 3. bitje az ADEN, ami 9 bites aszinkron módban működik, ha 1-esre állítjuk, akkor csak azok a csomagok fognak megérkezni, amelyeknek a 9-edik bitje 1. Így gyakorlatilag több PIC-et lehet rakni 1 vezetékre, és amelyeknek ez a cím felismerés be van kapcsolva, szelektálhatják a bejövő csomagokat. Az 5. bitje a SREN, aminek csak szinkronos master módban van jelentősége. 1-esre állítva egyszeri fogadásra állítódik, tehát 1 bájtot fogad utána megáll a fogadás. A CREN bitet folyamatos fogadásra állítva felülírja a SREN bitet. A 6. bitje

az RX9, ami 1, ha 9 bites fogadásra állítjuk, és 0, ha 8 bites fogadásra állítjuk. 7. bitje a SPEN bit, ami 1-esre állítva konfigurálja a B1 és B2 lábakat a fogadáshoz, valamint küldéshez.

Most nézzük meg egy példaprogrammal, hogyan használtam a PIC beépített RS232-es kommunikációját.

2 db PIC16F628 összekötése USART-tal

Be/kimenetek az adott lábakon:

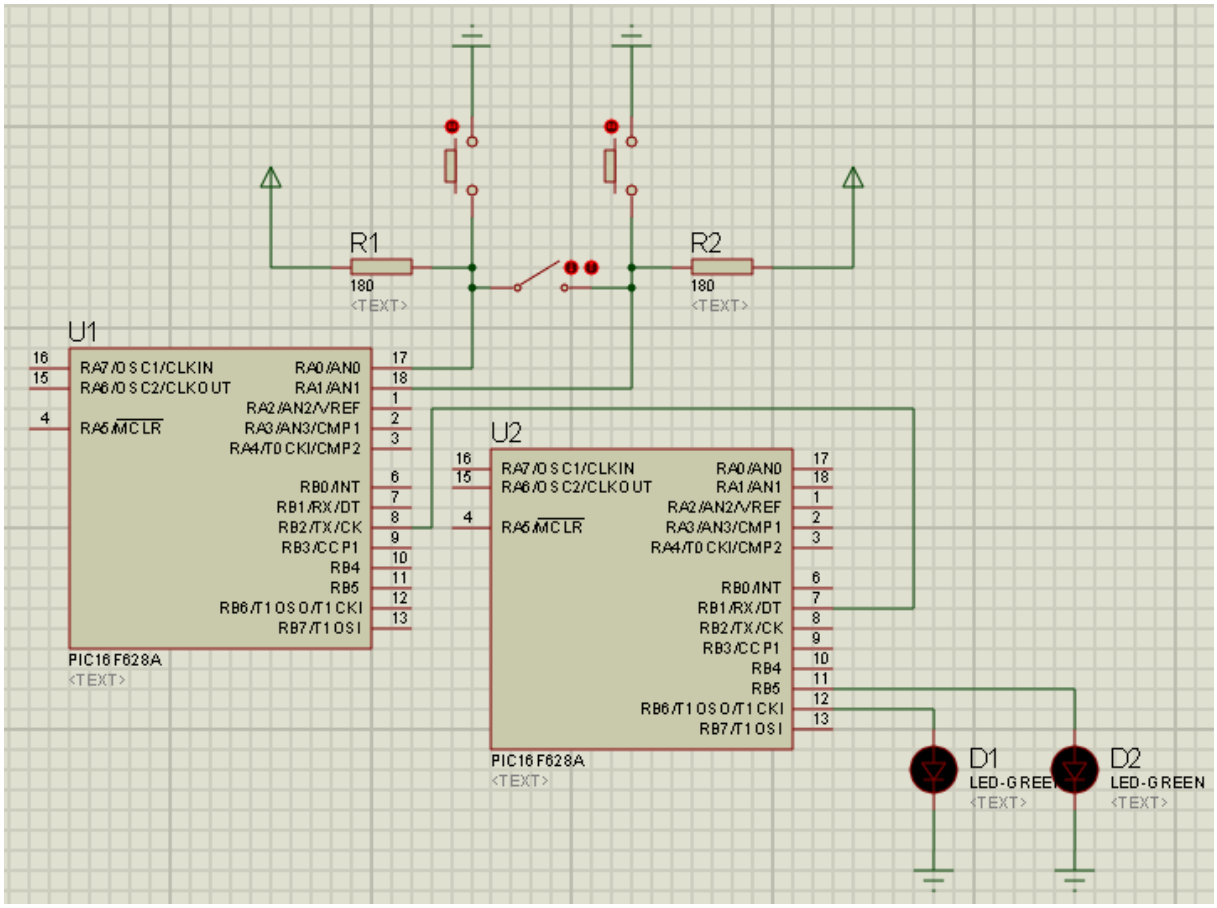
- Bemenetek:
  - bal gomb (1. PIC A0)
  - jobb gomb (1. PIC A1)
- Kimenetek:
  - 2 db zöld LED (2. PIC B5 és B6)
- USART által használt lábak, amelyeken a két PIC össze van kötve:
  - TX (1. PIC B2)
  - RX (2. PIC B1)

