Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Liptai András

SZÍVRITMUS FIGYELŐ

Szakdolgozat

KONZULENS

Benesóczky Zoltán

BUDAPEST, 2015

SZAKDOLGOZAT-FELADAT

Liptai András (BXXMY0)
szigorló villamosmérnök hallgató részére

Szívritmus figyelő

Egyes szívbetegségeknél a szív időnként kihagy. A szivdobbanás kimaradásának ideje a betegség előrehaladtával nőhet és veszélyessé válhat. A feladat egy olyan kicsi, hordozható készülék tervezése, mely optikai elven méri a szívritmust, eltárolja az adatokat, ill. beállítható mértékű kimaradás vagy túlzottan gyors szívritmus esetén figyelmeztető jelzést ad. Az eltárolt adatok kiolvashatók és megjeleníthetők.

1. Végezzen irodalomkutatást a szívritmus figyelők témakörökben! (A szívritmus figyelők működési elvei és felépítésük. A tervezendőhöz hasonló készülékek és ezek összehasonlítása tudás és ár szemponjából. A feladathoz felhasználható legmodernebb célchipek.)

2. Egészítse ki a megtervezendő egység specifikációját a megtervezéséhez szükséges részekkel!

3. Készítse el a készülék teljes hardver és szoftver rendszertervét, továbbá kapcsolási rajzát! A fontosabb tervezői döntéseket indokolja meg!

4. Fejlesztői kártya és modern célchip felhasználásával készítse el a rendszer deszkamodelljét! Írja meg a mikrokontroller programját olyan szinten, hogy a minimálisan elvárható funkciók működjenek!

5. Végezzen gyakorlati próbákat az elkészült deszkamodellel és értékelje az eredményeket!

6. Dokumentálja az elvégzett munkát!

Tanszéki konzulens: Dr. Benesóczky Zoltán, mestertanár

Budapest, 2015. 09. 27.

……

Dr. Jobbágy Ákos
tanszékvezető

Tartalomjegyzék

Összefoglaló **6**
Abstract **7**
1 Bevezetés **8**
1.1 A téma bemutatása 8
1.2 A szívritmus mérésének módszerei 8
1.2.1 Szív elektromos jelein alapuló szívritmus mérés [7][8][9] 8
1.2.2 Vérnyomáson alapú szívritmus mérés [7][8] 9
1.2.3 Szívhang alapú szívritmus mérés [7] 10
1.2.4 Fényelnyelésen alapuló szívritmus mérés [7][8][10] 10
1.3 Szívritmus-figyelésre alkalmas más eszközök 11
1.3.1 Fitbit Charge HR Wireless Activity Tracker Wristband [18] 12
1.3.2 Mio Alpha 2 Heart Rate Sport Watch [19] 12
1.3.3 Garmin Forerunner 225 GPS Strapless Heart Rate Monitor [20] 13
1.3.4 Zencro HRM-2105 Heart Rate Monitor [21] 13
1.4 A feladathoz felhasználható legmodernebb cél-chipek 14
1.4.1 Szenzorok 14
1.4.2 Jelkondicionáló áramkörök 15
1.4.3 Integrált cél-chipek 15
2 Megvalósítás **17**
2.1 Specifikáció 17
2.2 Hardver 18
2.2.1 Hardver rendszerterv 18
2.2.2 Mikrokontroller 19
2.2.3 Mérő-chip 23
2.2.4 Memória 26
2.2.5 A deszkamodell kapcsolási rajza 28
2.2.6 Végleges terv 30
2.2.7 Nyomtatott áramkör 32
2.3 Szoftver 35
2.3.1 I2C interfész kezelés 35
2.3.2 UART interfész 45

2.3.3 Adat-buffer 49
2.3.4 Mérő-chip használata 50
2.3.5 Analóg-digitális átalakító 56
2.3.6 Időzítők 56
2.3.7 Főprogram 57
3 Összefoglalás 64
Irodalomjegyzék 65

HALLGATÓI NYILATKOZAT

Alulírott **Liptai András**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2015. 12. 13.

...……
Liptai András

Összefoglaló

Egyes szívbetegségeknél a szív időnként kihagy. A szívdobbanás kimaradásának ideje a betegség előrehaladtával nőhet és veszélyessé válhat. A szívritmus figyelése, eltárolása, és az esetleges rendellenességek jelzése egy fontos diagnosztikai feladat.

A szakdolgozatban egy olyan eszköz tervezése a cél, amely egy modern cél-chip segítségével képes a szívverés észlelésére optikai módszerrel, mikrokontroller segítségével ebből szívritmust számol, az adatokat eltárolja, és kérés esetén továbbítja a PC-nek. Ezen kívül tervezendő egy deszkamodell, amely nem hordozható, egyszerűbb megvalósítása az eszköznek, de alkalmas a működés ellenőrzésére, és prototípus tervezésre.

A dolgozat egy bevezető résszel kezdődik, amelyben szó van a szívritmus mérésének módszereiről, és ezek összehasonlításáról. Ezután bemutatásra kerül néhány, napjainkban gyakran használt, modern cél-chip, amely optikai módszerrel képes a szívverés detektálására. A szakdolgozat végigvezet az említett eszköz hardverének megtervezésén, bemutatja a tervezés lépéseit, és a tervezési döntések indoklását, majd megismertet a mikrokontroller szoftverének felépítésén, működésén, implementálásán.

Abstract

In some cases of heart functional disorders, the heart might skip. The time of these dropouts might grow with the progress of the disease, and that can become dangerous. The monitoring, storing of heart rate, and signalling of disorders is a very important diagnostic task.

The purpose of this thesis is to design a device, that is capable of detecting the heart beats with an optical method, using a modern heart-rate monitor chip, measures the time between them with a microcontroller, stores these data, and if asked, transmits it to a computer. Besides, a simple model is designed, which is a non-portable, less complex realisation of the device, but it can serve the checking of functionality, and designing a prototype.

The thesis starts with an introduction, which is about the methods of heart measuring, and the comparison of them. After this, some often used modern chips are introduced, that are designed to detect heart beats with an optical method. The thesis demonstrates the designing of the said device, introduces the steps of designing, and the explanation of the important decisions. Then it shows the construction, functionality, and implementation of the microcontroller's software.

1 Bevezetés

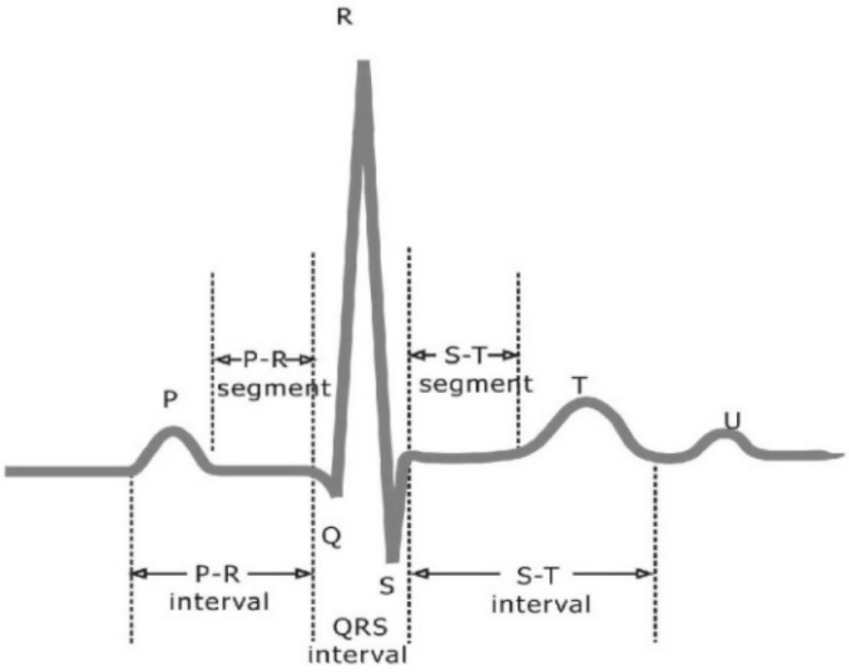
1.1 A téma bemutatása

A félév során a szakdolgozat című tárgy keretein belül a szívritmus mérésére alkalmas módszerek megismerésével foglalkoztam. Megterveztem, és elkészítettem egy modellt, ami eleget tesz ennek a követelménynek, és a mérési adatokat tárolni, valamint továbbítani képes. A modell egy nyomtatott áramköri lemezen megvalósított áramkörből, és egy hozzá kapcsolódó mikrokontroller fejlesztői kártyából áll.

1.2 A szívritmus mérésének módszerei

1.2.1 Szív elektromos jelein alapuló szívritmus mérés [7][8][9]

A vér keringését a szív izmainak periodikus összehúzódása és elernyedése okozza. A jobb pitvar falában található szinuszcsomó minden ciklusban elektromos impulzusokat bocsát ki, amely végigterjed a szív izmain, és összehúzódásra készteti azokat. Ezeket a jeleket a test különböző pontjain elhelyezett elektródákkal detektálni lehet. Ez a mérési elv az elektrokardiográfia (EKG vagy ECG). Az alábbi ábra egy szívverés elektrokardiográfiai hullámformáját mutatja:



1. ábra - EKG hullámforma

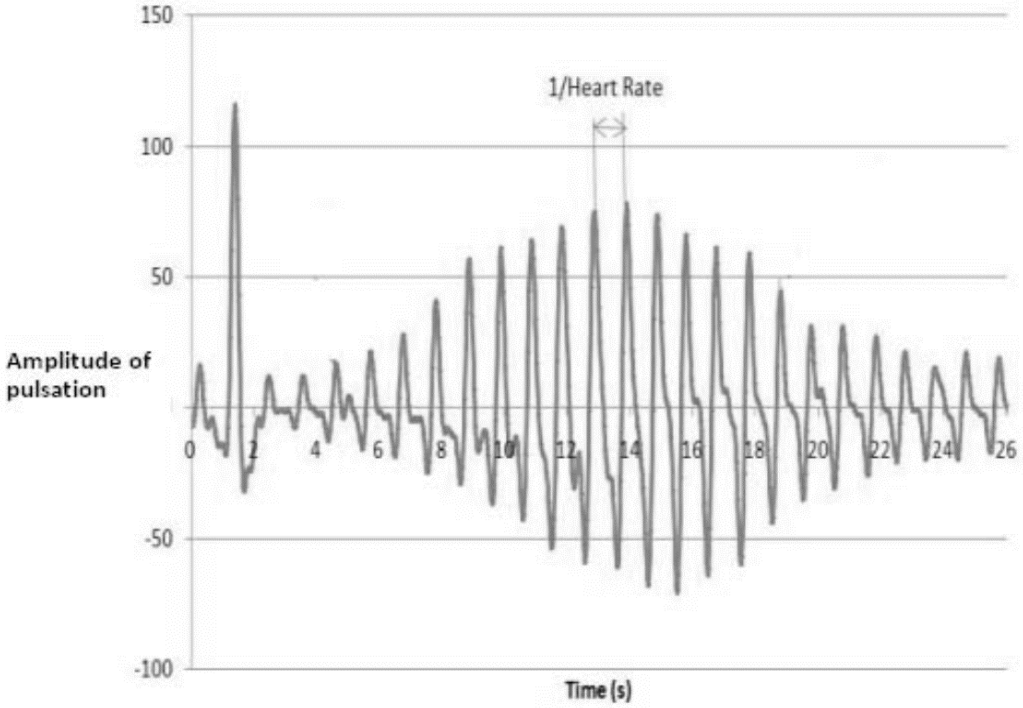
A módszer előnye, hogy nem csak a szívritmust, hanem a szív elektromos jeleit is lehet vizsgálni vele, amelyekből fontos orvosi következtetések vonhatók le.

A szívritmus méréséhez az R jelű csúcsok közt eltelt időt mérik. Természetesen ezt jelentős analóg jelfeldolgozás, szűrés előzi meg.

A módszer hátránya, hogy több elektródát is el kell helyezni a testen, amelyek viselése kényelmetlen. Egy olyan alkalmazásban, ahol egész napos viselés szükséges, viszont csak a szívritmus sebességére, vagy a szív esetleges kihagyására vagyunk kíváncsiak, nem ez a legoptimálisabb megoldás.

1.2.2 Vérnyomáson alapú szívritmus mérés [7][8]

A szív periodikus összehúzódása és elernyedése a vérerekben a vérnyomás pulzáló változását okozza. Ezt egy megfelelő testrészre (mint a csukló) helyezett nyomás érzékelővel mérni lehet, és így a szívritmus számolható. Az ily módon érzékelt, szűrt és erősített jelet oszcillometrikus jelnek nevezik, és az alábbi ábrán látható:

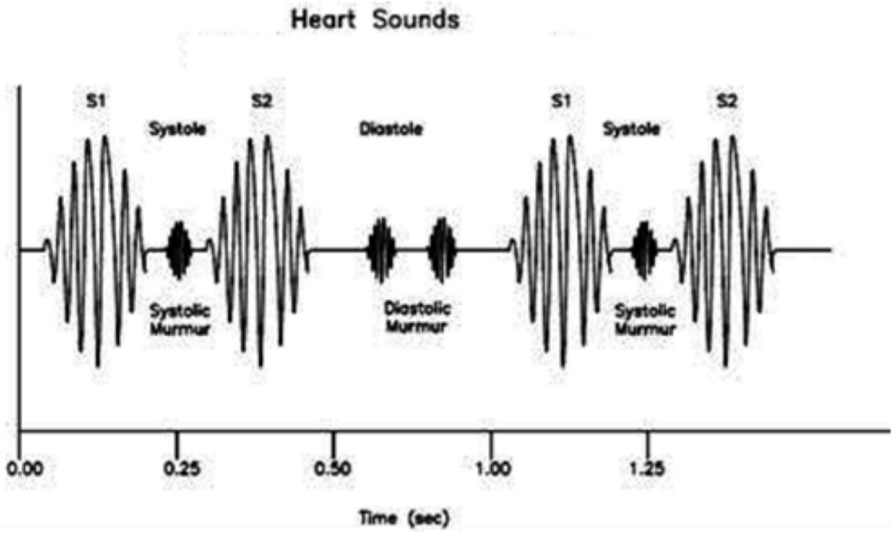


2. ábra - Nyomás alapú pulzusmérés

A mérési eredményről később le kell választani a szívverés szerint pulzáló összetevőt, és az alapján a szívritmus könnyen számolható.

1.2.3 Szívhang alapú szívritmus mérés [7]

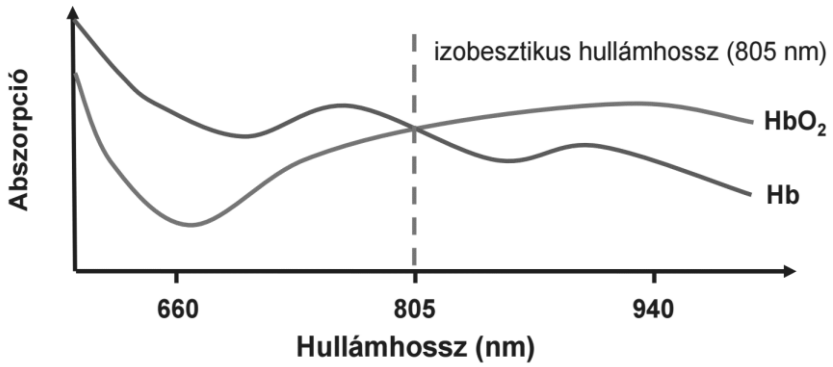
A szív billentyűinek nyílása és záródása hangokat ad az összehúzódás és az elernyedés alatt. Ezek a hangok a szívveréssel egy ritmusban történnek, így alkalmasak a szívritmus mérésére. A mérési elv alapja, hogy a hangokat egy mikrofonnal érzékelik. A normál hangokon kívül a rendellenes hangok is detektálhatóak. A módszer tehát nem csak szívritmus mérésre, hanem megfelelő szűréssel működési rendellenességek monitorozására is alkalmas. Természetesen itt is megfelelő erősítést és zajszűrést kell biztosítani. Az alábbi ábrán az ilyen módon felvett szívhangok láthatók:



3. ábra - Szívhangok

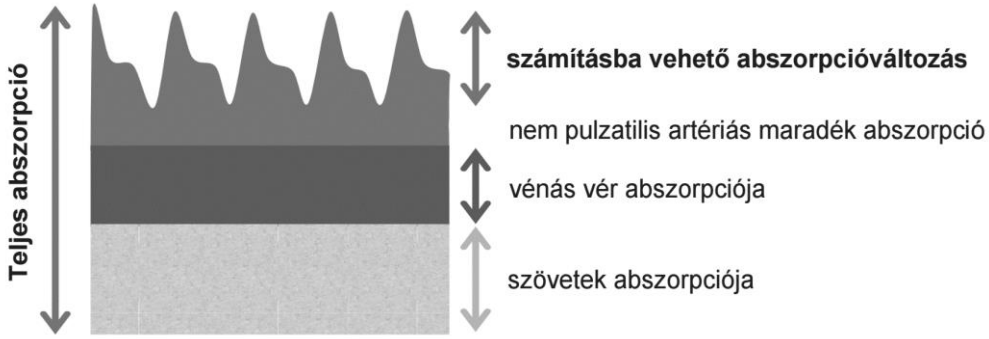
1.2.4 Fényelnyelésen alapuló szívritmus mérés [7][8][10]

Ennek az elvnek az alapja a vér fényelnyelésének mérése. Mint az alábbi ábrán látható, a hemoglobin fényelnyelési spektruma függ az oxigenizáltságtól.



4. ábra - A hemoglobin fényelnyelési spektruma

A vért adott hullámhosszúságú fénnyel megvilágítva a visszavert, vagy áthaladó fény intenzitásából következtetni lehet az oxigénszintre.[8] A kapillárisokban levő vér oxigéntartalma, vagyis az oxigenizált és a dezoxigenizált hemoglobin aránya a szívritmus ütemében változik. Egy vékony testrészt megvilágítva, és az átszűrődő fényt detektálva tehát a szívritmus szerint változó jelet kapunk. A visszaverődés után csak a fény egy része éri el a detektort, és ennek is csak kis része (a pulzáló rész) hordozza az információt.



5. ábra - Teljes fényelnyelés

A jelenségben közrejátszik még az ereknek a véráramlás sebességéből adódó térfogatváltozása is.

