**Név: Trombitás Péter Neptun kód: G08HLM**

**Beadás dátuma: 2016.05.26.**

**Házi feladat címe: Hőmérő riasztó funkcióval**

# Mikrokontrollerek alkalmazástechnikája házi feladat

A feladatot önállóan, meg nem engedett segítség nélkül oldottam meg:

…….
aláírás

# 1 Feladat leírása

8 bites PIC mikrokontroller segítségével megvalósított digitális hőmérő riasztó funkcióval.

Az eszköz folyamatosan figyeli a digitális hőmérő periféria által mért hőmérsékletet. A gombokkal, illetve UART-on keresztül beállítható (és lekérdezhető) alsó és felső határértékek átlépésekor egy piezo buzzer segítségével riasztást ad. A beállításokat, valamint a riasztási események részleteit egy nem felejtő memóriában tárolja, melynek tartalma lekérdezhető UART-on keresztül.

# 2 A feladat részletes specifikációja

A készülendő mikrokontrolleres eszköz elsődleges célja egy „fapados” termosztát megvalósítása. Az alkalmazási környezetben a fűtőegységek nem vezérelhetők automatikával, így ez az eszköz jelez, amikor emberi beavatkozásra van szükség (fűtés ki- és bekapcsolása, illetve szabályozása). Ezen kívül még probléma az is, hogy nagyon lassan melegednek be a berendezések, így ha csak akkor kapcsoljuk be, amikor már tényleg hideg van, akkor több óra alatt melegszik csak fel.

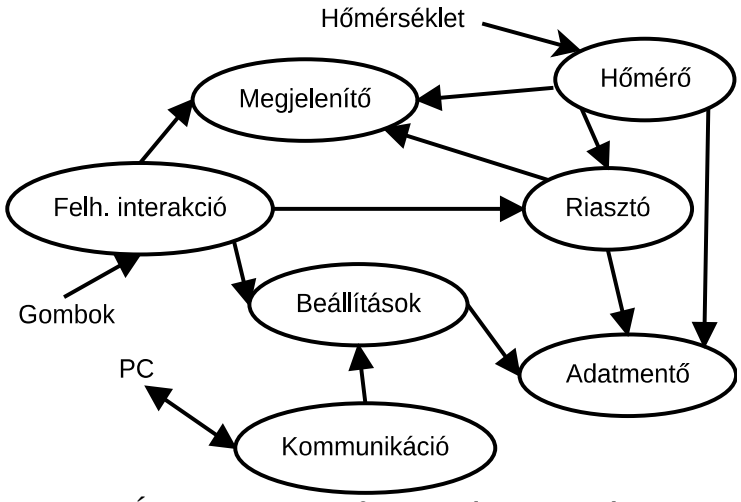
Az eszköz hangjelzéssel tudatja, amikor az általa mért hőmérséklet stabilan beállt a beállított korlátok alá, illetve fölé. Ehhez egy külső hőmérő perifériát, illetve egy piezo buzzert használ. A riasztást egy gomb lenyomásával kell nyugtázni, ellenkező esetben megadott időközönként újra jelez. A határértékek beállításait egy nem felejtő memóriában tárolja. Ezeknek a beállítása többféle módon történhet: egyfelől az eszköz UART-on keresztül elérhető PC-ről, és egyszerű parancsokkal konfigurálható. Másfelől tartalmaz egy LCD kijelzőt is, valamint gombokat, melyekkel egy egyszerű menüs rendszer segítségével is elvégezhetők a beállítások. A nem felejtő memóriában tárolja még a riasztási esemény tényét is.

A megvalósításhoz 8 bites PIC mikrokontrollert fogok használni, PIC16F1718 típusút. Dolgoztam már ilyennel, megbízható, és viszonylag bő erőforrásokkal rendelkezik. Kis fogyasztása miatt is ideális választás. A hőmérő egy DS18B20 típusú, ami 1Wire interfészen keresztül kommunikál a mikrokontrollerrel. A nemfelejtő memória  egy nem túl drága EEPROM chip (25AA040A), ezzel is dolgoztam már. Az USB-UART átalakítást egy egyszerű IC-vel fogom megoldani, lehetőleg olyannal, amely extra külső alkatrészeket nem igényel (lezáró ellenállások, kvarc). A stabil UART működéshez egy kvarc oszcillátort fogok használni órajel forrásként. A tápellátást USB-n keresztül oldom meg.