Maga a program nagyon egyszerű, a bal gomb lenyomásával kigyullad a bal LED, jobb gomb lenyomásával meg a jobb LED, ha egyszerre nyomjuk le őket, akkor meg mindkettő ég. Ha egy gombot felengedünk, akkor az adott gombhoz tartozó LED elalszik.

Az egyik PIC (1. PIC) a gombnyomásokat érzékeli, a másik PIC (2. PIC) pedig fogadja az első PIC által érzékelt gombnyomásokat USART kommunikációval, és ez alapján kapcsolja be és ki a LED-eket.

Itt látható az áramköri tervezet, amin leszimuláltam az összeköttetést az ISIS segítségével. A 2 nyomógomb közé azért raktam kapcsolót, hogy, ha azt bekapcsolom, akkor olyan, mintha

egyszerre lennének lenyomva. Az egérrel egyszerre csak 1 gombra lehet kattintani, így kellett egy kapcsoló, hogy az egyszerre való lenyomást is szimuláljam:



A két PIC egy vezetékkel van összekötve 8 bites aszinkron módban. Az 1. PIC TX lába van összekötve a 2. PIC RX lábával. A két PIC-ben a programkód egy része megegyezik, most nézzük meg ezeket a megegyező részeket. A ki- és bemenetek beállítását kihagyom, mert erre az előző két programomban lehetett elég példát látni. Annyi, hogy az USART által használt lábakat mindig bemenetre kell állítani, még akkor is, ha adatküldésre szolgálnak.

Kezdjük azzal a résszel, ahol beállítjuk a baud értéket, valamint a TXSTA és RCSTA regisztereket, jelenleg a BANK1-ben vagyunk:

```
movlw d'129'            ;9615 bps-ra állítás
movwf SPBRG
movlw b'00100100'       ;BRGH=1, aszinkron mód
movwf TXSTA
bcf STATUS,5            ;BANK0
movlw b'10010000'       ;folyamatos fogadás
movwf RCSTA
```

Az előző fejezetben lévő példában kiszámolt 129-es értékre állítom az SPBRG regisztert. Lényeges, hogy mindkét PIC ugyanazon bitsebességgel dolgozzon, hogy tudjanak egymással

kommunikálni. A TXSTA regiszter megkapja a 00100100 értéket, azaz magas sebességű küldésre állítom (BRGH=1), aszinkron módba állítom (SYNC=0), 8 bit küldésére állítom (TX9=0), továbbá bekapcsolom a küldést (TXEN=1). Az RCSTA regiszter meg megkapja az 10010000 értéket, azaz fogadó és küldő lábak beállítva (SPEN=1), 8 bites fogadásra állítom (RX9=0), folyamatos fogadásra állítom (CREN=1).