A gyakorlatban vörös, és infravörös hullámhossztartományon szoktak mérni, ezeken az oxigenizált, és a dezoxigenizált vér elnyelése jól elkülöníthető. Infravörös tartományon (~900nm) a dús vér nyel el jobban, míg vörös tartományon (~700nm) a kevesebb oxigént tartalmazó. Ezt a jelenséget használják fel pulzoximéterekben is, ahol a két különböző hullámhosszon történő fényelnyelésből következtetnek a vér abszolút oxigéntartalmára. Szívritmus méréshez azonban elég egy hullámhosszon mérni, ami jellemzően az infravörös.

Ennek a mérési módszernek a neve fotopletizmográfia. Hely- és energiatakarékos, viselése kényelmes, és olcsón megvalósítható. Hátránya, hogy nagy érzékenységet mutat a környezeti fényre, a szenzor elmozdulására, és különböző biológiai tényezőkre. A jelen projektben alkalmazott szenzor áramkör ilyen elven működik.

1.3 Szívritmus-figyelésre alkalmas más eszközök

Rengeteg féle szívritmus figyelő eszköz létezik a piacon. Többségük fitnesz, és egyéb sport célokra készült, de vannak, amik az alvás monitorozására is alkalmasak. A legmegbízhatóbb mérőeszközök a mellkas-pánttal rendelkező, lényegében az

elektrokardiográfia elvét felhasználó műszerek, illetve az ujjra csíptethető pulzoximéterek. Kizárólag pulzusmérésre is léteznek eszközök, amik a pletizmográfia elvét alkalmazzák. Ezek csuklóra erősíthető karórák, hátlapjukon egy optikai szenzorral. Ezek az eszközök gyakran elég pontatlanok, és csak nyugalomban lehet velük mérni. Az alábbiakban, mivel a piacon rengeteg ilyen termék van, a teljesség igénye nélkül összehasonlítanék néhány pánt nélküli pulzusmérő eszközt.

1.3.1 Fitbit Charge HR Wireless Activity Tracker Wristband [18]

Ez az eszköz egy folyamatos, automatikus szívritmus figyelő csuklópánt. Egész nap figyeli az aktivitást, szívritmust mér, méri a megtett távolságot, számolja az elégetett kalóriákat, számolja az aktív perceket, és hogy hány emeletet mászott meg a felhasználó. Automatikusan figyeli az alvást. Vezeték nélkül szinkronizálható okostelefonnal és számítógéppel, amelyekről így a mért adatok leolvashatóak. Valós idejű statisztikákat számol sportolás közben, és összegzést utána. Akkumulátora akár 5 napig is bírja. Ez egy modern eszköz, ami rendkívül sok funkcióval bír az említetteken kívül is. Ára körülbelül 52000 forint.

1.3.2 Mio Alpha 2 Heart Rate Sport Watch [19]

Ez az elsősorban sporthoz készített eszköz szintén egy nagyon fejlett pulzusmérő óra. Rengeteg funkciója közül csak néhányat emelek ki. 24 óra edzés adat tárolására képes, és az eredmény később szinkronizálható. Akkumulátora egyhuzamban 10 óra pulzusmérést tesz lehetővé. Beállíthatóak rajta különböző szívritmus-zónák, amelyek elhagyásakor az eszköz figyelmeztető jelzést adhat. Blootooth kapcsolattal csatlakoztatható különböző fitnesz alkalmazásokhoz. Ára 57000 forint.

1.3.3 Garmin Forerunner 225 GPS Strapless Heart Rate Monitor [20]

Ez a csúcskategóriás készülék is sport célokra készült elsősorban. Az alapvető fitnesz adatok számításán, mint például az elégetett kalória és a szívritmus mérésén kívül méri a megtett távolságot, a beépített gyorsulásmérőnek köszönhetően akár beltérben is. Párosítható okostelefonnal, különböző alkalmazásokkal. Rendelkezik közösségi hálós lehetőségekkel, valós idejű követéssel. A gyártó weboldaláról edzéstervek tölthetők az eszközre. Egy óra tétlenség után jelzi, ha ideje mozogni. Színes kijelzője kényelmes felhasználói felületet biztosít. Csak óraként használva 4 hét, pulzusmérés közben 10 óráig alkalmazható. 200 órányi aktivitás eltárolására képes. Ára körülbelül 87000 forint.

1.3.4 Zencro HRM-2105 Heart Rate Monitor [21]

Ez az eszköz kissé eltér az előbbiektől. Nem egy komplett szívritmus figyelő eszköz, csak egy ujjra vagy fülre csíptethető szenzor egy átalakítóval, amivel telefonhoz csatlakoztatható. Az átalakító jack dugóval csatlakoztatható a telefonhoz, és szívritmus adatokat küld. Több alkalmazás is van, ami kompatibilis az eszközzel, és ezek részletes fitnesz statisztikák számítására képesek. A szenzor USB kábelen keresztül számítógépre is csatlakoztatható, és ott a megfelelő programmal a szívritmus elemzése, vagy valós idejű kijelzése lehetséges. A termék ára 10000 forint körül van.

1.4 A feladathoz felhasználható legmodernebb cél-chipek

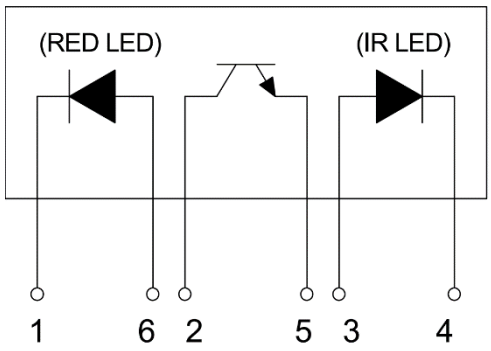
A szívritmus pletizmográfiával történő mérésének céljára alkotott áramkörök alapvetően három csoportba oszthatók: Az első két kategória a szenzorok, és a jelfeldolgozó áramkörök. A szenzorok általában csak egy fotodiódából, és néhány LED-ből állnak. A jelfeldolgozó áramkörök (front-end) pedig általában LED vezérlést, analóg-digitális átalakítót, analóg és digitális jelfeldolgozó áramkört, és valamilyen soros kommunikációs interfészt tartalmaznak. A harmadik kategória az olyan áramkörök, amelyek a kettőt integrálva tartalmazzák.

1.4.1 Szenzorok

Ezek egyszerű áramkörök, általában nem programozhatóak, lényegében csak LED-eket és fotodiódát tartalmaznak. Ennek ellenére hasznos eszközök, mert kialakításuk úgy van megcsinálva, hogy a lehető legoptimálisabb legyen a pletizmográfia alkalmazására, például megfelelő szűrő üveggel vannak lefedve, illetve a fotodióda, és a LED-ek közt elválasztó fal biztosítja, hogy a fény csak az ujjon keresztül jusson el a fotodiódához.

Ilyen, modern szenzor például a NJL5501R[14], egy infravörös és egy vörös LED-el ellátva, vagy a SFH7050[15], amely vörös, zöld, és infravörös LED-eket tartalmaz.

7. ábra - Az SFH7050 képe



6. ábra Az NJL5501R Blokkvázlata

1.4.2 Jelkondicionáló áramkörök

Ezek már jóval bonyolultabb áramkörök, gyakran programozhatók, és automatikus szabályzással is rendelkeznek.

1.4.2.1 AFE4400 [11]

Az AFE4400 a Texas Instruments modern, szívritmus figyelők és pulzoximéterek számára gyártott analóg jelfeldolgozó és digitalizáló áramköre. Működése, időzítése nagymértékben programozható. Beépített, beállítható LED vezérlővel rendelkezik. Órajelének létrehozásához egy 8Mhz-es kvarcot kell csatlakoztatni. A mikrokontrollerrel SPI interfészen keresztül képes kommunikálni. A chip a normál mérési adatok mellett az ambiens fényt is méri, vagyis úgy is vesz mintát, hogy egyik LED sem világít. Ezt az adatot elküldi a mikrokontrollernek, amely ez alapján beállíthatja az eszköz analóg ambiens fény elnyomását, hogy kiküszöbölje annak hatását. Ezen kívül az eszköz különböző diagnosztikai üzeneteket is küld, ha valami hibát észlel. Lassabb mintavételezés esetén az energiafogyasztás mA nagyságrendűre csökkenthető. Tápfeszültségének a 3.3V megfelelő.

Ez az áramkör könnyen kezelhető, és sok szempontból optimális. Munkámhoz azonban olyan áramkört kerestem, amelyben a szenzor is integrálva van.

1.4.3 Integrált cél-chipek

1.4.3.1 PAH8001EI-2G [12]

Ez egy nagy teljesítményű, alacsony energiafelhasználású integrált áramkör, ami analóg és digitális jelfeldolgozást tartalmaz. Csak 1 darab (zöld) LED van benne, így pulzoximetriára nem alkalmas, szívritmus mérésre viszont igen. Kétféle alvó móddal rendelkezik, amelyek csökkentik az áramfogyasztást. Az alvó módokon kívül, tehát normál működés esetén a chip áramfogyasztása 1.5mA. Változtatható kommunikációs interfésszel rendelkezik, amit I2C-re, két vezetékes SPI-ra, vagy négy vezetékes SPI-ra lehet beállítani. Ezek maximum 1Mbit/s sebességre képesek. A működéséhez 3.3V szükséges.

A chip nagymértékben programozható. Az alvó módokba lépéshez beépített időzítők vannak, amik egy érintés érzékelő alapján mérik az inaktivitást. Az eszköz a beágyazott szoftver-algoritmus miatt fejlett digitális jelfeldolgozással bír.

Az áramkör rendkívül praktikus, könnyen használható, energiatakarékos. Jelen projekthez már-már fölöslegesen fejlett.

1.4.3.2 MAX30100 [3]

A Maxim Integrated Products MAX30100 nevű integrált áramköre egy direkt pulzoximetriára és szívritmus mérésre alkalmazható áramkör. Két beépített LED-et (vörös és infravörös), fotodiódát, analóg jelfeldolgozó áramkört, és analóg digitális átalakítót tartalmaz. Áramfelvétele kicsi, vezérlése I2C interfészen keresztül történik, amelynek frekvenciája maximum 400 kHz lehet. Ez a chip is nagymértékben konfigurálható. Automatikus ambiens fény elnyomással rendelkezik. Bár pulzoximetriára is képes, megfelelő beállítás mellett optimális egyszerű szívritmus mérésre is. Áramfogyasztása rendkívül kedvező, akár 0.6mA-re is lecsökkenthető.

A chiphez készült egy fejlesztést és prototípus gyártást segítő kártya, amely tartalmazza az eszközt, és a hozzá szükséges kiegészítő áramkört.[16].

Mivel a konzulensemnek ez a chip állt rendelkezésére, a projekthez ezt használtam fel, működéséről ezért később részletesebben is írok.

2 Megvalósítás

2.1 Specifikáció

A feladat megvalósításához szükség volt arra, hogy a kezdeti specifikációt a téma körüljárása után tervezői szempontok figyelembevételével kiegészítsem, illetve pontosítsam.

A mérési eredményeket a mikrokontroller I2C interfészen keresztül lekérdezi, majd egy algoritmus segítségével detektálja a szívveréseket, és méri a köztük eltelt időt. A kiszámolt idő-adatokat az eszköz egy memóriába írja, amely kellő méretű, hogy el lehessen tárolni benne 24 óra adatait. A memóriának a tápellátás kimaradása, a készülék újraindulása, kikapcsolása, vagy lemerülése esetén is meg kell tartania az elmentett adatokat. Az eszköznek UART-on keresztül kommunikálnia kell a PC-vel, többek között el kell küldenie a mért idő-adatokat a PC-nek. Ehhez előre definiált parancsok kellenek az eszköz vezérlésére. Az UART kommunikáció USB kábelen keresztül zajlik, UART/USB átalakító segítségével. Az eszköz a reális pulzusszám tartományon tudjon mérni (50bpm-200bpm)!

A szívritmus figyelő eszköz deszkamodellje egy nyomtatott áramkör formájában legyen megvalósítva. Ez a nyomtatott áramkör az egyik oldalán tartalmazza a kiválasztott mérő-chipet, a másik oldalán pedig a többi szükséges alkatrészt. A nyomtatott áramköri lemezre tüskesorok legyenek forrasztva, amellyel az közvetlenül csatlakoztatható a fejlesztői kártyára forrasztott hüvelysorokba.

2.2 Hardver

2.2.1 Hardver rendszerterv

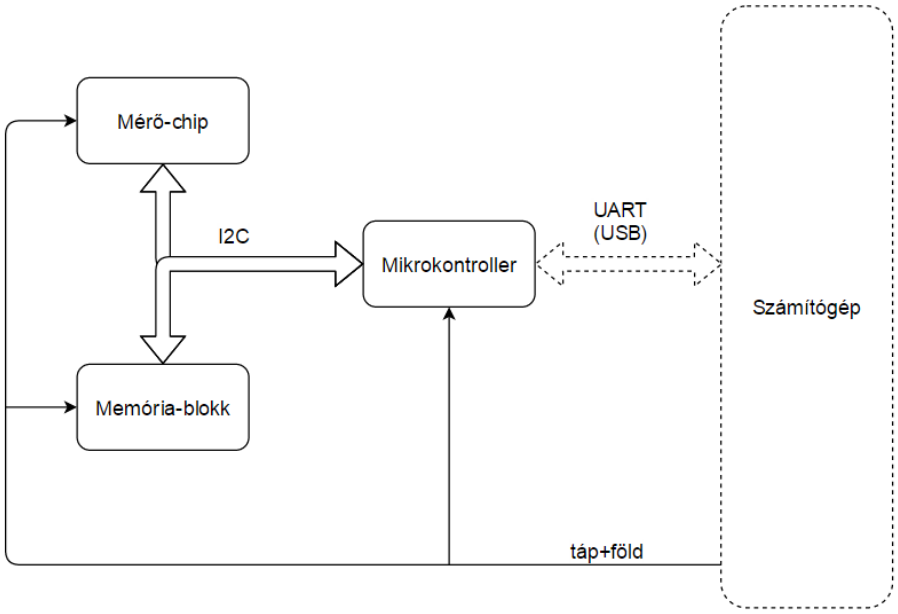
A hardver alapvetően 3 részből áll:

Egy szenzor, ami detektálja a szívritmussal változó fényerőt. Jelen esetben ez egy összetett, programozható mérő-chip, ami maga egy analóg jelfeldolgozó áramkör, és kommunikációs interfész is egyben. A mért adatokat digitális formában továbbítja az eszköznek.

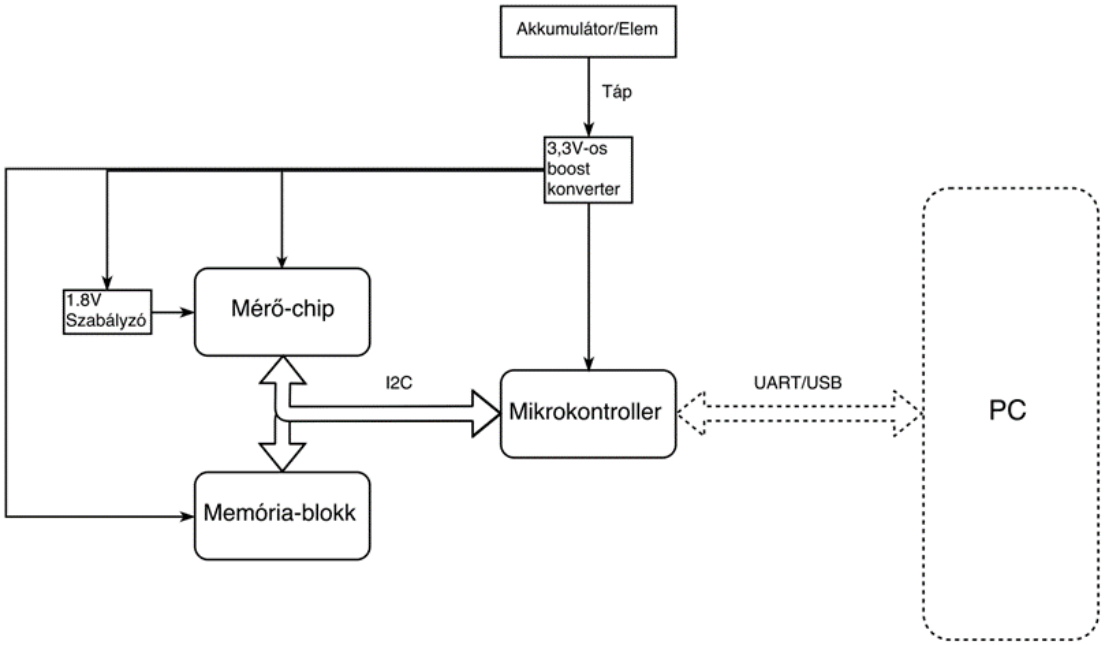
Egy memória, ahova az eszköz eltárolja a mérési eredményeket, és szükség esetén kiolvassa onnan.

Egy mikrokontroller, ami az egész eszköz működését koordinálja. Lekéri a szenzor mérési adatait, értelmezi azokat, átszámolja a kívánt formára, majd elküldi a memóriaegységnek. UART-on keresztül kommunikál a számítógéppel, és kezdetleges felhasználói vezérlést biztosít.

A deszkamodell, és a végleges eszköz funkcionális blokkvázlata az alábbi ábrákon látható. A számítógép azért van szaggatott vonallal jelölve, mert nem tartozik közvetlenül az eszközhöz, csak az adatok leolvasásához, illetve módok közti váltáshoz kell elvileg csatlakoztatni. A deszkamodellnél azonban mindig csatlakoztatva kell lenni hozzá, mert az eszköz egyelőre a tápot is innen kapja, USB kábelen keresztül.