# 3 Funkcionális blokkvázlat

A rendszer fő komponensei és feladataik:

- **Hőmérő** – Feladata a környezeti hőmérséklet mérése, a mért eredmények szűrése, és így a riasztás alapjául szolgáló adatok előállítása.
- **Adatmentő** – Feladata a riasztási határértékek, illetve a konkrét riasztási események adatainak mentése a nemfelejtő memóriába. Lehetőséget nyújt az elmentett adatok kiolvasására is.
- **Beállítások** – A riasztási határértékek beállítását végzi, validálja a beállítandó értékeket, majd az adatmentő egység segítségével menti azokat.
- **Riasztó** – Feladata a riasztási események fennállásának ellenőrzése, amelyet a hőmérő és adatmentő egységek által szolgáltatott adatok alapján tud megtenni. Esemény bekövetkeztekor az adatmentő, illetve a megjelenítő egységeknek szolgáltat adatot.
- **Megjelenítő** – Feladata mindennemű vizuális és egyéb adatok emberileg értelmezhető formába hozása. Része egy kijelző, valamint egy riasztási egység is (utóbbi hangjelzést tud leadni).
- **Felhasználói interakció kezelő** – A felhasználóval tartja a kapcsolatot. Kezdeményezhetni lehet rajta keresztül a beállítások változtatását, illetve nyugtázható a riasztási esemény.
- **Kommunikáció** – A számítógéppel való kommunikációs csatornát biztosítja. A konfigurációt lehet rajta keresztül lekérdezni, illetve beállítani, valamint megismerhetők az adatmentő egység által mentett riasztási események is.




*1. Ábra: A rendszer funkcionális blokkváza*

# 4 Hardver szoftver szétválasztás

A felhasználói interakcióhoz, illetve megjelenítéshez mindenképpen szükségem volt egy LCD kijelzőre, illetve 3 db gombra (OK, Fel, Le). Ezen kívül bekerült az LCD-hez kapcsolódóan egy potméter is, amivel a kijelző kontrasztját lehet állítani. Ennek a használata akkor lehet előnyös, amikor a kijelző háttérvilágítását váltjuk, illetve eltérő szögekből tekintünk a kijelzőre.

A riasztási rendszer egyetlen extra hardvereleme egy piezo buzzer, amivel egy egyszerű csippanás hangot lehet előidézni a hőmérsékleti korlátok átlépésekor.

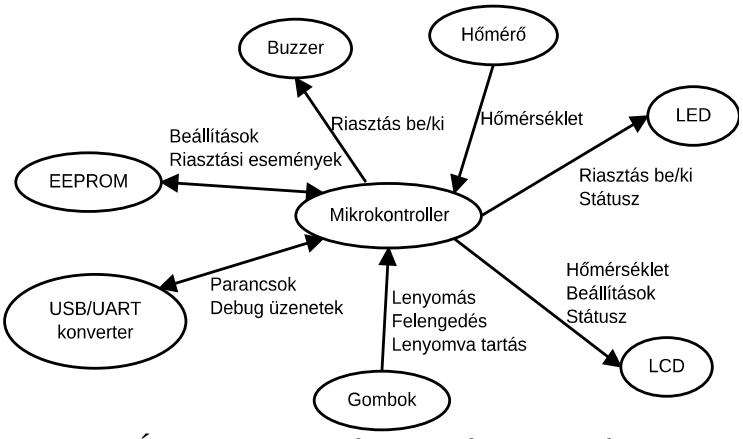
A hőmérő rendszerhez szükség volt egy mikrokontrollerhez csatlakoztatható eszközre, ez egy DS18B20-as. A hőmérsékleti adatok szűrését szoftverből végzem.

Az adatmentő egységhez tartozik egy 4 Kb-s EEPROM memória, amely használatát az indokolja, hogy a választott PIC nem rendelkezik beépített EEPROM-mal (a beépített programmemória írását pedig nem tartottam indokoltnak szoftverből kezelni).

A PC-s kommunikációhoz szükséges volt egy USB/UART kétirányú átalakító. Az MCP2221-re esett a választásom, ugyanis relatíve olcsó, és két darab kondenzátoron kívül semmilyen extra alkatrészt nem igényel.

# 5 Hardver rendszerterv

A hardver-szoftver szétválasztás után a hardeverelemek blokkváza már majdnemhogy magától értetődő. Külön említést érdemel a gombnyomások kezelése, lehetőség van a lenyomás/felengedés, illetve hosszan nyomva tartás kezelésére is – ezt azonban szoftverből kezelem. A kapcsolási rajz a mellékletek közt található.