Innentől kezdve eltér a 2 programkód. Az 1. PIC elküldi azt az értéket, amit a 2. PIC a PORTB regiszterébe megkap. A send szubrutin küldi el a w-ben letárolt értéket, így a w értékét kell módosítani az alapján, hogy mely gombokat nyomtam meg:

```
ciklus
clrwdt
movlw b'00000000'
btfss PORTA,0                ;bal gomb bevan-e nyomva?
movlw b'01000000'
btfss PORTA,1                ;jobb gomb bevan-e nyomva?
call checkA
call delay
call send
goto ciklus


checkA
movlw b'00100000'
btfss PORTA,0                ;bal gomb is bevan nyomva a jobb gomb mellé?
movlw b'01100000'
return
```

Az elején csupa 0-ára állítja a w-ét, ami így is marad, ha nem nyomunk be gombokat. Utána megvizsgáljuk, hogy a bal gomb be van-e nyomva, aztán azt, hogy a jobb gomb be van-e nyomva, és végül azt, hogy mindkettő be van-e nyomva. Mivel a 2. PIC PORTB regisztere fogja megkapni a küldött értéket, ezért olyan formátumban küldi ki, amivel a PORTB-ét felülírva pontosan úgy fognak világítani a LED-ek, ahogyan akarjuk. A 2. PIC 5-ös és 6-os lábára vannak kötve a LED-ek, így az 5. és 6. bit módosítása fogja őket be/kikapcsolni. A küldés előtt egy minimális késleltetés is szükséges, különben az elején megadott csupa 0-ás érték küldődik el.

A küldés szubrutinja a következőképpen néz ki:

```
send
movwf TXREG
bsf STATUS,5                ;BANK1
Transmit
btfss TXSTA,TRMT
goto Transmit
bcf STATUS,5                ;BANK0
return
```

A TXREG megkapja az elküldendő bájtot a w-ből, ekkor elindul a küldés, utána átugrunk a BANK1-be, mivel a TXSTA ott található, annak a TRMT bitjét figyeljük, ami 1-et kap értékül, ha a Transmit Shift Regiszter üres, tehát, ha befejeződött a küldés kiugrik a Transmit ciklusból a program, és visszalép a BANK0-ába.

Ott járunk, hogy elküldtük mely LED-eknek kell világítaniuk 2. PIC felé, most nézzük ennek a programkódját:

```
bsf PORTB,5
bsf PORTB,6
call delay
bcf PORTB,5
bcf PORTB,6
```

Ezzel megvillantjuk egyszer a LED-eket, azaz jelezzük, hogy áram alatt van a PIC, és ezután fogadjuk a bejövő adatot a PORTB-be:

```
ciklus
clrwdt
call receive
movwf PORTB
goto ciklus
```

W-be kapja meg a fogadott bájtot, amit rögtön átad a PORTB-nek, ami a kimenetek állapotát kezeli, így világítanak a megfelelő LED-ek. Most nézzük meg a receive szubrutint:

```
receive
clrwdt
btfss PIR1,RCIF
goto receive
movf RCREG,0                ;w-be mozgatja az RCREG értékét
return
```

Amíg a fogadó puffer nem telik meg bent marad a program a receive ciklusban, ugyanis a PIR1 regiszter RCIF bitje figyeli a fogadó puffer állapotát. 1-es értéket vesz fel, ha megtelt. Amint megtelik a puffer, átküldi a bájtot az RCREG-be, ahonnan kiolvassuk, és beírjuk a w-be a movf paranccsal.

Látható, hogy milyen hasznos tud lenni a PIC-be beépített USART. Mindent automatikusan elintéz, csak a bájtot kell megadni neki, amit küldeni akarunk majd kiolvasni a bájtot a fogadó oldalon. Sokkal egyszerűbb használni, mint bitenként kommunikálni a digitális hőmérővel. Be van építve az RS232 szabvány a PIC-be, és ez nagyban megkönnyíti az életet.