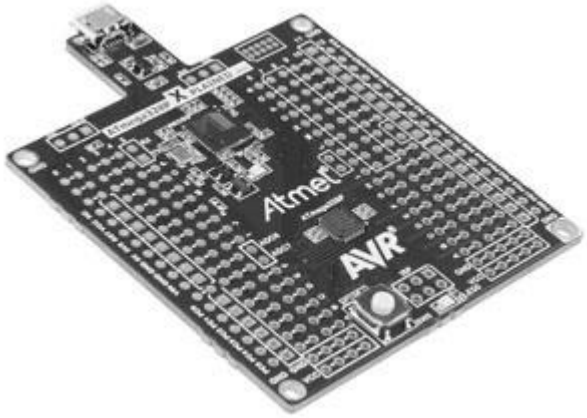


8. ábra – A deszkamodell funkcionális blokkvázlata



9. ábra - Végleges funkcionális blokkvázlat

2.2.2 Mikrokontroller



A fejlesztésre használt deszkamodellben és a végleges prototípus tervben felhasznált mikrokontroller az ATmega328P, az ATmega328P Xplained Mini fejlesztői kártyával. Először azért választottam ezt a kártyát, mert egy tavalyi munkámhoz már használtam, így egyrészt megvolt, másrészt valamennyire ismertem a kezelését. Természetesen ezután mérlegeltem, hogy a mikrokontroller valóban alkalmas-e a feladat megvalósítására.

Mivel a szoftver algoritmus nem olyan bonyolult, és nem igényel nagy számolási kapacitást, ez a mikrokontroller sebesség szempontjából épp megfelel a célnak. Fontos volt, hogy a mikrokontroller rendelkezzen I2C támogatással, hogy a kiválasztott mérő-chippel kommunikálni tudjon. Az ATmega328P által támogatott (Two-Wire-Interface) kompatibilis az I2C protokollal, így a mikrokontroller ebből a szempontból is megfelel. Szükség van még hardveres UART támogatásra és külső interrupt kérő lábra a szenzor chiphez. Az említett mikrokontroller ezeket is tartalmazza.

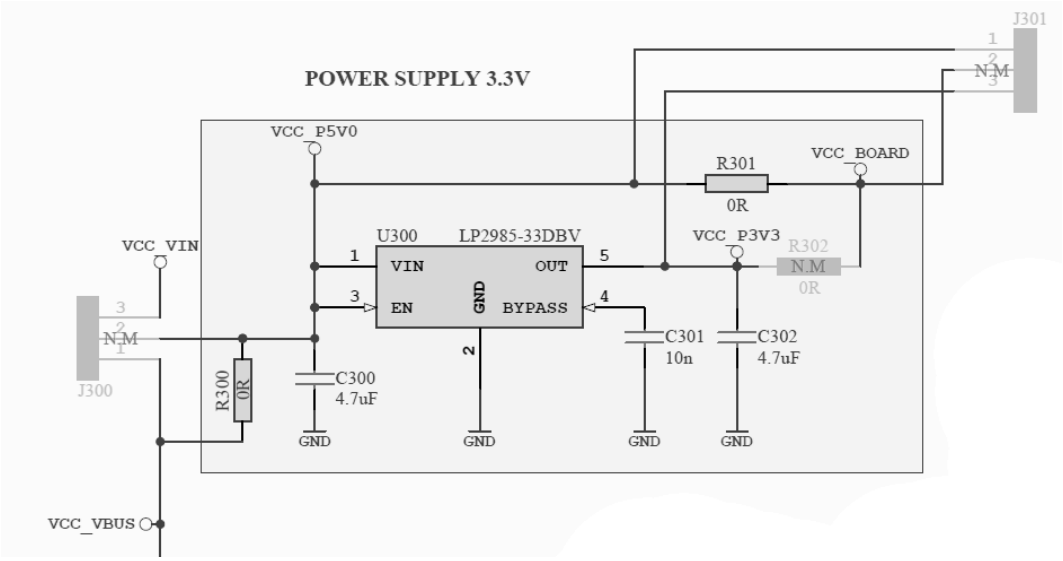
Ha olyan eszközt kellett volna tervezni, amiről az adatok USB-n keresztül, mint egy tárolóról leolvashatók, és kezelhetők, akkor célszerűbb lett volna olyan mikrokontroller, ami az USB protokollt hardveresen támogatja. Jelen esetben azonban erre nem volt szükség, megelégszünk azzal, hogy USB-n keresztül történő UART kommunikációval kommunikálunk a számítógéppel.

Az ATmega328P mikrokontroller olyan szempontból is jó, hogy nem csak 5V, hanem 3.3V tápfeszültséggel is tud működni, ami a rendszer többi alkatrészének is megfelelő. A fejlesztői kártya USB kábelen keresztül kapja 5V-os tápját. Ebből egy beépített feszültségszabályzó segítségével lehet előállítani a 3.3 V-os tápfeszültséget. Ehhez a fejlesztői kártyán véghez kellett vinni néhány változtatást. A fejlesztői kártya leírásában[1] találtam a következő táblázatot, amelyben kiemeltem a releváns sort:

Mode	J301 connection, target	J300 connection, board	Function
5V (Default)	pin2 connected to pin1	pin2 connected to pin1	Board and target powered by VBUS
3.3V USB	pin2 connected to pin3, remove R301	pin2 connected to pin1	Target powered by 3.3V and USB interface powered by VBUS
VIN	pin2 connected to pin1	pin2 connected to pin3, remove R300	Board and target powered by VIN (J202.8). 1.8V < VIN < 5.5V
3.3V VIN	pin2 connected to pin3, remove R301	pin2 connected to pin3, remove R300	Target powered by 3.3V. VIN (J202.8) as regulator input. 4V < VIN < 16V

10. ábra - Fejlesztői kártya tápellátás opciók

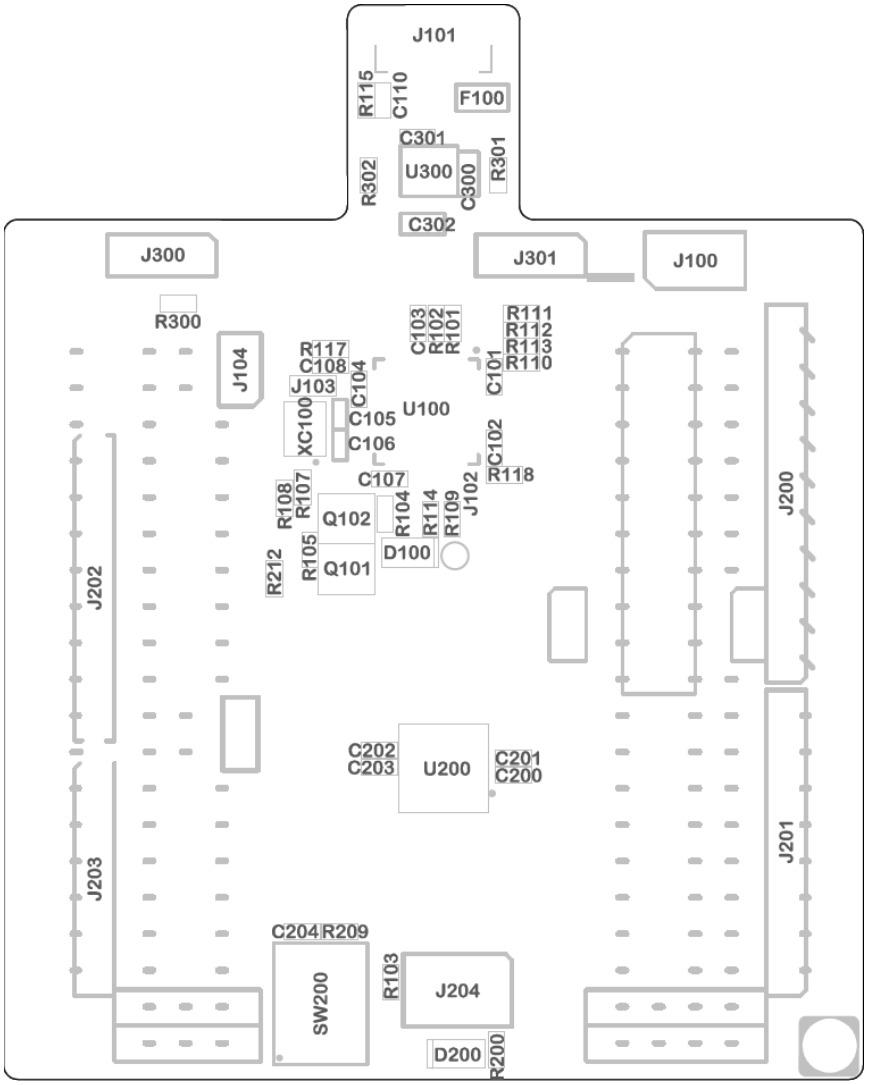
A fejlesztői kártya dizájn-dokumentációja[2] alapján beazonosítottam a megfelelő pin-eket és az R301 és R300 (0Ω-os) ellenállásokat.



11. ábra - A fejlesztői kártya tápellátását szabályzó áramkör

Látható, hogy a VCC_BOARD-ra akkor kerül 5V, ha R301 be van forrasztva, a J301 jumper a 2-es és a 3-as pinje pedig nincs összekötve, és akkor kerül 3.3V, ha az R301 ellenállás el van távolítva, a J301 2-es és 3-as pinje pedig össze van kötve.

A következő ábra alapján az is látható, hogy a fejlesztői kártyán hol helyezkednek el az említett elemek. A J301 jumper pin-jeit nem volt nehéz beazonosítani, mert a kártyán az 1-es pin-nél 5V, a 3-as pin-nél 3V3 felirat van.

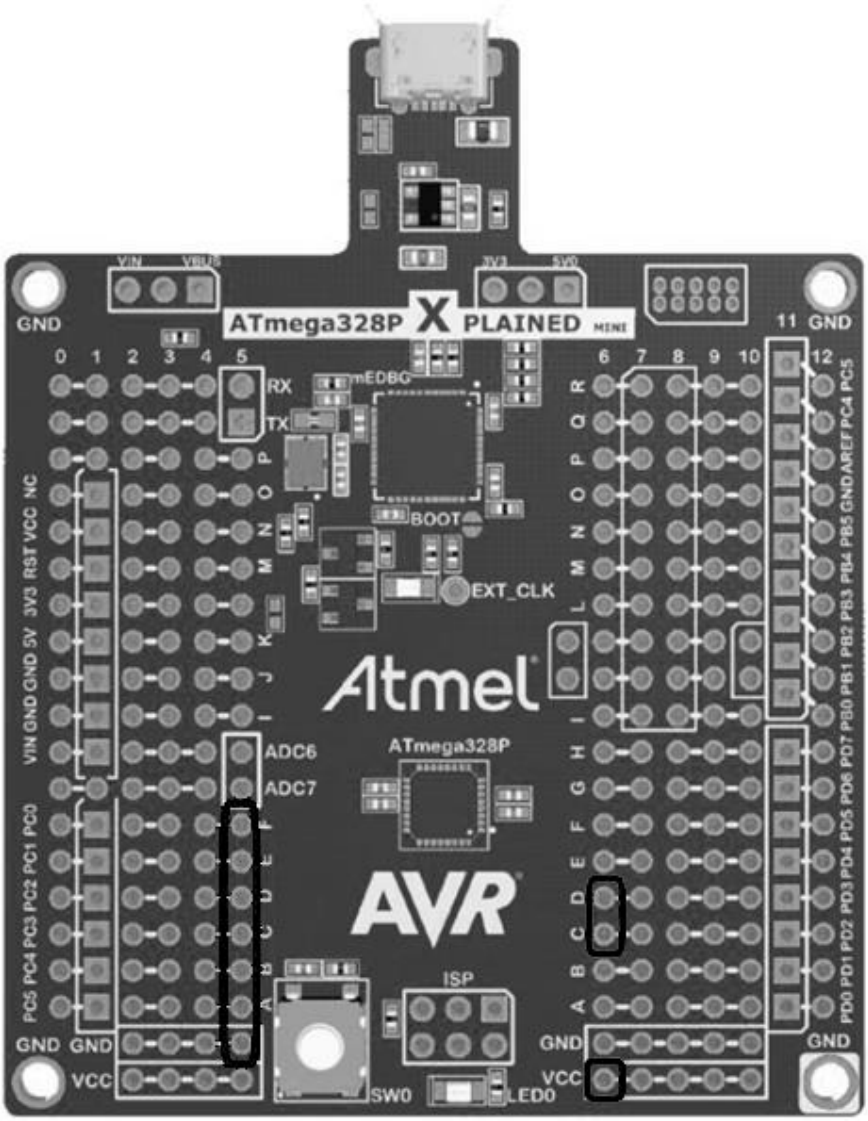


12. ábra - alkatrészek elhelyezkedése a fejlesztői kártyán

A fentiek alapján a fejlesztői kártya R301 ellenállását eltávolítottam, a J301-be pedig tüskesort forrasztottam, és a megfelelő két pin-t, vagyis a középső, és a 3V3 feliratú csatlakozást egy jumper-rel összekötöttem. Így a fejlesztői kártyán a táp 3.3V.

Az áramkört úgy terveztem, hogy a csatlakozó a fejlesztői kártya általános célra fenntartott csatlakozóira kapcsolódik. (Az alábbi ábrán a bekeretezett csatlakozók)

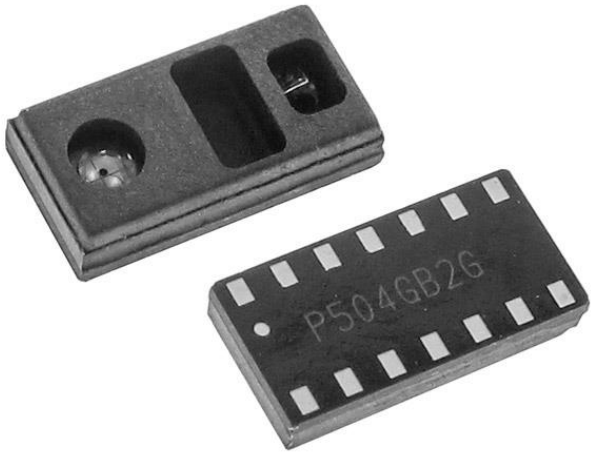
Ezekbe a csatlakozókba hüvelysort forrasztottam. Ezeket a csatlakozókat összekötöttem a megfelelő port-csatlakozókkal. A csatlakozás mindig abban a „sorban” történik, amelyik porthoz majd hozzá kell forrasztani.



13. ábra - Csatlakozások helye a fejlesztői kártyán

2.2.3 Mérő-chip

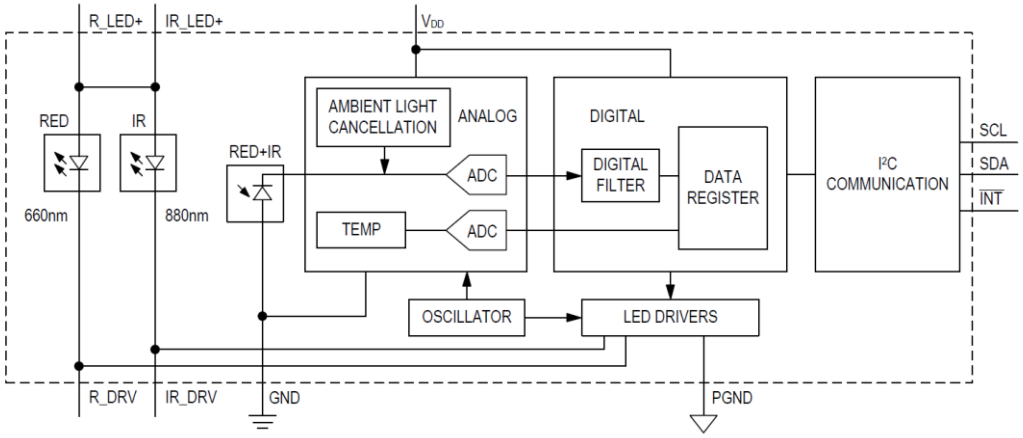
A projekthez kiválasztott mérő-chip a Maxim Integrated Products MAX30100 nevű integrált áramköre. Az eszköz teljes mértékben megfelel a feladatra, de főleg azért lett kiválasztva, mert a konzulensemnek ez állt rendelkezésére. Ez egy direkt pulzoximetriára és szívritmus mérésre alkalmazható áramkör.



14. ábra - A MAX30100 mérő-chip

Áramfelvétele kicsi, vezérlése I2C interfészen keresztül történik. Az eszköz kezeléséről az adatlapjából[3] tájékozódtam, amit az internetről töltöttem le.

A chip alapját egy szenzor alkotja. Ez a szenzor tartalmaz egy fotodiódát, és két LED-et, egy vöröset, és egy infravöröset. Tartalmaz továbbá egy analóg jelfeldolgozó áramkört, ami a szívritmussal arányos igen gyenge jelet erősíti, és leszűri. Az áramkör úgy van kialakítva, hogy méri az ambiens fényt is, vagyis azt, ami akkor szűrődik be, amikor épp egyik LED sem ég, és ezt visszacsatolva levonja a mért eredményből. Saját oszcillátorral rendelkezik, így nem kell külön órajelet biztosítani neki. A LED-ek meghajtása, vezérlése is automatikus, ezt sem a mikrokontrollerrel kell vezérelni. Az eszköz funkcinális blokkdiagramja az alábbi ábrán látható:



15. ábra - A MAX30100 blokkdiagramja

Az eszköz a magas fokú automatika mellett azonban jelentős mértékben programozható, beállítható. Az egyes egységek vezérlésére regisztereket tartalmaz, amelyekkel szabályozható a folyamat. Beállítható, hogy a chip pulzoximetria, vagy csak szívritmus mérés módban működjön. Az eszköz energiatakarékos módba váltható. Beállítható a szenzor mintavételi frekvenciája, a LED-ek pulzusszélessége, árama.

Pulzoximetria módban fontos figyelni a hőmérsékletet, mert ez befolyásolhatja a mérési eredményt. Ennek megkönnyítésére a chip-ben integrált hőmérséklet szenzor is található, amelynek adatait egy regiszterből ki lehet olvasni, és maga a hőmérséklet mérés is programozható. Jelen alkalmazásban ezzel nem kellett foglalkoznom, mert (ahogy az adatlapban is leírják) a pulzusmérést nem befolyásolja jelentősen a hőmérséklet.

A mérési adatok egy FIFO tárolóban tárolódnak el, amit az író és olvasó pointert tartalmazó regiszterek elérésével tudunk vezérelni.