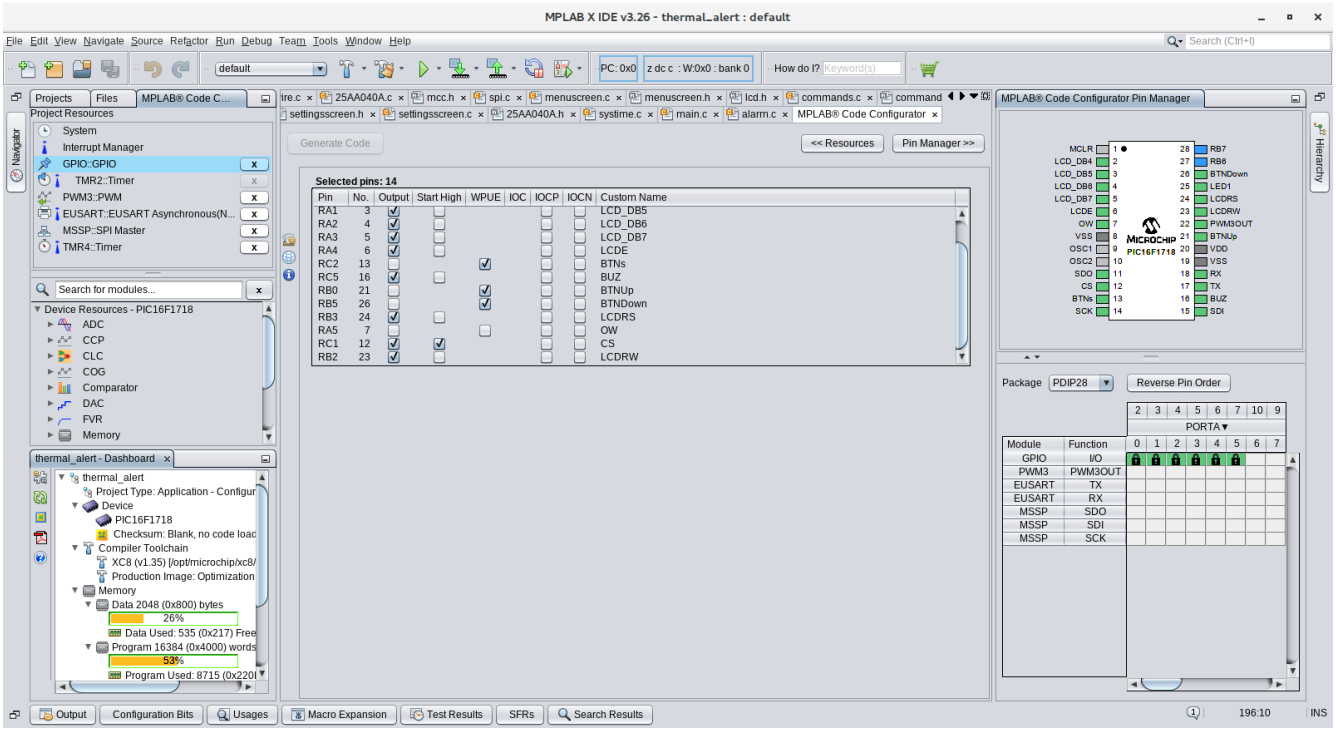

*2. Ábra: A hardver funkcionális blokkváza*

# 6 Szoftver rendszerterv

A szoftver forráskódja alapvetően három egységre tagolható: generált kód, driverek, illetve a működési logikát tartalmazó réteg.

## 6.1 Generált kód

A generált kódot a fejlesztéshez MPLAB X IDE egy beépülő modulja segítségével hoztam létre. Ez az MPLAB® Code Configurator névre hallgat, és egyszerű grafikus felületén „összekattintgatható” a PIC különböző moduljainak beállítása (5. Ábra). Így pofonegyszerűen be lehetett állítani az órajeltől kezdve a PWM-en át mindent. Külön összehasonlítási alap, hogy egyszer már dolgoztam ezzel a típusú PIC-kel, amikor még nem volt elérhető e kódgeneráló plugin. Sokkal több időbe került a dokumentáció végigolvasása alapján beállítani a modulokat, valamint nem is sikerült feltétlen mindent helyesen bekonfigurálni az első próbálkozásra.




*3. Ábra Az MPLAB® Code Configurator felülete*

A generált kód tehát létrehozta a perifériák kezeléséhez elengedhetetlen kódot. Ezek a következők:

- Rendszer órajelének konfigurálása: 4.0 MHz-es kvarc oszcillátort használva 4xPLL segítségével 16 MHz-es rendszerórajelet állítottam be.
- EUSART driver: UART kompatibilis módban használva alkalmas a PC-s kommunikációra.