Összefoglalás

Sok mindenbe belefogtam, és úgy érzem, hogy minél több mindent tudok, annál inkább zavar a tudatlanságom. Például még mindig nem jöttem rá, hogy miért nem tudom használni a B4 és A4 lábakat. Rájuk kötöttem LED-eket, de nem világítottak, pedig a programkód nagyon egyszerű volt. Úgy éreztem nem tévedhetek ilyen egyszerű kódnál, így valami van a PIC beállításával, amit még nem ismerek. Kezdetben ráfogtam erre a 2 lábra, hogy hibásak, de később, amikor a második PIC-et teszteltem, meglepődtem, hogy ugyanazon lábai nem működnek, mint az elsőnek. Ez már véletlen nem lehet, az én tudásom kevés hozzá. A számítógépes szimulációban ezek a lábak tökéletesen működnek, tehát vagy a szimulációs program írója felejtett ki valamit, vagy mindkét processzor egyformán hibás vagy valami beállítást hagytam ki az adott lábra.

A hőmérő zöld és sárga LED-jeit is kezdetben az A2-re és A3-ra kötöttem, de nem működtek úgy, ahogyan akartam, hogy működjenek. Egyszer jól világítottak máskor meg nem, a szimulációban itt is úgy világítottak, mint ahogyan akartam, hogy világítsanak. Ezért döntöttem úgy, hogy legyen 1-gyel kevesebb piros LED, és akkor a B1-re kötöttem rá a zöld LED-et. A sárga LED maradt az A3-on, ahol eddig nem működött jól. Most, hogy a mellette lévő A2 lábról eltűnt a zöld LED, tökéletesen működik. Pedig a komparátort kikapcsoltam, az nem kavarhat bele, nem értem mi a hiba.

Szóval egy csomó akadályba ütköztem, melyeket csak úgy tudtam kikerülni, hogy máshogyan oldottam meg a kitűzött feladatot, de végeredménynek mindig egy működőképes megoldást találtam.

Továbbá az A5 láb kihasználatlan, mivel oda mindig +5V-ot kell, hogy vezessek, hogy működjön a chip, pedig az nem egyezik meg a VDD lábbal, ahol szintén +5V-ot kap. Ha megszakítom az A5 láb kapcsolatát a +5V-al, akkor még fut a program mindaddig, míg a

vezeték végéhez érve le nem földelem. Ha ismét csatlakoztatom a +5V-hoz, megint elkezd futni az elejéről.

A feladatkitűzés során terveztük, hogy PC-ről lehessen hálózati kapcsolaton keresztül fellépni a PIC-re, és azon beállításokat végezni. Magyarul a hardveremnek kellett volna lennie egy saját IP címének. További megvalósítandó cél lehet, hogy a beléptető rendszer a felhasználó által beállított PIN kódot mentse el egy Pen Drive-on, vagy SD kártyán, hogy ne felejtse el mindig az áram megszűnésekor, majd bekapcsoláskor olvassa le a kódot az adattárolóról.

A hőmérő továbbfejlesztett változatában 7 szegmenses kijelzőt, vagy LCD panelt lehetne alkalmazni. Most, hogy a USART-ot tudom használni LCD panelnek is tudnék küldeni megjelenítendő adatokat. Kommunikálni meg az összes létező PIC tud egymással, amelyekben van USART beépítve.

Összességében elégedett vagyok az eredményekkel, annak ellenére, hogy a termékeim nem piacképesek. Ki az, aki egy 16 kombinációs beléptető rendszerrel biztonságban érezhetné lakását? Vagy ki az, aki átlagfelhasználóként játszi könnyedséggel olvasna le binárisan hőmérsékleteket? Bizonyára mindenki a 7 szegmenses kijelzős vagy LCD paneles változat mellett tenné le a pénztárcáját. Ráadásul a prototípusok egy tesztpanelen vannak elkészítve, a vezetékek könnyen kilazulhatnak, a sok be/ki húzgálástól elszakadhatnak, sérülhetnek.

Az egyedüli része a tesztpanelnek, ami változatlan maradt a diplomamunka végzése folyamán, az a feszültség átalakító rész, valamint a PIC programozó rész.