Az eszköz I2C interfészen keresztül képes kommunikálni a mikrokontrollerrel. A regisztereket ezen keresztül írhatjuk, és olvashatjuk. A mérő-chip csak slave módban képes működni, a master ebben az esetben a mikrokontroller. Mivel slave üzemmódban működik, nem képes kommunikációt kezdeményezni. Vannak azonban események, amelyek azonnali kiszolgálást igényelnek, például az, ha a FIFO majdnem tele van, vagy ha kész van egy új adat. Erre a célra a chip rendelkezik egy interrupt lábbal, amelyet ilyen esetekben 0-ba képes húzni. Ez egy olyan vezetékre kell, hogy csatlakozzon, ami valamilyen ellenálláson keresztül tápfeszültségre van kötve. Interrupt kérés esetén ezt a lábat a chip logikai nulla potenciálra képes húzni. Tekintve, hogy a chip lába nem alkalmas nagy áramok elnyelésére, és a tervezés során a takarékosság is szempont, ezt az ellenállást nagy értékűre célszerű választani, például 4.7 kΩ megfelelő. A mikrokontroller external interrupt lábára kötve tehát megszakítás kérhető. Ekkor a mikrokontroller kiolvassa az interrupt status regisztert, és ez alapján eldönti, mit kell reagálni.

Az I2C interfész maximális sebessége 400 kHz. Ebben a projektben ezt a sebességet használtam az I2C kommunikációra, mert a minták elég gyorsan készülnek, és a memóriába írás is I2C-n keresztül történik, így szükség lehet a nagy sebességre a kommunikáció zavartalan működéséhez. Ráadásul, amikor a felhasználó lekérdezi a memória eltárolt adatait, a nagy adatmennyiség miatt a lassabb órajel jelentős különbséget okozna a lekérdezés idejében. Az I2C órajelét a master, jelen esetben a mikrokontroller biztosítja. Ez az órajel csak az I2C interfész, és nem az egész áramkör órajele. A chip I2C

órajel lábát tehát a mikrokontroller I2C lábára kell csatlakoztatni, az adat lábat pedig a mikrokontroller I2C adat lábára.

Az órajel és az adat buszokat az I2C működéséből adódóan egy-egy felhúzó ellenállással tápfeszültségre kell kötni, ezek nélkül a kommunikáció nem mehet végbe. Erre a célra nagy ellenállásokat (jelen esetben 4.7 kΩ) érdemes alkalmazni. A fizikai megvalósításban ez a két felhúzó ellenállás a fejlesztői kártyán van beforrasztva. Az egyes eszközök a buszt nulla potenciálra képesek húzni, ez adja a kommunikáció alapját.

A MAX30100 esetében a konfigurációs és státuszregiszterek írása és olvasása egyértelmű, nem különbözik az I2C protokolltól. A regiszterek 1 bájton címezhetők, és adatuk is 1 bájt. Az I2C olvasás automatikusan inkrementálja a regiszter pointert, így egy regisztercím megadása után több bájt olvasása vagy írása esetén végigmegyünk a regisztereken. Ez egyedül a FIFO adatregiszterre nem igaz. Ha erről a címről olvasunk több bájtot, minden alkalommal ezt a regisztert olvassuk. Az adat azonban változik, mert a MAX30100 lépteti a pointert a FIFO-ban. Egy adat kiolvasásához 4 adatbájtot kell kiolvasni: 2 bájtban az infra LED adatait, majd 2 bájtban a vörös LED adatait. Ez utóbbi egyszerű szívritmus mérésnél mindig 0, de ennek ellenére ki kell olvasni.

A MAX30100 működéséhez két különböző tápfeszültséget használ. az egyik a chip általános tápfeszültsége, ami 1.8V, a másik pedig a LED-eket meghajtó feszültség, ami 3.3V-ot igényel. Ez utóbbi biztosítva van a fejlesztői kártyán. Az 1.8V-os feszültséget viszont elő kell állítani. A deszkamodellhez beszereztem a Lomex boltjában kapható, AP1084K18L-13 típusú 1.8V-os feszültségszabályzót[4]. A szabályzó bemenetére a 3.3V-os tápfeszültséget kell kapcsolni, kimenetére pedig a mérő-chip 1.8V-os bemenetét. A fenti a feszültségszabályzó egyébként viszonylag nagy, 5A-es áram kibocsátására képes. Ez a jelen alkalmazásban nemhogy elég, de fölöslegesen nagynak mondható. Ennek oka a következő: Az 1.8V-os tápfeszültség igen ritka a digitális áramköröknél, így aztán az ilyen szabályzó beszerzése nagyobb utánajárást igényel. A Lomex boltjában például ez volt az egyetlen 1.8V-os szabályzó. A kevés időre való tekintettel, és figyelembe véve azt, hogy az elkészített eszköz nem végleges verzió lesz, és nem hordozható, hanem a számítógép által biztosított tápról működik, úgy döntöttem, megfelel a projektbe az említett alkatrész. A végleges verzióhoz kisebb áramú szabályzó javasolt, például a MAX30100-hoz gyártott fejlesztői kártyán[16] is használt AP7331-ADJ.

A MAX30100 adatlapjában azt írják, hogy a LED-ekhez szükséges 3.3V-os tápfeszültségnek célszerű később megjelennie, mint az 1.8V-os feszültségnek. Ehhez arra van szükség, hogy valamilyen módon figyeljük az 1.8V-os szabályzó kimenetén lévő potenciált. A projektben ezt úgy oldottam meg, hogy egyszerűen összekötöttem a mikrokontroller egyik ADC bemenetével, és szoftveresen figyelem a jel értékét. Ha eléri a kívánt mennyiséget, egy MOS-FET segítségével rákapcsolom a chip-re a 3.3V-os feszültséget is.

2.2.4 Memória

Az adatok eltárolásához szükség volt valamilyen memória egységre. Mért adatok alatt az egyes szívverések közt eltelt időt értem. Fontos szempont volt, hogy a memória a tápfeszültség kimaradása esetén is megtartsa a tartalmát. Megfelelő választás lehet tehát valamilyen egyszerű EEPROM memória. El kellett dönteni, hogy mekkora tárhelyre van szükség az adatok eltárolásához. Ehhez a következő megfontolásokat tettem: Az eszközt úgy terveztem, hogy legalább egy teljes nap, tehát 24 óra adatait legyen képes eltárolni. Ha (biztonságosan fölülről becsülve) úgy számolunk, hogy a felhasználónak egész nap 120-as pulzusa van, akkor: $120\,\frac{adat}{perc} * 60\,\frac{perc}{óra} * 24\,\frac{óra}{nap} = \mathbf{172800}\,\frac{\boldsymbol{adat}}{\boldsymbol{nap}}$

Tehát naponta 172800 adatot kell eltárolni. Kérdés, hogy hány bájton fér el egy adat? Ehhez el kell dönteni, hogy milyen pontosságra van szükség a pulzusméréshez. Ha például egy 75-ös pulzust vizsgálunk, két szívverés közt $\frac{60s}{75} = 0{,}8s$ telik el, 74-es pulzus esetén pedig $\frac{60s}{74} \cong 0{,}811s$. Ahhoz, hogy körülbelül ilyen szívritmus esetén a pulzust egyetlen mérési eredményből egész pulzusszámra pontosan meg tudjuk mondani, 10ms-os pontosság tehát elég. A projekthez ezt a pontosságot választottam. 120-as pulzusnál ebből ugyan adódhat egy legfeljebb ±2 értékű eltérés, de ez több okból nem lényeges: Egyrészt a feladat elsősorban a szív túl hosszú kimaradása, esetleg szívritmus zavarok detektálására készült, a valószínűtlenül magas szívritmus értékek mérésének pontossága kevésbé fontos. Másrészt ez a ±2 eltérés csak az egyetlen időértékből számolt pulzus érték. Tényleges pulzusméréshez több szívverésre vonatkozó idők alapján szoktak számolni, úgy pedig ez a pontatlanság elenyésző.

Tehát a pontosság 10ms. A tárolt értékek praktikusan a két szívverés közt eltelt időt jelentik, 10ms-ban. Egy hosszabb, például 3 másodperces szünet van a szívritmusban, akkor a memóriába a 300 értéket kell beírni. Már ez sem fér el 1 bájton,

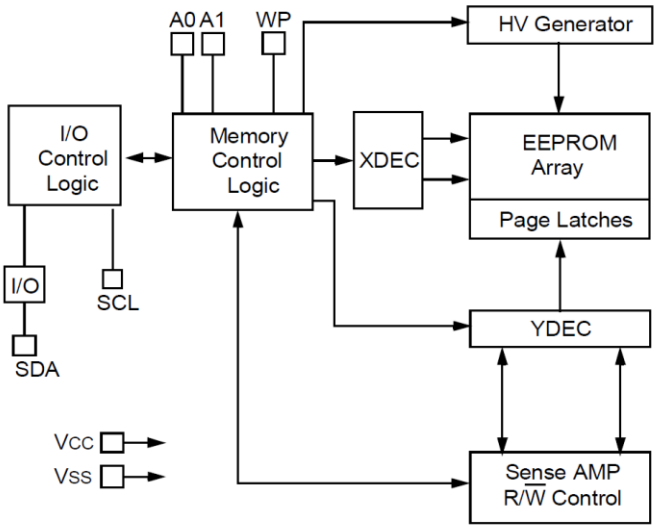
úgyhogy 2 bájtra mindenképp szükség van egy adat tárolásához. A 2 bájton tárolható leghosszabb idő $(2^{16} - 1) * 0.01s = 655{,}35s$, úgyhogy 2 bájt biztosan elég.

Visszatérve a memória méretéhez, egy napi adat eltárolásához szükséges tárhely: $172800 * 2byte = 345600byte = 2764800bit = \mathbf{2700kbit}$.

Ez három darab 1024kbit-es EEPROM-mal megvalósítható. Mivel ez elég nagy, mindenképpen valamilyen soros kommunikációra képes memóriákra van szükség. A legpraktikusabb az lenne, ha az EEPROM-ok is I2C interfészen keresztül lennének elérhetőek, hisz akkor a szoftverben az I2C kezelő függvényekkel mind a mérő-chip vezérlését, mind a memória kezelést meg tudnánk oldani, így egyszerűsödne a kód. Mivel a memóriába csak szívverésenként, vagyis legrosszabb esetben 0.5 másodpercenként kell írni, az I2C busz terheltségét elenyésző mértékben növeljük, ezzel a problémával nem kell foglalkozni.

Olyan EEPROM-ot kell tehát találni, ami 1024kbit tárolására alkalmas, és I2C kommunikációra képes, 400kHz órajel mellett. A választott memória típusa: 24FC1025.[5] Ez egy bájtos szervezésű I2C soros EEPROM, ami 1.8V-5.5V tápfeszültséggel működik, tehát a 3.3V megfelelő. 2.5V felett 1MHz-es órajellel is képes működni, így a 400kHz sem lesz akadály. Három darab „slave address” lába van, amelyekből az egyiket mindenképp tápfeszültségre kell kötni, de a másik kettő a saját címének beállítására szolgál. Mivel 2 biten 4 különböző kombinációt tudunk kódolni, ezért összesen 4 ilyen memória működtethető ugyanazon az I2C buszon, ami megfelel, hisz 3-ra van szükség. A memória két darab 512 bites blokkból áll. Az I2C üzenetben, amiben a slave címét küldjük ki, a harmadik slave cím bit helyett a blokk számát kell küldeni.

A memória blokkvázlata:



16. ábra - 24FC1025 soros EEPROM blokkvázlata

A projektben a három memória slave cím lábait tápra illetve földre kötéssel 00, 01 és 10 állásba kötöttem. Mindhárom memória I2C adat lábát összekötöttem az I2C adat vezetékkel, az I2C órajel lábát pedig az órajel vezetékkel. A memóriáknak írás-védelem lába is van. Ezeket a lábakat a mikrokontroller egy-egy portlábához csatlakoztattam, hogy legyen lehetőség azokat szoftveresen vezérelni.

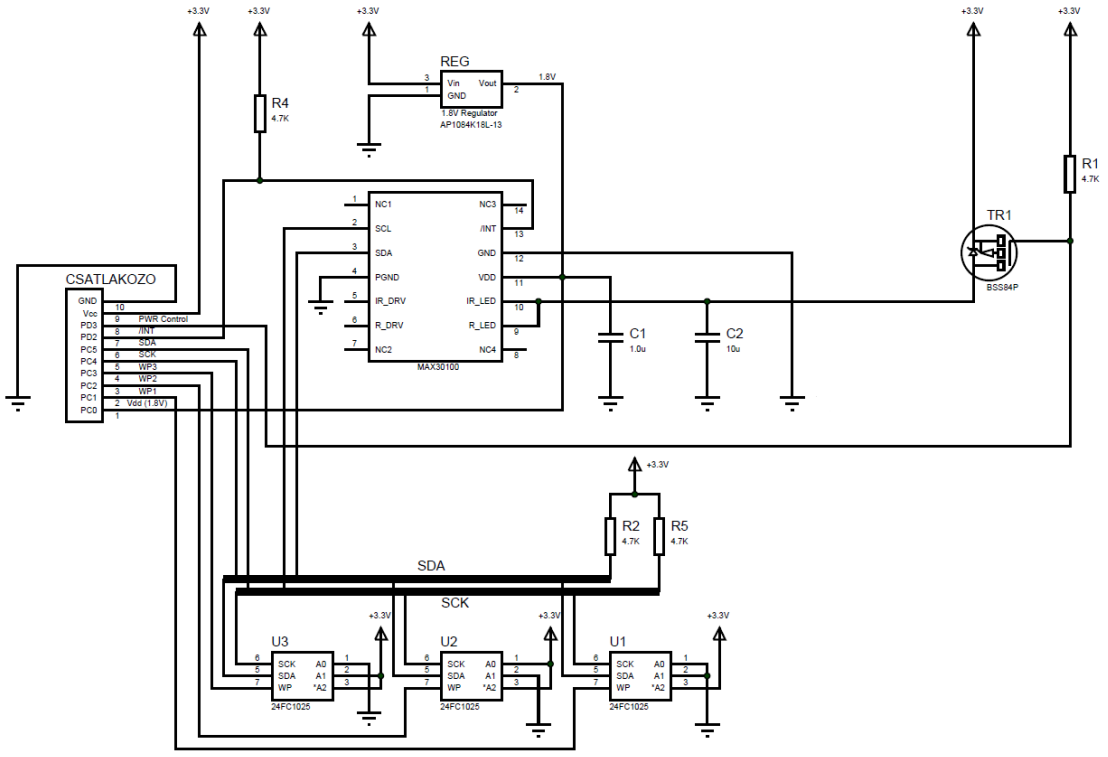
A memória kezelése egyszerű, I2C szabvány alapján működik, eltekintve a fenti kis különbségtől. A cím 2 byte hosszan küldendő a memóriának, majd tetszőleges mennyiségű adat olvasható egy blokkon belül. Az áramkör működéséről és használatáról bővebben az adatlapjából[5]. tájékozódtam.

2.2.5 A deszkamodell kapcsolási rajza

Miután ki lettek választva a megfelelő alkatrészek, meg kellett tervezni az eszköz konkrét kapcsolását. A kapcsolási rajzot a Proteus 8 nevű programmal készítettem. Ez egy fejlett, elsősorban mikrokontrolleres rendszerek tervezésére és szimulálására alkalmas szoftver. Lehet benne kapcsolási rajzot létrehozni, a tervben szereplő mikrokontrollerre programot írni, és az így létrehozott rendszert szimulálni, valamint nyomtatott áramkör tervező funkcióval is el van látva. A program kezelését főként youtube videók, és internetes fórumok alapján tanultam meg.

Több alkatrész, mint például a memóriák, alapból benne voltak a program adatbázisában. Azokat az alkatrészeket, amik nem szerepeltek benne, nekem kellett létrehozni.

A projekt kapcsolási rajza az alábbi ábrán látható:



17. ábra - Teljes kapcsolási rajz

Látható a MAX30100-as mérő-chip (MAX), a három darab 24FC1025 típusú I2C soros EEPROM (U1, U2 és U3), és az 1.8V feszültséget előállító AP1084K18L-13 típusú feszültségszabályzó (REG). Ezen kívül létrehoztam egy külön elemet CSATLAKOZO néven, ami a mikrokontrollerhez való csatlakozást jelképezi, és a nyomtatott áramkör tervezésnél saját footprint tartozik hozzá, a fejlesztői kártya port-jainak elhelyezkedése szerint.

Látható még a kapcsolási rajzon két kapacitás, egy 10µF-os (C2) és egy 1µF-os (C1). Ezek zajszűrésre szolgálnak, méretezésük pedig a MAX30100 adatlapjában[3] javasolt értékek alapján történt.

A MAX30100 adatlapjában[3] azt javasolják, hogy a 3.3V-os bemeneten csak akkor jelenjen meg a feszültség, ha az 1.8V-os bemeneten már jelen van. Ennek megvalósítására a 3.3V-os bemenetre egy MOS-FET kapcsolódik, ami úgy van bekötve,

hogy, ha gate-jére logikai nulla potenciál kerül, akkor a mérő-chip bemenetére 3.3V kerül, különben a földre kötött 4.7kΩ-os ellenállás (R3) miatt földpotenciálon marad. A tranzisztor gate-je pedig a mikrokontroller egyik portjára van csatlakoztatva („PWR Control” feliratú vezeték).

A „Vdd(1.8V)” feliratú vezeték az 1.8V-os bemenethez, és a mikrokontroller analóg-digitál átalakító bemenetére csatlakozik. Így a mikrokontroller szoftveresen tudja figyelni az 1.8V-os bemeneten lévő feszültségszintet.

A MAX30100 /INT kimenete egy 4.7kΩ-os felhúzó ellenállással tápfeszültségre van kapcsolva. Ez a vezeték a mikrokontroller egyik külső megszakítás-kezelő lábára csatlakozik. A chip földre tudja húzni a vonalat, hogy jelezzen a mikrokontrollernek.

A három memória ugyanúgy van bekötve, eltekintve a slave címüket beállító bemenetektől (A1, A0). Látszik, hogy ezek mindhárom esetben más kombinációt alkotnak. Az írás-védelem bemenetek (WP) mindhárom esetben egy-egy mikrokontroller portlábhoz kapcsolódnak.

Az I2C órajel és adat vezetékekhez az összes I2C interfészt használó eszköz megfelelő lába csatlakoztatva van. Az ezek működéséhez szükséges 4.7kΩ-os felhúzó ellenállások a fejlesztői kártyán vannak bekötve.

A mérő-chip többi lábát nem kellett csatlakoztatni semmihez. A NC jelzésű lábak csak a rögzítést szolgálják, az IR_DRV és R_DRV lábakat pedig az adatlap szerint lebegve kell hagyni.