- PWM driver: az LCD kijelző háttérvilágítását lehet szabályozni.
- SPI driver: az EEPROM-mal való kommunikációt teszi lehetővé. Itt különösen nagy segítségemre volt a grafikus felület, ugyanis a használt PIC egy MSSP elnevezésű modult tartalmaz, amely egyszerre alkalmas I2C és SPI kommunikációra, ebből következően különösen kell figyelni a megfelelő felkonfigurálásra.
- Timer modulok konfigurálása: a TMR2 és TMR4 időzítőket használja a szoftver. A TMR2 64 us periódusidőre van felkonfigurálva, és a PWM modul használja. A TMR4 pedig a maga 1 ms-es periódusidejével a rendszeridő mérésére szolgál (ez nem a generált kód része teljes egészében).
- GPIO lábak konfigurálása: itt lehetett a perifériák kezeléséhez szükséges ki- és bemeneteket konfigurálni, lehetőség van beállítani többek közt belső felhúzó ellenállást az adott portra, alapértelmezett értéket a kimenetre stb. Nagy segítség a GUI-ban, hogy megjeleníti a kiválasztott tokozás lábkiosztását, így vizuálisan nagyon könnyen ellenőrizhető, hogy minden úgy van-e beállítva, mint ahogy azt a kapcsolási rajzon megterveztem.
- Interruptok kezelése: a TMR2, TMR4 és EUSART modulokhoz engedélyezett interrupt rutinok. Az alapfeladatot elvégzik a generált függvények, azonban a TRM4 megszakításkezelőjét kiegészítettem a rendszeridőt növelő kóddal.

## 6.2 Driverek

### 6.2.1 Külső forrásból származó driverek

Mivel a legtöbb használt periféria széles körben elterjedt, szerencsére elérhetők hozzá szabadon felhasználható driverek. Végül kettő ilyet használtam, az egyiket a hőmérő kommunikációs protokollja, az 1-Wire kezelésére, a másikat pedig az LCD kijelző párhuzamos protokolljának kezelésére.

Az 1-Wire protokoll driver kódja a `lib1wire` mappában található. Ezt a PIC-et is gyártó Microchip készítette, és elég megbízhatónak bizonyult. Az LCD kezelő drivert pedig az interneten találtam, de széles körben használatos PIC-es projektekben. Ezt az eXtreme Electronics India tette elérhetővé hobbiprojektek számára. A forráskód a `liblcd` mappában található, amelyet egy, a háttérvilágítást kapcsoló függvénnyel egészítettem ki:

```
/**
 * Ki/be kapcsolja az LCD háttérvilágítását (50%-os fényerőn).
 */
void LCD_ToggleBacklight();
```

### 6.2.2 Saját kódot használó driverek

Az 1-Wire kommunikációs drivert használva már elég egyszerű volt a hőmérő kezelésének megírása. Az elküldendő parancsok, illetve a pontos időzítések, üzenetformátumok leírása megtalálható a DS18B20 adatlapján (a mellékletek közt megtalálható).

A használt hőmérő több felbontású mért adatot is támogat. Az alapértelmezett 0.625°C pontosság feleslegesen nagy – bőven elég a 0.5°C-os pontosság is, ami 9 bites felbontást jelent. A 9-12 bites felbontás beállításán kívül azonban semmi extrát nem tartalmaz a driver. A definiált függvények az alábbiak:

```
typedef enum
{
    DS18B20_Resolution_9bit  = 0x0,
    DS18B20_Resolution_10bit = 0x1,
    DS18B20_Resolution_11bit = 0x2,
    DS18B20_Resolution_12bit = 0x3
} DS18B20_Resolution;

/**
 * Inicializálja a DS18B20 szenzor 1-wire kommunikációját.
 * Küld egy "reset" jelet, és megvárja, hogy visszajöjjön a "presence" jel.
 * @return Sikerült-e az inicializáció.
 */
bool DS18B20_ResetAndDetect();

/**
 * Elindítja a hőmérséklet konverziót a szenzoron.
 * Az adat felbontásától függően a konverzió akár 750ms ideig is tarthat.
 * Ennyi időt kell várni a hőmérséklet értékek kiolvasása előtt.
 */
void DS18B20_StartConversion();

/**
 * Kiolvassa a szenzor regiszteréből a konverzió után ott található értéket.
 * A helyes működéshez meg kell hívni a DS18B20_StartConversion() függvényt, majd
 * megvárni, hogy befejeződjön a konverzió (max 750ms lehet).
 * @return A kiolvasott hőmérséklet értéke NYERSEN. Felbontástól függ a pontosság.
 */
int16_t DS18B20_ReadRawTemperature();

/**
 * Kiolvassa a hőmérő felbontás beállítását.
 * @return Hány bites felbontással üzemel a hőmérő.
 */
DS18B20_Resolution DS18B20_ReadResolution();

/**
 * Beállítja a felbontást a hőmérőn. (9-12 bit lehet).
 * @param res   A felbontás értéke, DS18B20_Resolution enum tagja.
 */
void DS18B20_SetResolution(DS18B20_Resolution res);
```