Szembetűnő, hogy csak 1 fajta PIC-et használtam, a PIC16F628-at, ezért se kellett új PIC programozó részt építenem. Vannak ennél sokkal többet tudó és komplexebb PIC-ek is. Például a 40 lábú PIC18F45K50 a maga 71 parancsával. Hasznos dolog ismerni az általam megismert parancsokat, ugyanis a PIC16F628-as 35 parancsának nagy része benne van a nagytestvérében is. A PIC-em jó választás volt a feladatok végrehajtására, jó, hogy ezt javasolta a témavezetőm, memóriája éppen elég volt a programkód eltárolására. 3,5 kB program memóriája van, és a legnagyobb programom a termosztát programja volt, a maga 1,91 kB-jával. Ehhez a feladathoz például egy 0,375 kB memóriájú PIC10F200 kevés lett volna, ennek a 33 parancsa az OPTION és a TRIS parancsok kivételével ismerősek a PIC16F628 parancsaiból. Egyáltalán nem csoda, hogy miért olyan elterjedt ez a PIC16F628, egyszerűbb használni, mint 500 oldalas kézikönyvű nagytestvérét, de sokkal több kapacitása

van, mint kistestvérének, ugyanakkor a használat bonyolultságában közel megegyezik kistestvérével. Ráadásul még USART-ot is tartalmaz kistestvérével ellentétben.

Az egyetlen dolog, amit én hiányoltam a PIC-ből, az egy analóg-digitális átalakító. Ugyanis hőmérésre elegendő lett volna egy diódának az ellenállás változása hő hatására, ha lett volna benne ilyen átalakító, de mivel nem volt, így csak egy digitális hőmérő jöhetett szóba. Aminek végeredményképpen örültem, mert nem is volt olyan bonyolult vele a kommunikáció, mint eredetileg gondoltam. Ráadásul a digitális hőmérő tartalmaz analóg-digitális átalakítót, így ez megoldotta a PIC analóg-digitális átalakító hiányának problémáját.

A fejlesztésekben nagy örömöt leltem, remélhetőleg a fejlesztésekről olvasni és videókat nézni legalább akkora öröm volt.

A teljes programkódok, a videó és képanyagok, valamint a szakdolgozat pdf formátuma megtalálhatóak a CD mellékleten.

Irodalomjegyzék

1. Épületfelügyeleti rendszerre jelentkező igények:
http://www.boschsecurity.hu/content/language1/html/2757_HUN_XHTML.asp
2. PIC16F628 használati utasítása:
http://ww1.microchip.com/downloads/en/devicedoc/40044d.pdf
3. DS18B20 használati utasítása:
http://datasheets.maximintegrated.com/en/ds/DS18B20.pdf
4. LM7805 használati utasítása:
http://www.sparkfun.com/datasheets/Components/LM7805.pdf
5. Programozó áramkör és szoftver:
http://web.mit.edu/6.115/www/pic.shtml
6. Jimpic-1.9 használati utasítás:
http://web.mit.edu/6.115/www/miscfiles/jimpic-1.9.txt
7. USART leírása:
http://www.eti.pg.gda.pl/katedry/ksg/dydaktyka/dla_studentow//usart.pdf
8. RS-232 wikipédia:
http://hu.wikipedia.org/wiki/RS-232
9. Baud wikipédia:
http://hu.wikipedia.org/wiki/Baud

Köszönöm témavezetőmnek, Vitéz Attilának a hasznos tanácsokat, ötleteket, tippeket! Köszönöm a tesztpanelt, és, hogy lehetőséget nyújtott a munkám elvégzésére!

Köszönöm barátomnak, Róth Richárdnak, aki segített az elindulásban, és egy egyszerűen megépíthető programozó tervrajzot talált a PIC-emhez!

Köszönöm kedves szomszédomnak, Gaál Józsefnek a hiányolhatatlan elektronikai szakértelmét, mely nélkülözhetetlen volt a programozó megépítéséhez!

Köszönöm Potornai Editnek, akivel együtt kezdtem bele eme lehetetlennek látszó feladatba, bármely más utat választ, sok sikert kívánok neki!

Köszönöm egész családomnak, hogy végig mellettem álltak, és állnak most is!