2.2.6 Végleges terv

A fenti kapcsolási rajz egy deszkamodellnek megfelelő, azonban a végleges tervnél fontos a hordozhatóság, és az energiatakarékosság. Az áramkör nem fejlesztői kártyával, hanem egyszerű mikrokontrollerrel tervezendő, az UART vonalhoz csatlakozással, és a tápellátáshoz szükséges kiegészítő áramkörrel.

2.2.6.1 Tápellátás

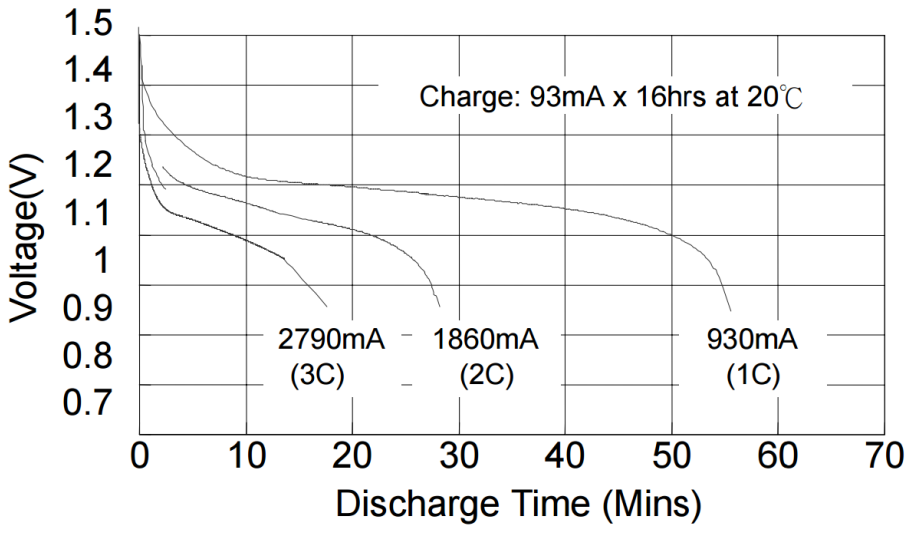
Az eszköz tápellátását külső akkumulátor adja. A tápfeszültségnek állandóan 3.3V-on kell maradni az eszköz helyes működéséhez, ezért szükség van valamilyen feszültségszabályzóra, hogy az elem alacsonyabb feszültségéből állandó potenciált hozzon létre, még az elem merülése esetén is. Erre a célra a Texas Instruments TPS61097A-33[17] step-up konverterét néztem ki. Használatára vonatkozóan az

adatlapjában sok segítség található. Nagyon alacsony, µA nagyságrendű áramfogyasztása van, maximális kimeneti árama pedig 400 mA.

Az akkumulátornak olyan kapacitással kell rendelkezni, hogy legalább 24 óráig működhessen az eszköz töltés, vagy csere nélkül. Ehhez becslést számoltam a teljes eszköz **maximális** áramfogyasztására:

- MAX30100 mérő-chip: 20.2 mA
- ATmega328P mikrokontroller: 10mA
- AP7331 szabályzó: 1mA
- 3 db 24Fc1025 EEPROM: 2mA
- Az áramkörben levő ellenállások: 2mA

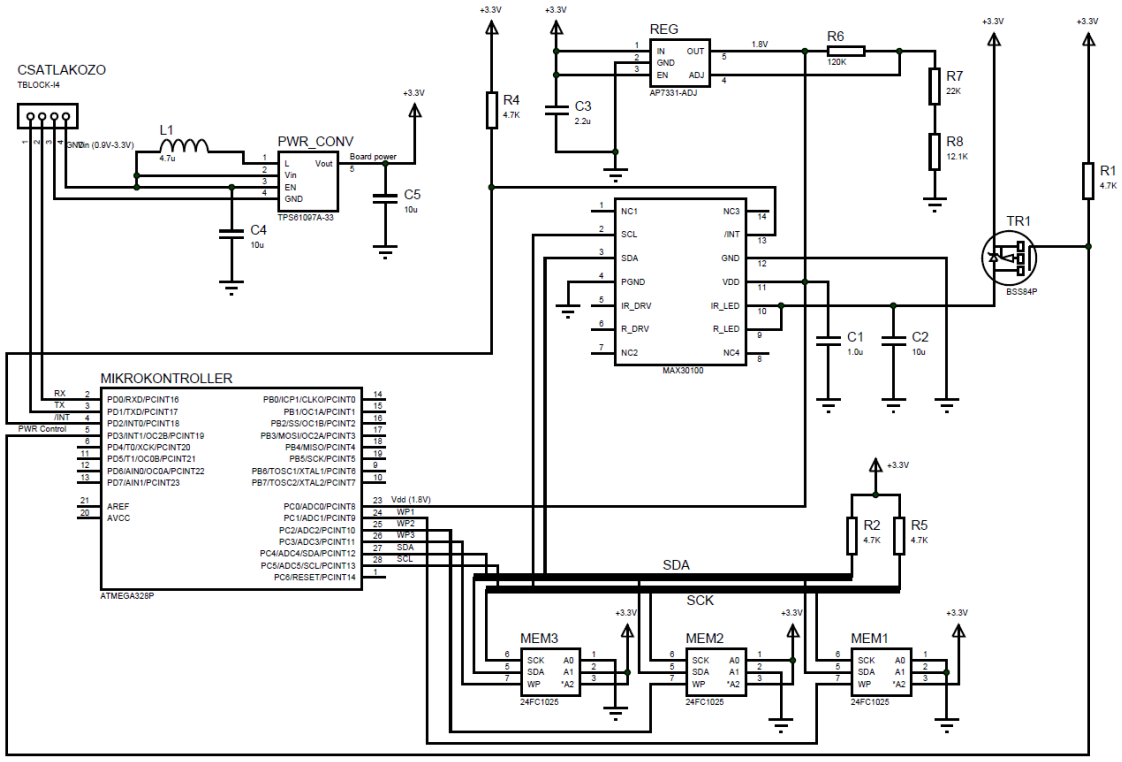
A teljes eszköz fogyasztása tehát legfeljebb 35,2 mA. Ez egy maximális érték, valójában bőven a tényleges fogyasztás fölött van. Ha azt akarjuk, hogy ilyen fogyasztás mellett 24 óráig működjön a készülék, akkor elvileg legalább $35.2mA * 24h = 844.8mAh$, kapacitású akkumulátorra lenne szükség. A fenti feszültségszabályzó azonban csak minimum 0.9V feszültségből képes 3.3V-ot előállítani, tehát az akkumulátort csak addig szabad lemeríteni, amíg a feszültsége 0.9V alá csökken. Megvizsgáltam a GP100AAAHC[22] nikkel metál hidrid akkumulátor merülési karakterisztikáját. Az azonos típusú akkumulátorok merülési görbéje jellegre hasonló, ezért ebből, és még néhány az interneten talált karakterisztikából általános következtetéseket vontam le a lemeríthetőség becslésére. A karakterisztika az alábbi ábrán látható:



18. ábra - Akkumulátor merülési görbe

930 mA-rel működtetve az akkumulátor elvileg 60 perc alatt merülne le teljesen. Ezt a görbét vizsgálva látható, hogy körülbelül 55 perc után a feszültség 0.9V alá csökken. Az akkumulátort tehát csak kapacitásának $\frac{60-55}{60} = \frac{1}{12}$ részére lehet lemeríteni a megfelelő tápellátás tartásához. 24 órás működéshez tehát $844.8mAh * \frac{12}{11} = 921.6mAh$ kapacitású akkumulátor szükséges.

A kapcsolási rajz az alábbi ábrán látható:



19. ábra - Végleges kapcsolási rajz

2.2.7 Nyomtatott áramkör

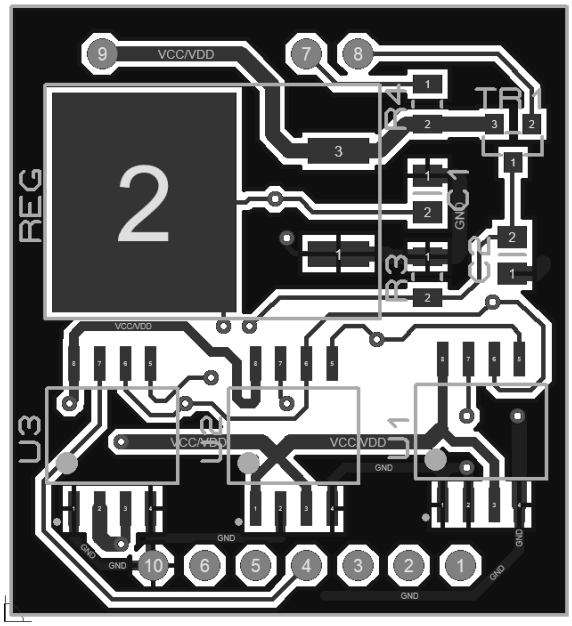
A nyomtatott áramkör megtervezéséhez a már említett Proteus 8 programot használtam. Az adatbázisban nem szereplő alkatrészeknek meg kellett alkotni a footprint-jét. Ezeket az alkatrészek adatlapjai alapján szerkesztettem meg. A csatlakozó footprint-jét pedig úgy, hogy a nyomtatott áramkör épp illeszkedjen az ATmega328P Xplained Mini fejlesztői kártya megfelelő helyre beforrasztott hüvelysoraiba. A MAX30100 footprint-jének tervezésénél a pad-eket jóval nagyobbra terveztem, mint amit az adatlapban javasoltak, mert olyan apró pad-ekre kézi forrasztással nem lehetne megfelelően beforrasztani a chip-et.

A kapcsolási rajz közvetlenül átvihető volt a program nyomtatott áramkör tervező módjába. Ott fel voltak sorolva a kapcsolási rajzon szereplő alkatrészek, amiket el kellett helyezni a nyomtatott áramköri terven. A program ekkor vonalakkal jelezte, hogy mely pad-eknek kell összekötve lenniük. Az alkatrészeket úgy helyeztem el, hogy ezek a vonalak a lehető legkevésbé keresztezzék egymást, ezzel egyszerűsítve az alkatrészek összekötését. A mérő-chipet arra az oldalra helyeztem, amelyik a fejlesztői kártyára szerelve fölfelé néz, hiszen arra kell tenni az ujjat a pulzusméréshez. Ez az oldal a nyomtatott áramkör terven egyébként a BOTTOM réteg. Az összes többi alkatrész a másik oldalra került. (TOP réteg)

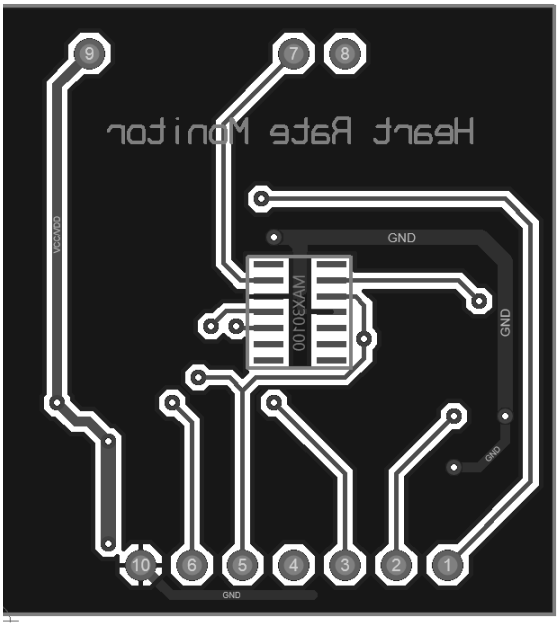
Miután elhelyeztem az alkatrészeket, két lehetőségem volt összekötni a megfelelő pad-eket: Az egyik, hogy kézileg összekötöm őket, a másik, hogy használom a program beépített, intelligens auto-rooting funkcióját, ami automatikusan összeköti a pad-eket egy fejlett algoritmust használva, paraméterezhető szabályokat figyelembe véve, a lehető legoptimálisabban. Ez utóbbi módszert kipróbálva elégedett voltam az eredménnyel, ezért ezt vettem alapul, csak kisebb változtatásokat végeztem utána. Például a kanyarodó vezetékek több helyen túl élesen kanyarodtak, és több helyen lehetett egyszerűsíteni is.

Ezután, szintén a program beépített funkcióját használva, a fennmaradó területeken föld-kiöntést hoztam létre mindkét oldalon. Az alkatrészeket a TOP SILK és a BOTTOM SILK rétegen feliratokkal láttam el, hogy egyértelmű legyen, mit hova kell majd forrasztani. A mérőchip oldalára egy „Heart Rate Monitor” feliratot is terveztem, így könnyebb beazonosítani a nyomtatott áramköri lemezt.

Az alábbi ábrán látható a nyomtatott áramkör TOP és BOTTOM oldalának terve:



21. ábra - TOP oldal



20. ábra - BOTTOM oldal

A földre kötendő pad-eket összekötő GND jelzésű vezeték a földkiöntés ellenére megmaradt, de ez a kész nyomtatott áramkört nem befolyásolja. A BOTTOM ábrán a mintázat tükrözve látható, hiszen ebben a nézetben a TOP oldal felől nézzük a BOTTOM oldalt. Miután kész lett a terv, exportáltam a gyártáshoz szükséges Gerber fájlokat, és leadtam a konzulensemnek, aki továbbította az Elektronikai technológia tanszéknek.

Az alábbi táblázatban összeszedtem, hogy mi az összefüggés a kapcsolási rajzon található jelek, a csatlakozás NYÁK footprint-jének furat-sorszámai, az ATmega328P mikrokontroller jelei, a mikrokontroller portok és az Xplained Mini fejlesztői kártya csatlakozói között:

Kapcsolás jel	**NYÁK csatl.**	**µC jel**	**Port/Pin**	**Kártya csatl.**	**megjegyzés**
SCK	6	SCL	PC5	5 A	I2C órajel (output)
SDA	5	SDA	PC4	5 B	i2C adat (output)
WP3	4	PC3	PC3	5 C	3. Memória write protect (output)
WP2	3	PC2	PC2	5 D	2. Memória write protect (output)
WP1	2	PC1	PC1	5 E	1. Memória write protect (output)
Vdd (1.8V)	1	ADC0	PC0	5 F	1.8V meglétének vizsgálata (input)
/INT	7	INT0	PD2	6 C	Mérő chip interrupt kérése (input)
PWR Control	8	PD3	PD3	6 D	Mérő chip tápjának bekapcsolása (output)
GND	10	GND	GND	GND	A fejlesztői kártya GND csatlakozója
Vcc	9	Vcc	Vcc	VCC	A fejlesztői kártya Vcc csatlakozója

2.3 Szoftver

2.3.1 I2C interfész kezelés

Az I2C interfész kezelése interruptok (megszakítások) segítségével történik. A függvények nagy részben általánosak, de egyes helyeken alkalmazás-specifikusak. Az I2C interfész megfelelője az ATmega328P mikrokontrolleren szerzői jogi ügyek miatt a TWI (Two-Wire-Interface) elnevezést kapta. Ez gyakorlatilag ugyanaz, mint az I2C, és teljesen kompatibilis vele.

A kommunikáció vezérlésére létrehoztam egy TWI_comm_type nevű struktúra típust, ami az I2C üzenetek tulajdonságait tartalmazza.

```
typedef struct
{
   char SLA;   //Slave cím
   long int addr; //cím
   long int data; //Adat, amit írunk, vagy amibe olvasunk
   char A0, A1, B;   //Memória slave cím és blokk kijelölő változók
   char AddressLength;  //A cím bájtjainak száma
   char DataLength;  //Az adat bájtjainak száma
   Read_Action_type Read_Action; //Miből olvasunk?
   device_type device;  //Memóriával, vagy mérő-chippel kommunikálunk?
   R_W_type R_W;  //Azt jelöli, hogy írunk, vagy olvasunk
} TWI_comm_type;
```

Ehhez először létrehoztam egy enum változó típust, ami olvasás esetén azt jelzi, hogy az adott I2C üzenet miből olvas (EEPROM, vagy a mérő-chip valamelyik regisztere). Az olvasás befejeztével ez alapján dönti el a program, hogy mit kell kezdeni a kiolvasott adattal. A regiszterből olvasáshoz tartozó értékeket úgy állítottam be, hogy megegyezzen az adott regiszter címével, így a címből a Read_Action egyszerű értékadással számolható. Létrehoztam még két enum változó típust, amik jelölik, hogy memóriával, vagy mérő-chippel kommunikál az üzenet, illetve azt, hogy írás vagy olvasás.