A gombok kezelésére igazából nem is feltétlen szükséges a driver, azonban viszonylag egyszerűen egy nagyon kényelmesen használható réteget lehet definiálni, amivel többféle esemény is kezelhető.

```
/**
 * Gomb lenoymásakor meghívandó függvény típus.
 */
typedef void (*ButtonPressCallback)(void);

/**
 * Gomb felengedésekor meghívandó függvény típus.
 * @param wasLongPress  Hosszú nyomvatartás előzte-e meg a felengedést.
 */
typedef void (*ButtonReleaseCallback)(bool wasLongPress);

// A "Fel" gomb eseménykezelő callback függvényei
extern ButtonPressCallback     ButtonUp_PressCB;
extern ButtonReleaseCallback   ButtonUp_ReleaseCB;
extern ButtonPressCallback     ButtonUp_LongPressCB;

// A "Le" gomb eseménykezelő callback függvényei
extern ButtonPressCallback     ButtonDown_PressCB;
extern ButtonReleaseCallback   ButtonDown_ReleaseCB;
extern ButtonPressCallback     ButtonDown_LongPressCB;

// Az "Ok" gomb eseménykezelő callback függvényei
extern ButtonPressCallback     ButtonSet_PressCB;
extern ButtonReleaseCallback   ButtonSet_ReleaseCB;
extern ButtonPressCallback     ButtonSet_LongPressCB;

/**
 * A gombok állapotát lekérdező és kezelő függvény.
 * Nem nyújt pergésmentesítést, érdemes két hívás közt néhány (10) ms-t várni.
 */
void handleButtons();
```

A program főciklusában 10ms-enként hívom meg a `handleButtons()` függvényt.

A használt EEPROM egy 4 Kbit-es, 25AA040A típusú, amellyel SPI buszon keresztül lehet kommunikálni. Az adatlapja (mellékletek közt) alapján egyszerűen megvalósíthatók voltak a driver függvényei. Ugyanakkor ügyelni a kellett az időzítésre, ugyanis egy-egy írási művelet után 5 ms ideig várni kell az újabb parancsok küldésével, különben az írás sikertelen. Az alapvető függvényeken kívül még tesztelési, hibakeresési céllal írtam egy olyan függvényt is, amely az EEPROM teljes tartalmát a soros port kimenetére írja, laponként új sorba.

```
/**
 * Kiolvas egy bájtot az EEPROM adott címéről.
 *
 * @param address 9 bites cím, ahonnan az adatot szeretnénk olvasni.
 *
 * @return A kiolvasott bájt.
 */
uint8_t EEPROM_25AA040A_ReadByte(uint16_t address);
```

```
/**
 * Több bájtot olvas egy pufferbe az EEPROM adott címétől kezdve.
 *
 * @param address A kiolvasandó adatok kezdőcíme (9 bites).
 *
 * @param dst   A kiolvasott adatok erre a memóriaterületre lesznek másolva.
 *
 * @param dataLength A kiolvasandó bájtok száma.
 *                   A túlcsorduló rész a 0-s címtől folytatódóan lesz kiolvasva.
 */
void EEPROM_25AA040A_ReadToBuffer(uint16_t address, uint8_t* dst, uint16_t
dataLength);

/**
 * A megadott címre írja a kapott bájtot.
 *
 * @param address   9 bites cím, ahova írni szeretnénk.
 *
 * @param data      A beírandó bájt.
 */
void EEPROM_25AA040A_WriteByte(uint16_t address, uint8_t data);

/**
 * A megadott címtől kezdve dataLength bájtot ír az EEPROM-ba, laponként.
 *
 * @param address A kezdőcím, ahonnan kezdve írjuk a bájtokat.
 *
 * @param data          Az írandó adatok pointere.
 * @param dataLength    Az írandó bájtok száma, maximum 512 - address.
 */
void EEPROM_25AA040A_WriteBuffer(uint16_t address, uint8_t* data, uint8_t
dataLength);

/**
 * A soros port kimenetre olvassa az EEPROM teljes tartalmát.
 */
void EEPROM_25AA040A_Dump();
```

Valahol a driverek fölött, de az alkalmazás logikája alatt helyezkedik el a rendszeridő mérésére szolgála `SysTime`. Ez milliszekundum pontosan képes az idő mérésére. A mért értékek volatile típusként vannak definiálva, mert a TMR4 intterupt rutinjából állítjuk őket.

```
volatile uint16_t TMR4_MsCntr = 0;
volatile uint32_t TMR4_SecCntr = 0;
```

A könnyebb kezelést egy alkalmasan létrehozott struktúra teszi lehetővé:

```
typedef struct _SysTime
{
    volatile uint32_t    Sec;
    volatile uint16_t    Ms;
} SysTime;
```

A rendszeridő lekérdezhető, illetve vannak segédfüggvények az idők különbségének kiszámolására is.