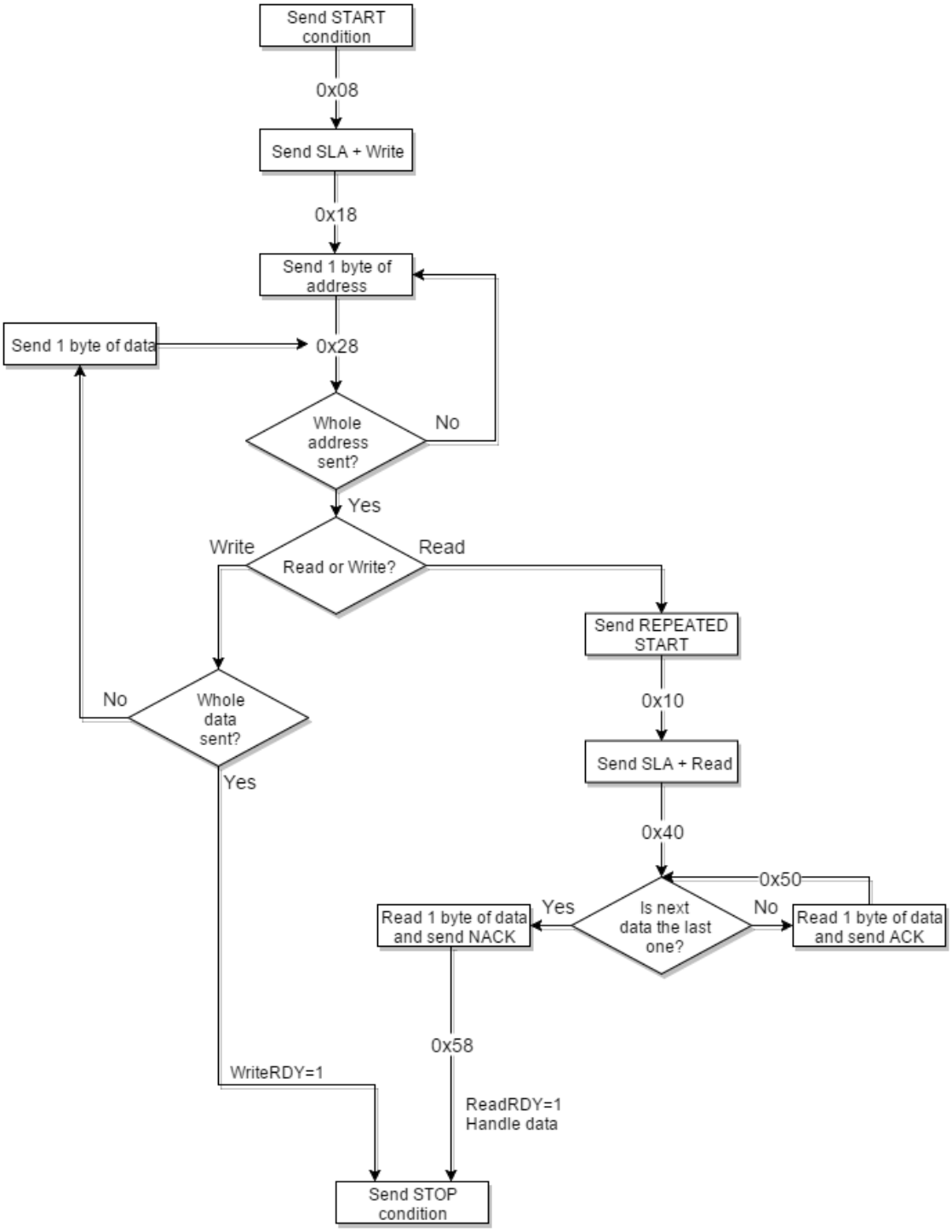
```
typedef enum
{
Read_INT_state=0,
Read_FIFO_wr_ptr=0x2,
Read_OVF_cnt=0x3,
Read_FIFO_rd_ptr=0x4,
Read_FIFO_dat=0x5,
Read_EEPROM=0xAA
} Read_Action_type;
typedef enum {MEM, MAX} device_type; //Melyik eszköz?
typedef enum {WRITE = 0, READ=1} R_W_type; //Olvasás vagy írás?
```

Egy darab alkalmazásból hívható függvény van az I2C kommunikáció indítására. Ez a függvény a *TWI_start*. Célja, hogy létrehozzon egy új TWI_comm_type változót,

beállítsa a szükséges paramétereit, és üres busz esetén elindítsa az I2C kommunikáció folyamatát, amely utána megszakítások által vezérelt. Ha a busz foglalt, beírja a létrehozott TWI_comm_type változót egy ilyen változókból álló ciklikus pufferbe. A busz felszabadulása esetén a megszakítás kezelő függvény majd kiolvassa a soron következő üzenetet a pufferből, és elindítja a küldést. A függvény visszatérése nem jelenti azt, hogy a kommunikáció le is zajlott, csak azt, hogy elindult, vagy el fog indulni.

A kommunikáció a következőképp zajlik: Kiküldünk valamilyen üzenetet, és várunk, hogy befejeződjön a küldés. Eközben haladhat a főprogram. A hardver egy interrupt kéréssel jelzi, hogy kész van az adott művelet, el kellene dönteni, hogy most mit küldünk. A mikrokontroller erre kijelölt regiszterében diagnosztikai kódokat találunk, ami alapján megállapíthatjuk, hogy milyen művelet lett most épp kész, és milyen reakció érkezett. Ezután különböző változók alapján meghatározzuk a következő lépést, és elindítjuk.

A kommunikációt vezérlő szoftver folyamatábrája az alábbi képen látható. A nyilakra írt számok jelzik a kiolvasott diagnosztikai kódot hexadecimálisan. A téglalapok végrehajtandó műveleteket, a rombuszok pedig döntést jelentenek.



22. ábra - Az I2C kezelés folyamatábrája

Ez alapján már csak implementálni kellett a fenti automatát. Szükség volt egy függvényre, ami beállítja a működéshez szükséges változókat, és elindítja a folyamatot, egy inicializáló függvényre, az interrupt kiszolgálás függvényére, és egy függvényre, ami kezeli a kiolvasott adatot.

2.3.1.1 TWI_init()

A használat előtt inicializálni kell az I2C interfészt. Erre szolgál a TWI_init() függvény. Lényegében annyit csinál, hogy a mikrokontroller megfelelő regisztereit beállítja. A regiszterek bitjeinek jelentését az ATmega328P adatlapjában[1] olvastam.

A függvény a következőképp néz ki:

```
void TWI_init()
{
   TWBR = 0x2;    //Bit Rate Register
   TWSR = 0;   //Utolsó két bit: PrescalerValue : 1
}
```

A TWBR Bit Rate regiszter értékét a mikrokontroller adatlapjában található képlet alapján határoztam meg: $SCL\ frequency = \frac{CPU\ Clock\ frequency}{16+2(TWBR)*(PrescalerValue)}$

Ez alapján a 400kHz-es I2C órajelhez, 1 értékű PrescalerValue esetén, 8MHz-es rendszer-órajel mellett 2 értéket kell beállítani.

2.3.1.2 TWI_buff_rd és TWI_buff_wr függvények

A végrehajtandó I2C kommunikációk egy ciklikus pufferben tárolódnak:

```
#define BUFFSIZE 100    //A puffer mérete
TWI_comm_type TWI_buffer[BUFFSIZE];
```

Ez a két függvény ebből a pufferből való olvasást, illetve a pufferbe írást valósítják meg:

```
void TWI_buff_wr(TWI_comm_type TWI_comm_in)  //beírja az adatot a pufferbe
{                              //(Ha a buffer tele van, felülírja a legrégebbi adatot)
   TWI_buffer[TWI_wr]=TWI_comm_in;  //Adat beírása

   //Ha utolértük az olvasó indexet, de nem azért, mert üres, akkor a puffer tele van,
   if(TWI_wr==TWI_rd && TWI_isempty==0)
      TWI_rd=(TWI_rd+1)%BUFFSIZE;// léptetjük az olvasó indexet.

   TWI_wr=(TWI_wr+1)%BUFFSIZE;//Léptetjük az író indexet
   TWI_isempty=0;  //Mostmár biztosan nem üres.
}

char TWI_buff_rd(TWI_comm_type* TWI_comm_out)//Kiolvas egy adatot a pufferből
{
   if(TWI_isempty)//Ha üres a puffer, nem tudunk olvasni
      return 0;

   *TWI_comm_out=TWI_buffer[TWI_rd];//adat kiolvasása
   TWI_rd=(TWI_rd+1)%BUFFSIZE;// Léptetjük az olvasó indexet


   if(TWI_rd==TWI_wr)//Ha utolértük az író indexet, akkor a puffer üres lett.
      TWI_isempty=1;
   return 1;
}
```

2.3.1.3 TWI_Start() függvény

A függvény a megadott adatok alapján paraméterez egy új I2C kommunikációt. Üres busz esetén START feltétel kiküldésével megkezdi az I2C kommunikációt, míg foglalt busz esetén elmenti egy pufferbe a struktúrát. A kód megértéséhez feltüntetem a függvényben szereplő változókat és definíciókat, és elmagyarázom a hasznukat.

Definiáltam a két használandó eszköz slave címét a kód áttekinthetősége érdekében. A memóriák címének utolsó három bitje azonban nem fix, aszerint változik, hogy melyik memóriát, és annak melyik blokkját akarom elérni. Ezeket egyelőre 0-ba állítottam.

```
#define MEM_ADDRESS 0b10100000
#define MAX_ADDRESS 0b10101110
```

Az alábbi változókra a folyamat lezajlásához van szükség. Ha memóriát címzünk, a slave címbe (SLA) be kell írni a blokk bitet (B), és a hardveresen definiált slave cím konfigurációt (A0, A1). Ezeket a függvény majd a memóriacím felső három bitjeként kapja meg. A TWI_RDY változókkal követjük, hogy épp szabad kommunikációt kezdeményezni, vagy az előző kommunikáció még zajlik.

```
char A0, A1, B;    //blokk kijelölő bit, és a slave address kiegészítése
char AddressByte_Sent;  //Globális változó. A már elküldött cím bájtok száma
char DataByte_Sent;  // Globális változó. A már kiolvasott, vagy elküldött adatbájtok száma
TWI_comm_type ActualTWI;// Globális változó. Aktuálisan zajló I2C művelet
char TWI_RDY=1;// Globális változó. Szabad-e az I2C busz?
```

A TWI_RDY változó figyelésével vizsgáljuk, hogy rögtön elindíthatjuk-e a kommunikációt, vagy csak vegyük fel a pufferbe. Ha azonban a vizsgálat után, de még a pufferbe írás közben, vagy a TWI_RDY változó 0-ba állítása előtt egy megszakítás érkezik, amelyik új kommunikációt kezdeményez, annak nem várt következményei lehetnek. Ennek elkerülésére a vizsgálat előtt letiltom a megszakításokat, és csak a végrehajtás után engedélyezem újra.

A TWI_Start() függvény-t a következőképp implementáltam:

```
void TWI_start(R_W_type R_W_in, char Slave_address, long int address, int length_of_data,
long int data_to_send)
{
   TWI_comm_type NewTWI;
   char A0, A1, B;//Memória slave cím és blokk kijelölő változók

   //Meghatározzuk, hogy melyik eszközre írunk.
   if(Slave_address == MEM_ADDRESS)
      NewTWI.device = MEM;
   else if(Slave_address == MAX_ADDRESS)
      NewTWI.device = MAX;
```

```
if(NewTWI.device == MEM)//Ha memória
   {
      NewTWI.Read_Action=Read_EEPROM;   //Jelezzük, hogy a memóriából olvasunk

      //A blokk bit, és a slave address kiválasztó
      //bitek a cím meghosszabbításaként szerepelnek
      B  = (address>>16) & 1;
      A0 = (address>>17) & 1;
      A1 = (address>>18) & 1;
      NewTWI.AddressLength=2; //A memóriánál a cím hossza 2 byte
      NewTWI.addr = address & 0xFFFF;  //Beállítjuk a címet tartalmazó globális változót
   }
   else if(NewTWI.device == MAX) //Ha a MAX30100-ról olvasunk
   {
      //A megcímzett regiszter címe alapján tudjuk, hogy melyik regiszterről olvasunk.
      NewTWI.Read_Action = address;
      B = A0 = A1 = 0;  //Ezekre most nincs szükség
      NewTWI.addr = address & 0xFF; //Beállítjuk a címet tartalmazó globális változót
      NewTWI.AddressLength=1; //MAX30100 esetén a cím hossza 1 byte
   }
   NewTWI.SLA = Slave_address | (B<<3) | (A1<<2) | (A0<<1);//Létrehozzuk a slave address-t

   //Beállítjuk az adat hosszát jelölő globális változót
   NewTWI.DataLength = length_of_data;

   if((NewTWI.R_W = R_W_in)==READ)
      NewTWI.data=0;
   else
      NewTWI.data=data_to_send;

   cli();
   if(!TWI_RDY)
   {
      TWI_buff_wr(NewTWI);
   }
   else
   {
      TWI_RDY=0;
      //Kinullázzuk a szükséges változókat
      DataByte_Sent = 0;
      AddressByte_Sent = 0;

      ActualTWI = NewTWI;

      TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)|(1<<TWIE);   //Kiküldünk egy Start feltételt.
   }
   sei();
}
```

2.3.1.4 StateHandler függvények

Ezek a függvények az I2C interrupt kiszolgáló rutinban hívódnak meg, és az adott állapotokban elvégzendő műveleteket tartalmazzák. A megszakítás-kezelés a jobb áttekinthetőség végett lett szétszedve ezekre a függvényekre. Implementálásuk a fenti folyamatábra alapján történt. A függvénynevekben a „StateHandler” karaktersorozat után az adott állapot kódja szerepel.

A függvények megértéséhez ismerni kell az ATmega328P ide tartozó regiszterei, és a bennük lévő bitek elnevezését:

A TWDR regiszter az adatregiszter. Írás esetén ebbe kell rakni az adatot, olvasás esetén innen olvasható ki.

A TWCR regiszter az interfész vezérlésére szolgál. TWINT és TWIE bitjeit mindegyik műveletnél 1-be kell állítani, előbbivel tisztázzuk a megszakítás flag-et, utóbbival engedélyezzük a megszakításokat. A TWEN bit engedélyezi az I2C működését. A TWSTA és a TWSTO bitek start és a stop feltételeket küldenek ki. A TWEA bit azt határozza meg, hogy a következő adat-érkezés esetén a mikrokontroller az acknowledge bitet behúzza nullába (ACK), vagy hagyja magas szinten (NACK), jelezve hogy nem vár több adatot.

A függvényekben használt definíciók és változók jelentését már korábban leírtam. Érdemes még megemlíteni, hogy az R_W nevű enum változó két állapota, ami azt jelzi, hogy írás, vagy olvasás zajlik, úgy van definiálva, hogy a slave címmel bitenkénti „*vagy”* kapcsolatba hozva épp az adott művelet esetén kiküldendő üzenetet kapjuk.

A StateHandler_0x58() függvény végén meghívott DataHandler() függvény egy alkalmazás-specifikus függvény a mérő-chip kezelésére, ami eldönti, hogy mit csináljon a beérkezett adattal. Működését később részletezem.

Ezek alapján már értelmezhető az alábbi, StateHandler függvényeket implementáló kódrészlet:

```
void StateHandler_0x08() /*A START condition has been transmitted*/
{
   AddressByte_Sent = DataByte_Sent = 0;

   //Kiküldjük a slave címét, egyben azt is jelezve, hogy írni szeretnénk
   TWDR = ActualTWI.SLA | WRITE;
   TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE); //elindítjuk a küldést
}

void StateHandler_0x18() /*SLA+W has been transmitted; ACK has been received*/
{
   AddressByte_Sent = 1;

   //a cím 1 bájtjának küldése (legfölső)
   TWDR = (ActualTWI.addr >> (8*(ActualTWI.AddressLength-AddressByte_Sent))) & 0xFF;

   TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE); //elindítjuk a küldést
}

void StateHandler_0x28() /*Data byte has been transmitted; ACK has been received*/
{
   if(AddressByte_Sent < ActualTWI.AddressLength)   //Ha van még kiküldendő cím bájt
   {
      AddressByte_Sent++;

      //Kiküldjük a következő cím bájtot
      TWDR = ((ActualTWI.addr >> (8*(ActualTWI.AddressLength-AddressByte_Sent))) & 0xFF);

      TWCR = (1<<TWINT | (1<<TWEN) | (1<<TWIE)); //elindítjuk a küldést
```

```
   }
   else  //Ha kiküldtük a címet
   {
      if(ActualTWI.R_W == WRITE) //Ha éppen írás zajlik
      {
         if(DataByte_Sent < ActualTWI.DataLength)  //Ha még van kiküldetlen adat
         {
            DataByte_Sent++;

            //Kiküldünk egy adatbájtot
            TWDR = (ActualTWI.data>>(8*(ActualTWI.DataLength-DataByte_Sent))) & 0xFF;

            TWCR = (1<<TWINT | (1<<TWEN) | (1<<TWIE)); //elindítjuk a küldést
         }
         else  //Ha kiküldtük az adatot
         {
            TWCR = (1<<TWINT)|(1<<TWSTO)|(1<<TWEN)|(1<<TWIE);  //STOP kondíció küldése

            if(TWI_buff_rd(&ActualTWI))
               TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)|(1<<TWIE);  //START
            else
               TWI_RDY=1;
         }
      }
      else if (ActualTWI.R_W == READ)  //Ha épp olvasás zajlik
      {
         TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)|(1<<TWIE);  //REPEATED START kondíció
      }
   }
}

void StateHandler_0x10() /*A repeated START condition has been transmitted*/
{
   //Kiküldjük a slave címét, egyben azt is jelezve, hogy olvasni szeretnénk
   TWDR = ActualTWI.SLA | READ;
   TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE); //elindítjuk a küldést
}

void StateHandler_0x40() /*SLA+R has been transmitted; ACK has been received*/
{
   DataByte_Sent++;  //előre növeljük a kiolvasott adatbájtok számát
   if(DataByte_Sent < ActualTWI.DataLength)  //Ha még ezen kívül is van várunk adatbájtot
   {
      //folytatjuk az olvasást, majd ACK-al válaszolunk.
      TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE) | (1<<TWEA);
   }
   else  //Ha a következő az utolsó lesz
   {
      //folytatjuk az olvasást, majd NACK-al válaszolunk.
      TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
   }
}

void StateHandler_0x50() /*Data byte has been received; ACK has been returned*/
{
   long int tmp=0;
   tmp = TWDR;

   //Beírjuk az adatot a megfelelő helyre
   ActualTWI.data |= (tmp<<(8*(ActualTWI.DataLength - DataByte_Sent)));
   DataByte_Sent++;
   if(DataByte_Sent < ActualTWI.DataLength)  //Ha nem az utolsó adatbájtot várjuk
   {
      //folytatjuk az olvasást, majd ACK-al válaszolunk.
      TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE) | (1<<TWEA);
   }
   else //Ha a következő az utolsó lesz
   {
      //folytatjuk az olvasást, majd NACK-al válaszolunk.
      TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
```

```
   }
}

void StateHandler_0x58() /*Data byte has been received; NOT ACK has been returned*/
{
   //Beírjuk a bájtot a data megfelelő helyére
   ActualTWI.data |= TWDR<<(8*(ActualTWI.DataLength - DataByte_Sent));
   TWCR = (1<<TWINT)|(1<<TWSTO)|(1<<TWEN)|(1<<TWIE);  //STOP kondíció küldése

   DataHandler(); //Kezeljük a kész adatot
   if(TWI_buff_rd(&ActualTWI))
      TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)|(1<<TWIE);  //START
   else
      TWI_RDY=1;  //Jelezzük, hogy kész az olvasás
}

void StateHandler_0x20()/*SLA+W has been transmitted; NOT ACK has been received*/
{
   TWCR = (1<<TWINT)|(1<<TWSTO)|(1<<TWEN)|(1<<TWIE);  //STOP kondíció küldése
   _delay_ms(1);
   if(ActualTWI.Read_Action == Read_FIFO_dat && ActualTWI.R_W == READ)
   {
      //Kiolvassuk a FIFO read és write pointereket,
      //hogy megtudjuk, mennyit kell még kiolvasni
      TWI_start(READ, MAX_ADDRESS, FIFO_rd_ptr_reg, 1, 0);
      TWI_start(READ, MAX_ADDRESS, FIFO_wr_ptr_reg, 1, 0);
   }

   TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)|(1<<TWIE); //START
}
void StateHandler_0x30()/*Data byte has been transmitted; NOT ACK has been received*/

{
   TWCR = (1<<TWINT)|(1<<TWSTO)|(1<<TWEN)|(1<<TWIE);  //STOP kondíció küldése
   _delay_ms(1);
   if(ActualTWI.Read_Action == Read_FIFO_dat && ActualTWI.R_W == READ)
   {
      //Kiolvassuk a FIFO read és write pointereket,
      //hogy megtudjuk, mennyit kell még kiolvasni
      TWI_start(READ, MAX_ADDRESS, FIFO_rd_ptr_reg, 1, 0);
      TWI_start(READ, MAX_ADDRESS, FIFO_wr_ptr_reg, 1, 0);
   }
   TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)|(1<<TWIE); //START
}
void StateHandler_0x38()/*Arbitration lost in SLA+R/W or data bytes*/
{
   TWCR = (1<<TWINT)|(1<<TWSTO)|(1<<TWEN)|(1<<TWIE);  //STOP kondíció küldése
   _delay_ms(1);
   if(ActualTWI.Read_Action == Read_FIFO_dat && ActualTWI.R_W == READ)
   {
      //Kiolvassuk a FIFO read és write pointereket,
      //hogy megtudjuk, mennyit kell még kiolvasni
      TWI_start(READ, MAX_ADDRESS, FIFO_rd_ptr_reg, 1, 0);
      TWI_start(READ, MAX_ADDRESS, FIFO_wr_ptr_reg, 1, 0);
   }
   TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)|(1<<TWIE); //START
}
void StateHandler_0x48()/*SLA+R has been transmitted; NOT ACK has been received*/
{
   TWCR = (1<<TWINT)|(1<<TWSTO)|(1<<TWEN)|(1<<TWIE);  //STOP kondíció küldése
   _delay_ms(1);
   if(ActualTWI.Read_Action == Read_FIFO_dat && ActualTWI.R_W == READ)
   {
      //Kiolvassuk a FIFO read és write pointereket,
      //hogy megtudjuk, mennyit kell még kiolvasni
      TWI_start(READ, MAX_ADDRESS, FIFO_rd_ptr_reg, 1, 0);
      TWI_start(READ, MAX_ADDRESS, FIFO_wr_ptr_reg, 1, 0);
   }
   TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)|(1<<TWIE); //START
}
```

2.3.1.5 Interrupt kiszolgálás

Mint említettem, az I2C vezérlés megszakítások segítségével zajlik. Az utolsó lépés az volt, hogy a már megírt függvényeket megfelelően meghívjam egy I2C interrupt érkezésekor. Ha megszakítás érkezett, a diagnosztikai kódok alapján el kell dönteni, hogy milyen művelet történt most, és az alapján kell meghívni a megfelelő StateHandler függvényt. Ehhez egyszerűen ki kell olvasni a státusz regisztert (TWSR), és le kell maszkolni, mert az alsó három bitje nem a státuszt jelzi, azoknak más funkciójuk van. Az implementálás tehát egyszerű, a következőképp néz ki:

```
ISR(TWI_vect)
{
   volatile char State;
   State = (TWSR & 0xF8);

switch (State)
{
      /*A START condition has been transmitted*/
      case 0x08:  { StateHandler_0x08(); break; }

      /*A repeated START condition has been transmitted*/
      case 0x10:  { StateHandler_0x10(); break; }

      /*SLA+W has been transmitted; ACK has been received*/
      case 0x18:  { StateHandler_0x18(); break; }

      /*Data byte has been transmitted; ACK has been received*/
      case 0x28:  { StateHandler_0x28(); break; }

      /*SLA+R has been transmitted; ACK has been received*/
      case 0x40:  { StateHandler_0x40(); break; }

      /*Data byte has been received; ACK has been returned*/
      case 0x50:  { StateHandler_0x50(); break; }

      /*Data byte has been received; NOT ACK has been returned*/
      case 0x58:  { StateHandler_0x58(); break; }

      /*Data byte has been transmitted; NOT ACK has been received
      case 0x30:  { StateHandler_0x30(); break;}

      /*SLA+W has been transmitted; NOT ACK has been received*/
      case 0x20:  { StateHandler_0x20(); break;}

      /*Arbitration lost in SLA+R/W or data bytes*/
      case 0x38:  { StateHandler_0x38(); break;}

      /*SLA+R has been transmitted; NOT ACK has been received*/
      case 0x48:  { StateHandler_0x48(); break;}

      default:    {break;}
   }
}
```

2.3.2 UART interfész

A számítógéppel a kommunikáció UART interfészen keresztül történik. Ez is megszakítások segítségével zajlik. Ha adatot szeretnénk küldeni, de még nem végzett a hardver az előző üzenet elküldésével, akkor az adatot beírjuk egy pufferbe, és az adatküldés végét jelző interrupt érkezésekor újból megpróbáljuk elküldeni. A felhasználói vezérléshez a szoftver egyszerű parancsok értelmezésére képes. Minden parancs három ASCII karakterből áll, amiket szóközzel, vesszővel, vagy pontosvesszővel kell elválasztani. A program ez alapján dönti el, hogy milyen módban fusson, illetve mit csináljon. A '**d**' karakter elküldésével törölhető az elkezdett szekvencia. A parancsok a következők:

w *AA DD*: Beírja a *DD* adatot a memória *AA* címére. Ez a funkció csak diagnosztikai célt szolgál, az EEPROM-mal történő I2C kommunikációt teszteltem vele.

r *AA LL*: Kiolvas *LL* bájtnyi adatot az *AA* címről. Ez is csak tesztelésre készült.

c *xx xx*: (Collect Data) Lekérdezi az EEPROM tartalmát, majd megáll. (*xx*: dont care)

m m *xx*: (Mode: Measure) Elindítja a mérést.

m p *xx*: (Mode: Pause) Ideiglenesen egállítja a mérést.

m s *xx*: (Mode: Stop) Alaphelyzetbe áll, minden pointert 0-ba állít, és megáll.

i l *E*: (Info: Live stream) Élő idejű megjelenítés ki/be kapcsolása (E=1 => be)

i t *E*: (Info: Time) Idő adatok küldése (E=1 => be)

i h *E*: (Info: Heart rate) Szívritmus adatok küldése (E=1 => be)

Ezeknek a parancsoknak a kezelése csak a főprogramban kerül implementálásra. Az UART kezelésbe csak az érkező parancsok elmentése tartozik bele. Az ATmega328P egy darab UART interfésszel rendelkezik, amely a 0 számot viseli.

2.3.2.1 UART_init() függvény

Ez a függvény inicializálja az UART interfészt. A sebességet a lehető legnagyobbra választottam, hogy gyors legyen a PC-re történő adatküldés. Engedélyeztem az interrupt-okat. Az alább beállított regiszterek az interfész konfigurálására szolgálnak. A függvény implementálása a következő:

```
void UART_init()  //UART inicializálása
{              //Aszinkron, 8 adatbit, 1 stopbit, páros paritás

   UBRR0=0; //Baudrate: 1M bps, táblázatból leolvasva

//             ,----------7: RXCIEn: RX Complete Interrupt Enable n
//             |,---------6: TXCIEn: TX Complete Interrupt Enable n
//             ||,--------5: UDRIEn: USART Data Register Empty Interrupt Enable n
//             |||,-------4: RXENn: Receiver Enable n
//             ||||,------3: TXENn: Transmitter Enable n
//             |||||,-----2: UCSZn2: Character Size n
//             ||||||,----1: URXB8n: Receive Data Bit 8 n
//             |||||||,---0: TXB8n: Transmit Data Bit 8 n
//             ||||||||
//             76543210
      UCSR0B=0b11011000;
//             ,----------7: UMSELn1: USART Mode Select bit1
//             |,---------6: UMSELn0: USART Mode Select bit0
//             ||,--------5: UPMn1: Parity Mode bit1
//             |||,-------4: UPMn0: Parity Mode bit0
//             ||||,------3: USBSn: Stop Bit Select
//             |||||,-----2: UCSZn1: Character Size bit1
//             ||||||,----1: UCSZn0: Character Size bit0
//             |||||||,---0: UCPOLn: Clock Polarity
//             ||||||||
//             76543210
      UCSR0C=0b00100110;
}
```

2.3.2.2 transmit_char() függvény

A függvény elküld egy bájtot UART-on. Ha nem sikerül, mert foglalt az UART interfész, akkor eltároljuk az adatot a pufferbe, majd később elküldjük, ha végzett. A küldés sorrendjének biztos megőrzése miatt először mindenképp beírja a pufferbe az adatot, és egy újonnan kiolvasott adatot küld el, mert lehet, hogy van adat, amelyik régebb óta vár küldésre. Az UART adatregisztere az UDR0. Az ebbe történő írás elindítja az adatküldést, Ha pedig adat érkezik, innen lehet kiolvasni. Azt, hogy üres-e a regiszter, az UCSR0A regiszter UDRE0 bitjével tudjuk ellenőrizni. Az UART interruptot ideiglenesen le kell tiltani, mert ha a transmit függvény meg van szakítva, annak nem várt következményei lehetnek. Az UART kimeneti pufferbe írás függvényét később részletezem.

```
void transmit_char(char UART_data)  //Egy bájt küldése
{
   char tmp;
   UCSR0B &= ~(1<<TXCIE0); //UART megszakítás tiltása
   UART_out_buff_wr(UART_data);

   if(UCSR0A & 1<<UDRE0)   //Ha üres az adatregiszter, küldhetjük az adatot
   {
      if(UART_out_buff_rd(&tmp))
      UDR0=tmp;
   }
   UCSR0B |= (1<<TXCIE0);  //UART megszakítás engedélyezése
}
```

2.3.2.3 UART_out_buff_wr() függvény

Beír egy adatot az UART kimeneti pufferbe. Ez egy ciklikus puffer, egy író és egy olvasó pointerrel rendelkezik. Úgy implementáltam, hogy mindig lehet bele írni, akkor is, ha tele van, csak akkor a legrégebbi adat elvész. A szükséges változók és definíciók:

```
#define BUFFSIZE 100    //A puffer mérete

char UART_out_puffer[BUFFSIZE];  //A puffer
unsigned char UART_out_wr=0, UART_out_rd=0;  //író és olvasó pointerek
char UART_out_isempty=1;//Azt jelzi, hogy üres-e a puffer.
```

A függvényt a következőképp implementáltam:

```
void UART_out_buff_wr(char UART_data)
{
   UART_out_buffer[UART_out_wr] = UART_data; //Adat beírása
   UART_out_isempty = 0;   //Mostmár biztosan nem üres.
   UART_out_wr=(UART_out_wr+1)%BUFFSIZE;  //Léptetjük az író indexet

   //Ha utolértük az olvasó indexet, tehát a puffer tele van,
   //léptetjük az olvasó indexet.
   if(UART_out_wr==UART_out_rd)
      UART_out_rd=(UART_out_rd+1)%BUFFSIZE;
}
```

2.3.2.4 UART_out_buff_rd() függvény

A függvény kiolvas egy adatot az UART kimeneti pufferből a megadott változóba. 0-val tér vissza, ha a puffer üres, és 1-el, ha sikerült az olvasás. Csak a fent említett definíció és változók kellenek hozzá. Implementációja a következő:

```
char UART_out_buff_rd(char* UART_data) //Kiolvas egy adatot az UART_out_bufferből
{
   if(UART_out_isempty) //Ha üres az UART_out_buffer, nem tudunk olvasni
      return 0;

   *UART_data=UART_out_buffer[UART_out_rd];  //adat kiolvasása
   UART_out_rd=(UART_out_rd+1)%BUFFSIZE;  // Léptetjük az olvasó indexet

   //Ha utolértük az író indexet, akkor az UART_out_buffer üres lett.
   if(UART_out_rd==UART_out_wr)
      UART_out_isempty=1;

   return 1;
}
```

2.3.2.5 Interrupt kiszolgálás

Az UART interfész két esetben kér megszakítást: Az egyik, amikor elküldtünk egy adatot, a másik, amikor beérkezett egy adat.

Ha beérkezett egy üzenet, akkor aszerint, hogy az érkező parancs hányadik bájtjánál tartunk, el kell tárolni a bájtot, és ha kész a parancs, azt jelezni kell. Ehhez a következő változókat és definíciót használtam:

```
#define SEQ_LENGTH 3     //Az UART-on érkező parancs bájtjainak száma

char UART_sequence[SEQ_LENGTH];  //A beérkező parancs bájtjait tartalmazó tömb
volatile char UART_sequence_state=0; //Itt követjük, hogy hányadik bájtnál járunk
volatile char sequenceRDY=0;  //Ezzel jelöljük, ha beérkezett egy teljes parancs.
```

A beérkező üzenet megszakításkérése ezek alapján a következőképp van kiszolgálva:

```
ISR(USART_RX_vect) //Olvasatlan beérkezett adat IT. Beolvassuk a kapott szekvenciát.
{
   char tmp=UDR0;
   if(tmp=='d')//Ha ’d’, elölről kezdjük a szekvenciát.
   {
      UART_sequence_state=0;
      sequenceRDY=0;
   }
   else
   {
      if(sequenceRDY == 0) //Ha nincs kezeletlen (kész) parancs
      {
         if((tmp==' ') || (tmp==',') || (tmp==';'))//Ha elválasztó karakter
         {
            UART_sequence_state++;  //Akkor léptetjük a státuszt

            if(UART_sequence_state >= SEQ_LENGTH)  //Ha kész, jelezzük, és nullázunk.
            {
               sequenceRDY=1;
               UART_sequence_state=0;
            }
         }
         else
         {
            if(UART_sequence_state==0)//Első üzenet
            {
               for(int i=0; i<SEQ_LENGTH; i++)//nullázzuk a szekvenciát
                  UART_sequence[i]=0;

               UART_sequence[0]=tmp;//beírjuk az új elemet
            }
            else//Ha nem az első
            {
               //Feltoljuk fél bájttal, mert egy
               //hexa karakterként értelmezzük, és jön a következő.
               UART_sequence[UART_sequence_state]=UART_sequence[UART_sequence_state]<<4;

               //Ha hexa karakternek megfelelő betű, számoljuk az értékét és beírjuk.
               if(tmp>='0' && tmp<='9')
                  UART_sequence[UART_sequence_state] |= (tmp-'0');

               else if(tmp>='A' && tmp<='F')
                  UART_sequence[UART_sequence_state] |= (tmp-'A' + 0xA);

               else if(tmp>='a' && tmp<='f')
                  UART_sequence[UART_sequence_state] |= (tmp-'a' + 0xA);
               else
                  UART_sequence[UART_sequence_state] |= (tmp);
            }
         }
      }
   }
}
```

Az arra vonatkozó interrupt kérést, hogy egy üzenetet sikeresen elküldtünk az UART-on keresztül, egyszerűbb kezelni. Ilyenkor csak meg kell nézni, hogy van-e még kiküldendő üzenet, és, ha igen, elküldeni:

```
ISR(USART_TX_vect)//transmit shiftregiszter kiürült (IT)
{
   char tmp;
   while(!(UCSR0A & (1<<UDRE0)));//Megvárjuk, hogy tényleg kiürüljön az adatregiszter
   if(UART_out_buff_rd(&tmp))
      UDR0=tmp;
}
```

2.3.3 Adat-buffer

A mérési adatok tárolását egy adat-puffer biztosítja. A pufferbe megszakításokkal vezérelve, automatikusan íródnak be az adatok, amiket aztán a főprogramban olvasunk ki belőle, és használjuk fel. Működése nagyon hasonló az UART kimeneti puffer működéséhez. A felhasznált változók:

```
long int data_buffer[BUFFSIZE];  //A ciklikus puffer
char wr=0, rd=0;     //író és olvasó pointer
char isempty=1;   //Ezzel jelezzük, ha üres a puffer.
```

2.3.3.1 buff_rd() függvény

Kiolvas egy adatot a pufferből a megadott változóba. 0-val tér vissza, ha a puffer üres, és 1-el, ha sikerült az olvasás.

```
char buff_rd(long int* data)  //Kiolvas egy adatot a pufferből
{
   if(isempty) //Ha üres a buffer, nem tudunk olvasni
      return 0;

   *data=data_buffer[rd];  //adat kiolvasása
   rd=(rd+1)%BUFFSIZE;  // Léptetjük az olvasó indexet


   if(rd==wr)  //Ha utolértük az író indexet, akkor a puffer üres lett.
      isempty=1;

   return 1;
}
```

2.3.3.2 buff_wr() függvény

Beír egy adatot az adatpufferbe. Ha a puffer tele van, felülírja a legrégebbi adatot:

```
void buff_wr(long int data)
{
   data_puffer[wr] = data; //Adat beírása
   isempty=0;  //Mostmár biztosan nem üres.
   wr=(wr+1)%BUFFSIZE;  //Léptetjük az író indexet

   //Ha utolértük az olvasó indexet, tehát a puffer tele van,
   //léptetjük az olvasó indexet.
   if(wr==rd)
      rd=(rd+1)%BUFFSIZE;
}
```

2.3.4 Mérő-chip használata

2.3.4.1 MAX30100_init() függvény

Ebben a függvényben inicializáljuk a mérő-chipet. Ezt az egyes regisztereibe való I2C írással tehetjük meg. Több regiszter beállítása is külön magyarázatra szorul. A következő kódrészletek mind a MAX30100_init() függvényen belül vannak, csak az érthetőség kedvéért külön-külön megmagyarázom őket:

```
void MAX30100_init()
{
```

A Mode Configuration regiszterben beállíthatom, hogy milyen módban működjön az eszköz, és legyen-e hőmérsékletmérés. A Mode Control jelentését az adatlapban szereplő táblázatból olvastam ki. A Shutdown Control bit alvó állapotba állítja az eszközt. A Reset Control pedig minden regisztert alap állapotba állít, újraindítja az eszközt. A függvény először visszaállítja az eszközt, majd elkezdi beállítani a regisztereket.