## 6.3 Alkalmazás logika

A hőmérséklet beolvasását, illetve az adatok szűrését egy periodikusan meghívandó függvény, „taszk” végzi. Ezt a főciklus minden egyes lefutásakor meghívjuk.

```
/**
 * Periodikusan meghívandó függvény, amely a hőmérséklet beolvasását, valamint az
adatok szűrését végzi.
 */
void updateTemperature();
```

Ez a függvény egy diszkrét idejű aluláteresztő szűrőt alkalmaz a zajos adatok szűrésére, amelynek a mintavételezési idő (5 másodperc) az időállandója. Indítás után azonban az első 3 értékre nem alkalmazza a szűrést, ezzel elérve azt, hogy nem jelenjenek meg az aktuális hőmérsékletnél jóval alacsonyabb értékek a kijelzőn.

Az aktuális szűrt adatokat pedig változóban tároljuk:

```
// A mért környezeti hőmérséklet egész része.
extern int16_t CurTempInt;
// A mért környezeti hőmérséklet tört része.
extern uint8_t CurTempFract;
```

A riasztási eseményeket egy külön periodikusan futtatandó függvény figyeli és kezeli. Ez minden futáskor megvizsgálja, hogy a hőmérséklet átlépte-e valamelyik korlátot. Ha ez „épp most” történt, akkor menti a riasztás részleteit az EEPROM-ba. Ehhez a következő struktúrát használja:

```
typedef __pack struct
{
    SysTime     Time;
    int16_t     TempInt;
    int16_t     TempFract;
    int16_t     LimitInt;
    int16_t     LimitFract;
    bool        Acknowledged;
    uint8_t     Pad;    // Így lesz egy EEPROM page a mérete
} AlarmEvent;
```

Ennek a felépítésnek két sajátossága van: Az egyik a „Pad” byte a végén, ez arra szolgál, hogy az EEPROM-ba laponként lehessen menteni egy-egy eseményt, ezzel nagyban egyszerűsítve a mentést. A másik a __pack kulcsszó a típusdefinícióban. Erre azért van szükség, mert csak így lehetünk biztosak benne, hogy a struktúra ténylegesen annyi bájtot foglal el, mint amennyit feltétlenül szükséges (a fordítók néha elhelyeznek töltelék bájtokat a struktúrákban, hogy egyes adattagok szóhatárra essenek).

Azt, hogy hány eseményt tárolunk az EEPROM-ban, egy darab bájton tároljuk – szintén az EEPROM-ban, magától értetődő okokból. A lassú kiolvasásokat elkerülendő egy RAM-ban található

változóban is tároljuk az értékét, amelyet indításkor egyszer beolvasunk, illetve minden egyes változott érték írásakor ezt is változtatjuk.

A határértékek beállítására külön függvények vannak, amelyek ellenőrzik a beállítandó értékeket, hogy ne lehessen olyat beállítani, amit nem tud kimérni a hőmérő, illetve hogy ne lehessen „fordítva” beállítani az értékeket (a felső korlát nem lehet alacsonyabb, mint az alsó, és fordítva).

A riasztási határértékek is az EEPROM-ban laknak. Egy ilyen határérték (kissé talán pazarlóan, azonban a jövőben bővítésre lehetőséget adva) 4 bájtot foglal el. Ezeket a nem felejtő memória legelején tároljuk. Az EEPROM-ban tárolt adatok struktúrája egyébként a következő:

| | 0h | 1h | 2h | 3h | 4h | 5h | 6h | 7h | 8h | 9h | Ah | Bh | Ch | Dh | Eh | Fh |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0h | Felső határ | | | | Alsó határ | | | | | | | | | | | EventCtr |
| 0h | Event #0 | | | | | | | | | | | | | | | |
| 2h | Event #1 | | | | | | | | | | | | | | | |
| 3h | Event #3 | | | | | | | | | | | | | | | |
| … | … | | | | | | | | | | | | | | | |

A parancsok értelmezéséhez az `strncmp` függvénnyel hasonlítom össze az UART bemeneti pufferét beprogramozott stringekkel. A sima lekérdezés jellegű függvények esetében a következő lépés egy függvényhívás, a beállító függvényeknél azonban még értelmezni kell a kapott paramétereket. Mivel sajnos a használt fordító ezen a platformon nem támogatja a `scanf` függvényt, így az `strol` használatára voltam kényszerülve. Ennek következtében a tizedes tört értékeket nem egy darab float-ként olvasom be, hanem kettő darab int-ként (egész- és törtrész).

Az LCD kijelzőn megjelenő menü kezelését a gombok vezérlik. Megfelelő callback függvények definiálásával és beállításával lehet a menüpontok közt navigálni. Ezeknek a tárolása egy struktúra tömbben történik.

```
typedef struct
{
    /**
     * Érvényes-e ez az elem. Ha MENU_MAX_ITEMS számú elemnél kevesebbet szeretnénk
     * megjeleníteni, akkor egy olyat kell betenni lezárónak, ahol ez a tag false.
     */
    bool        IsValid;

    /** A kijelzőn megjelenítendő szöveg */
    const char  String[MENU_LINE_WIDTH];

    /** A menüpont kiválasztásakor lefuttatandó függvény. */
    void        (*Callback)(void*);
} MenuItem;
```

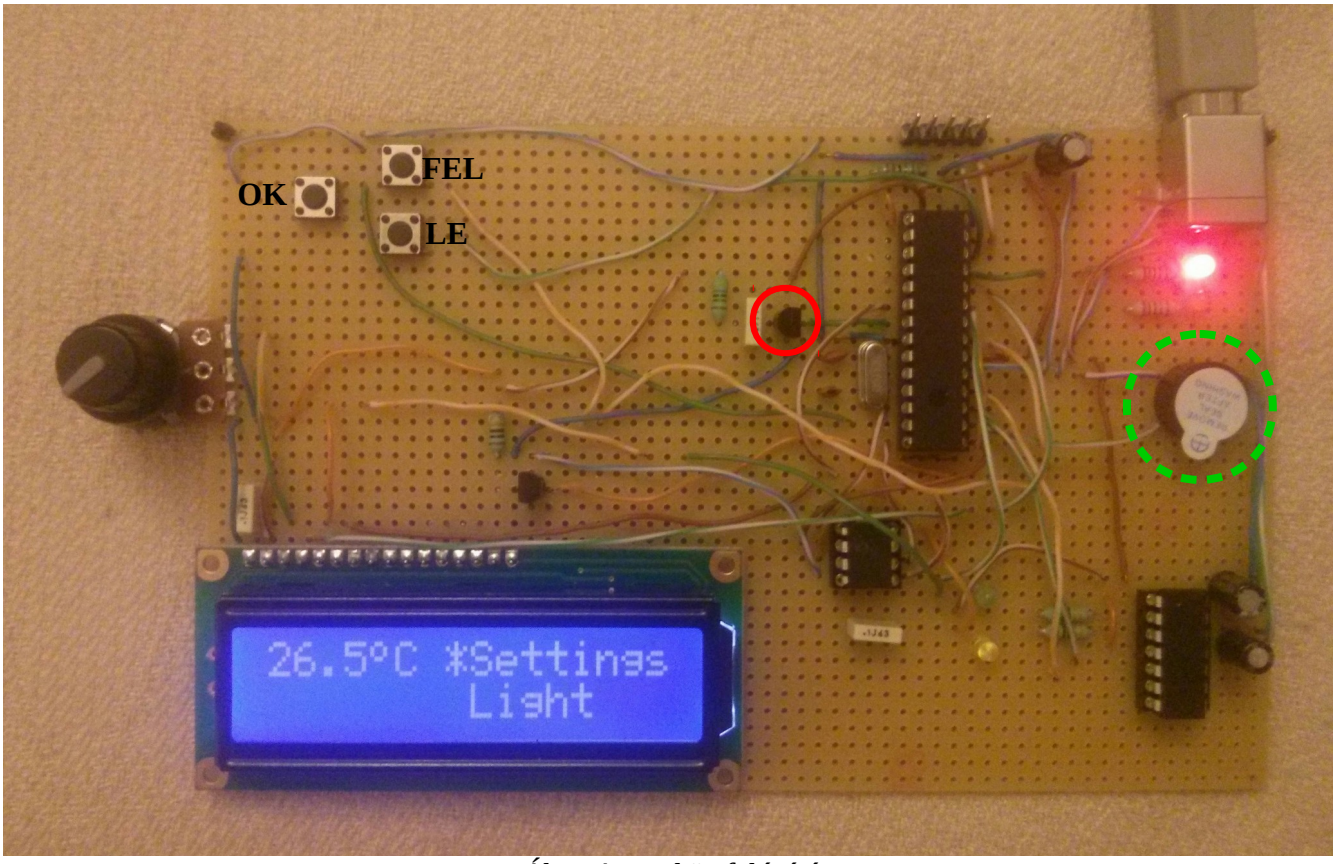
```
MenuItem MainMenu[MENU_MAX_ITEMS] = {
    { true, "Settings", ChangeToSettingsScreen},
    { true, "Light", LCD_ToggleBacklight},
    { false, "", NULL}
};
```

A beállítások képernyőre váltást a `ChangeToSettingsScreen` függvény végzi el. Ez elmenti a gombok callbackjeinek aktuális értékét, majd újakat állít be helyette, illetve az eddig megjelenített menüt is törli a képernyőről, és a beállításokat mutatja a felhasználónak. A képernyő elhagyásakor menti a beállításokat az EEPROM-ba (az adatellenőrzést már a beállítandó értékek állításakor elvégzi), majd visszaállítja a callback-eket az eredetire, végül pedig kirajzolja a főmenüt. Így el lehet érni, hogy a lehető legegyszerűbben tudjunk váltogatni a menüszintek közt, úgy, hogy a legutoljára elhagyott menü állása sem veszik el.

# 7 Felhasználói leírás és kezelési útmutató

Az eszköz 5V feszültséggel működik, amelyet a jobb felső sarokban található USB Type B csatlakozón keresztül kap. Megfelelő tápellátás esetén a csatlakozó melletti piros LED folyamatosan, viszonylag erős fénnyel világít. Ugyanezen csatlakozón keresztül történik a kommunikáció a PC-vel. Az egy virtuális COM portként jelenik meg a számítógép számára.

A hőmérő egység a mellékelt ábrán (4) egybefüggő piros körrel van jelölve. A kijelző bal felső sarkában megjelenő érték az aktuális hőmérséklet. A jobb oldalon látható sorok a menüpontok, * karakter jelzi az aktuálisan kiválasztottat. A menüpontok közt a FEL/LE gombokkal tudunk váltani, az OK gombot megnyomva aktiváljuk az aktuálisat.




*4. Ábra Az eszköz felépítése*

Az „Settings” menüben (5. Ábra) végezhetjük el a riasztási határértékek beállítását. Az OK gombbal válthatunk a két érték közt („U” a felső, „B” az alsó határérték), a FEL/LE segítségével állíthatók az értékek. Az OK gombot hosszan nyomva tartva, majd felengedve visszatérhetünk a főmenübe, ekkor mentődnek a beállítások.


*5. Ábra: A beállítások képernyő*

A második menüpont („Light”) a kijelző háttérvilágításának ki- és bekapcsolására használható. A kijelző kontrasztja a bal oldalon található potméterrel állítható.

A határértékek átlépésekor az buzzer egy rövid, sípoló hangot ad (4. Ábra, zöld szaggatott vonallal rajzolt körrel jelölve). Ezen kívül a kijelző bal alsó sarkában megjelenik az „ALARM” felirat. A sípolás fél percenként ismétlődik, amíg a hőmérséklet vissza nem álla riasztási értékeken belülre, vagy meg nem nyomjuk hosszan az OK gombot – az „ALARM” felirat mindkét esetben eltűnik a kijelzőről.

Az eszközzel USB-n keresztül lehet kommunikálni, egy alkalmas soros terminál segítségével – egy USB-to-Serial emulált COM porton keresztül. Működés közben az eszköz ír ki az aktuális állapotával kapcsolatos információkat, azonban parancsok feldolgozására is képes.

A kiadható parancsok (számít a kis/nagybetű):

- `greet` – az eszköz „Hello world!” üzenettel visszaköszön.
- `temp` – az aktuális hőmérséklet lekérdezésére szolgál.
- `ulimit` – a felső riasztási határérték kezelését teszi lehetővé. Amennyiben nem adunk meg paramétert, az érték lekérdezésére szolgál. A beállításhoz kell írni utána egy szóközt, majd a kívánt hőmérsékletet. Ez utóbbi lehet pozitív vagy negatív, és lehet egész, vagy ponttal tagolt tizedes tört. Tehát például: „`ulimit 20`”, „`ulimit -10.5`”, „`ulimit 30.2`” mind érvényes parancsok. Érvénytelen érték esetén a parancsra adott válaszban jelez az eszköz.
- `blimit` – ugyanaz, mint az `ulimit`, csak az alsó határértéket kezeli.
- `events` – lekérdezhető a múltban történt és elmentett összes riasztási esemény, részletes információkkal.
- `clrevents` – törli az összes eddigi mentett eseményt
- `dump` – kiírja a nemfelejtő memória byte-jait hexadecimális formában