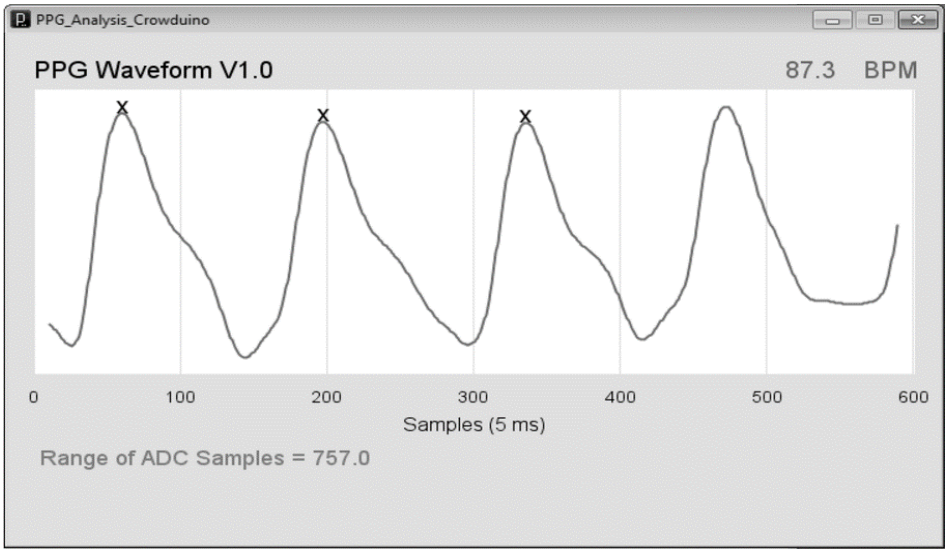
```
//                              ,__________________REG: MODE CONFIGURATION REGISTER
//                              |         ,__________7: Shutdown Control (SHDN)
//                              |         |,_________6: Reset Control (RESET)
//                              |         ||,________5: RESERVED
//                              |         |||,_______4: RESERVED
//                              |         ||||,______3: Temperature Enable
//                              |         |||||,_____2: Mode Control 2 }
//                              |         ||||||,____1: Mode Control 1 }} HR only
//                              |         |||||||,___0: Mode Control 0 }
//                              |         ||||||||
//                             REG        76543210
TWI_start(WRITE, MAX_ADDRESS, 0x06, 1, 0b01000000);//Reset küldése
_delay_ms(100);                                     //Várakozás
TWI_start(WRITE, MAX_ADDRESS, 0x06, 1, 0b00000010);//Pulzusmérés mód beállítása
```

Az SpO2 Configuration Regiszterben beállíthatjuk a mintavételi sebességet, és a LED világításának pulzusszélességét egy mintavételben.

A mintavételi sebesség meghatározása egy kis átgondolást igényelt. Azt már fentebb kifejtettem, hogy időmérés szempontjából 10ms-os pontosságot szeretnék, vagyis ez a maximum, amennyit „tévedhet” egy pulzusdetektálás. Ez alapján $100\frac{mintavétel}{másodperc}$ a minimális mintavételi frekvencia.

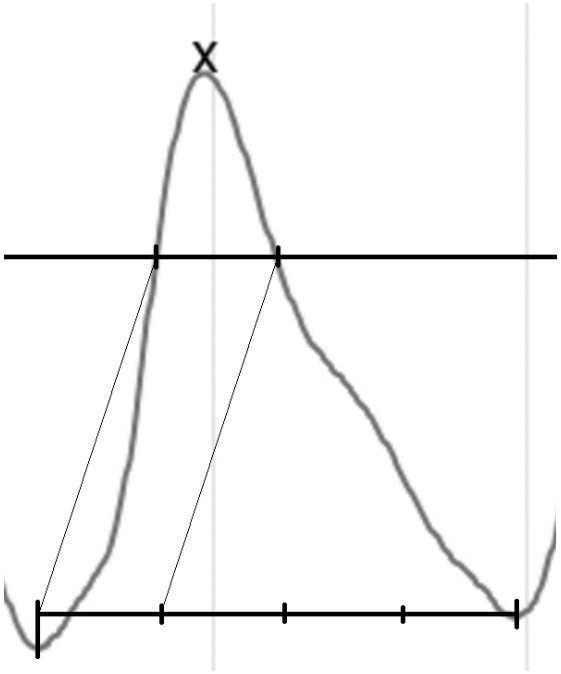
De itt azt is meg kell fontolni, hogy ez elég-e ahhoz, hogy egy szívverést biztosan detektáljunk, tehát elég gyakran vesszük-e a mintákat ahhoz, hogy biztosan legyen olyan mintánk, amely átlépi a beállított küszöböt, amelytől már szívverést detektálunk. Ehhez ismerni kellene a várható jelet. Ennek érdekében kutattam az interneten hasonló projektek

után, és találtam egy munkát[6], ami szintén fotopletizmográfiát használva, Arduino segítségével méri a szívritmust, és számítógépen megjeleníti azt. A megjelenített mérési eredményt az alábbi ábra mutatja:



23. ábra - Egy másik munka[6] mérési eredménye

Az ábrán elsősorban a jel alakja érdekes. Mint az alábbi ábrán látszik, ha behúzunk egy szintet, amely átlépése esetén egyértelműen szívverésről van szó, azt látjuk, hogy az az időtartam, amíg a jel a vonal fölött van, körülbelül a teljes periódusidő negyede. Az ábrán levő hosszúságmérések csak szemmértékkel történtek, de az eredmény így is egyértelmű:



24. ábra - A detektálás időtartama

Nézzünk tehát egy gyors, 120-as pulzust, aminek a periódusideje 0.5 másodperc, vagyis 500ms. Ekkor a pulzus detektálására $\frac{500ms}{4} = 125ms$ áll rendelkezésre. 10ms-os mintavétel mellett tehát bőven lesz olyan mintánk, amely a kitűzött szint felett van. Következésképpen a $100\frac{mintavétel}{másodperc}$ megfelelő sebesség. Később az eszköz és a szoftver tesztelésekor kiderült, hogy a sebesség valóban megfelelő.

A LED-ek pulzusszélességének a pontos analóg-digitális konverzió céljából a legmagasabb, 1.6ms-os értéket választottam.

A leírtak alapján az SpO2 Configuration regisztert a következőképp inicializáltam:

```
//                              ,----------------REG: SpO2 CONFIGURATION REGISTER
//                              |         ,----------7: RESERVED
//                              |         |,---------6: SpO2 High Resolution
//                              |         ||,--------5: RESERVED.
//                              |         |||,-------4: SpO2 Sample Rate 2 }
//                              |         ||||,------3: SpO2 Sample Rate 1 }} 100
//                              |         |||||,-----2: SpO2 Sample Rate 0}
//                              |         ||||||,----1: LED Pulse Width 1}
//                              |         |||||||,---0: LED Pulse Width 0}} 1.6 ms
//                              |         ||||||||
//                             REG       76543210
TWI_start(WRITE, MAX_ADDRESS, 0x07, 1, 0b01000111);
```

A LED Configuration Regiszterrel lehet szabályozni a LED-ek áramát. A vörös LED-re nincs szükség egyszerű pulzusmérésnél. Az infravörös LED áramát a jobb mérési eredményekért a maximális, 50mA-es értékre állítottam.

```
//                              ,----------------REG: LED CONFIGURATION REGISTER
//                              |         ,----------7: Red LED Current }
//                              |         |,---------6: Red LED Current }} 0 mA
//                              |         ||,--------5: Red LED Current }
//                              |         |||,-------4: Red LED Current }
//                              |         ||||,------3: IR LED Current }
//                              |         |||||,-----2: IR LED Current }
//                              |         ||||||,----1: IR LED Current }} 50 mA
//                              |         |||||||,---0: IR LED Current Control }
//                              |         ||||||||
//                             REG       76543210
TWI_start(WRITE, MAX_ADDRESS, 0x09,  1, 0b00001111);
```

Az Interrupt Enable Regiszterben azt állíthatjuk be, hogy mely események húzzák földpotenciálra az INT lábat, ezzel jelezve a mikrokontrollernek. Az ENB_SPO2_RDY és az ENB_TEMP_RDY interruptokra csak pulzoximetria módban lenne szükség, egyszerű pulzusmérésnél nem. A többi megszakítást engedélyeztem.

```
//                              ,----------------REG: INTERRUPT ENABLE
//                              |         ,----------7: FIFO Almost Full
//                              |         |,---------6: Temperature Ready
//                              |         ||,--------5: Heart Rate Data Ready
//                              |         |||,-------4: SpO2 Data Ready
//                              |         ||||,------3: RESERVED
//                              |         |||||,-----2: RESERVED
//                              |         ||||||,----1: RESERVED
//                              |         |||||||,---0: RESERVED
//                              |         ||||||||
//                             REG        76543210
TWI_start(WRITE, MAX_ADDRESS, 0x01, 1, 0b10100000);
```

Ezután ki kell olvasni az interrupt státusz regisztert, mert ez biztosítja, hogy az /INT vonal felszabaduljon, és újabb megszakítások érkezhessenek.

```
TWI_start(READ, MAX_ADDRESS, Read_INT_state, 1, 0);
}
```

2.3.4.2 DataHandler() függvény

A függvény a StateHandler_0x58() függvényben hívódik meg, vagyis akkor, ha kiolvastunk egy adatot valamelyik eszközről az I2C interfészen keresztül. Feladata, hogy az alkalmazásnak megfelelően, az olvasás típusa alapján eldöntse, hogy mit kell csinálni a kiolvasott adattal.

A kódban lévő definíciók és változók, amiket még nem említettem, elég beszédes nevűek, ezért most csak röviden írok róluk. A DataToRead változóban követjük, hogy hány adatot kell még biztosan kiolvasni a mérő-chip FIFO-jából. Ez a változó a chip olvasó és író pointere alapján számolódik. Az alábbi sorban található maradék négy változó a chip adott regisztereiből kiolvasott értékek:

```
volatile char DataToRead=0, IT_State=0, FIFO_wr_ptr=0, FIFO_rd_ptr=0, OVF_cnt=0;
```

Az OVF_cnt azt számolja, hogy hány minta veszett el a kiolvasás elmaradása miatt. Ezt a változót egyelőre nem használom.

Az alábbi definíciók a MAX30100 mérő-chip adatai alapján készültek, és a kód áttekinthetőségét segítik. Az MSK végű definíciók az interrupt státusz maszkolására szolgálnak:

```
#define FIFO_length 16  //A FIFO-ban található (4 bájtos) minták maximális száma
#define ALMOST_FULL_MSK (1<<7)//FIFO majdnem tele
#define HR_RDY_MSK (1<<5)  //Minta kész
```

A függvény implementációja a következő:

```
void DataHandler()
{
   switch(ActualTWI.Read_Action) //vizsgáljuk, hogy melyik olvasási művelet ment végbe
   {
      case Read_INT_state: //A mérőchip interrupt státusz regisztere
      {
```

```
            IT_State=ActualTWI.data;
            if(mode!=STOPPED)
            {
               if(IT_State & ALMOST_FULL_MSK)//Ha majdnem tele a FIFO
               {
                  //Tudjuk, hogy még ennyit biztosan ki kell olvasni
                  DataToRead=FIFO_length-1;
                  //elkezdünk olvasni a FIFO-ból
                  TWI_start(READ, MAX_ADDRESS, FIFO_dat_reg, 4, 0);
               }
               if(IT_State & HR_RDY_MSK)  //Ha kész egy adat
               {
                  //Kiolvassuk a FIFO read és write pointereket,
                  //hogy megtudjuk, mennyit kell még kiolvasni
                  TWI_start(READ, MAX_ADDRESS, FIFO_rd_ptr_reg, 1, 0);
                  TWI_start(READ, MAX_ADDRESS, FIFO_wr_ptr_reg, 1, 0);
               }
            }
            break;
         }

         case Read_FIFO_wr_ptr:  //Ha mérőchip FIFO write pointere
         {

            FIFO_wr_ptr = ActualTWI.data; //frissítjük a pointer értékét

            //kiszámoljuk, mennyi adatot kell olvasni:
            if(FIFO_wr_ptr>FIFO_rd_ptr)
               DataToRead = FIFO_wr_ptr - FIFO_rd_ptr;
            else
               DataToRead = FIFO_rd_ptr - FIFO_wr_ptr;

            if(mode!=STOPPED)
            {
               if(DataToRead>0)  //Ha van olvasandó adat
               TWI_start(READ, MAX_ADDRESS, FIFO_dat_reg, 4, 0);   //Elkezdjük az olvasást
            }
            break;
         }

         case Read_OVF_cnt:   { OVF_cnt   = ActualTWI.data;  break; }
         case Read_FIFO_rd_ptr:  { FIFO_rd_ptr  = ActualTWI.data;  break; }

         case Read_FIFO_dat:  //Egy adat a FIFO-ból
         {
            //Pulzusmérésnél csak a felső két bájt számít,
            //mert ez az IR LED adata.
            buff_wr(ActualTWI.data>>16);  //beírjuk a pufferbe.
            if(mode!=STOPPED)
            {
               //Kiolvassuk a FIFO read és write pointereket,
               //hogy megtudjuk, mennyit kell még kiolvasni
               TWI_start(READ, MAX_ADDRESS, FIFO_rd_ptr_reg, 1, 0);
               TWI_start(READ, MAX_ADDRESS, FIFO_wr_ptr_reg, 1, 0);
            }
            break;
         }

         case Read_EEPROM: //Ha a memóriából olvastunk
         {
            transmit_char(ActualTWI.data/100 + '0');
            transmit_char('.');
            transmit_char((ActualTWI.data%100)/10 + '0');
            transmit_char(ActualTWI.data%10 + '0');
            transmit_char('s');
            transmit_char(' ');
            break;
         }
      }
   }
```

2.3.4.3 Külső megszakítás kezelése

Ha a mérő-chip jelezni szeretné a mikrokontrollernek, hogy valamilyen esemény történt, amit a mikrokontrollernek kellene kezelnie, például ki kellene olvasni valamelyik regisztert, akkor az INT vonalat földpotenciálra tudja húzni. Ez a vonal a mikrokontroller external interrupt lábára csatlakozik, és beállítás alapján alacsony szint, szintváltozás, illetve fel- vagy lefutó él hatására megszakítást kér.

Ennek kezelésére két függvény szolgál. Az első függvény inicializálja a mikrokontroller külső megszakítás-kezelését. Engedélyezi a 0 számmal jelölt külső megszakítás interrupt kérését, és beállítja, hogy lefutó élre kérjen megszakítást:

```
void INT0_init()
{
//   ,----------------REG: External Interrupt Control Register A
//   |       ,----------7: Reserved
//   |       |,---------6: Reserved
//   |       ||,--------5: Reserved
//   |       |||,-------4: Reserved
//   |       ||||,------3: ISC11: Interrupt Sense Control 1 Bit 1
//   |       |||||,-----2: ISC10: Interrupt Sense Control 1 Bit 0
//   |       ||||||,----1: ISC01: Interrupt Sense Control 0 Bit 1
//   |       |||||||,---0: ISC00: Interrupt Sense Control 0 Bit 0
//   |       ||||||||
//  REG      76543210
   EICRA = 0b00000010;

//   ,----------------REG: External Interrupt Mask Register
//   |       ,----------7: Reserved
//   |       |,---------6: Reserved
//   |       ||,--------5: Reserved
//   |       |||,-------4: Reserved
//   |       ||||,------3: Reserved
//   |       |||||,-----2: Reserved
//   |       ||||||,----1: INT1: External Interrupt Request 1 Enable
//   |       |||||||,---0: External Interrupt Request 0 Enable
//   |       ||||||||
//  REG      76543210
   EIMSK = 0b00000001;
}
```

A második függvény a beérkező megszakítás kezelésére szolgál. Mindössze annyit csinál, hogy elindít egy olvasást a mérő-chip interrupt státuszt tároló regiszteréből. Ha az olvasás kész, az eredményt a már említett DataHandler() függvény értelmezi.

```
ISR(INT0_vect)
{
   TWI_start(READ, MAX_ADDRESS, Read_INT_state, 1, 0);
}
```

2.3.5 Analóg-digitális átalakító

Az ADC-re azért volt szükség, hogy mérjük a mérő-chip belső áramköreihez használt, 1.8V-os tápján megjelenő feszültséget, és ha az kellően megközelíti a kívánt értéket, csak akkor kapcsoljuk rá a LED-ekhez felhasznált 3.3V-os tápot. A perifériát úgy állítottam be, hogy nem mér automatikusan, csak szoftveres irányítással, hisz erre csak egyszer, a program elején van szükség. A 0 számú csatornát használom, referencia feszültségnek a kártya 3.3V-os tápját állítottam be. Az inicializáló függvény:

```
void ADC_init()//Belső 3.3V-os referencia, ADC0 channel (PC0), 128-as prescaler,
               //Single Conversion, Left Adjust Result
{
//   ,----------------REG: ADC Multiplexer Selection Register
//   |     ,----------7: REFS [1]: Reference Selection Bits 1 }} AVCC
//   |     |,---------6: REFS [0]: Reference Selection Bits 0 }
//   |     ||,--------5: ADLAR: ADC Left Adjust Result
//   |     |||,-------4: Reserved
//   |     ||||,------3: MUX [3]: Analog Channel Selection Bits 3 }
//   |     |||||,-----2: MUX [2]: Analog Channel Selection Bits 2 }}ADC0
//   |     ||||||,----1: MUX [1]: Analog Channel Selection Bits 1 }
//   |     |||||||,---0: MUX [0]: Analog Channel Selection Bits 0 }
//   |     ||||||||
//  REG    76543210
   ADMUX=0b01100000;

//    ,----------------REG: ADC Control and Status Register A
//    |     ,----------7: ADEN: ADC Enable
//    |     |,---------6: ADSC: ADC Start Conversion
//    |     ||,--------5: ADATE: ADC Auto Trigger Enable
//    |     |||,-------4: ADIF: ADC Interrupt Flag
//    |     ||||,------3: ADIE: ADC Interrupt Enable
//    |     |||||,-----2: ADPS[2:0]: ADC Prescaler Select Bits }
//    |     ||||||,----1: ADPS[2:0]: ADC Prescaler Select Bits }} 128
//    |     |||||||,---0: ADPS[2:0]: ADC Prescaler Select Bits }
//    |     ||||||||
//   REG    76543210
   ADCSRA=0b10000111;
}
```

2.3.6 Időzítők

A projekt megvalósításához a mikrokontroller két időzítőjét használtam fel: egy 16 bites időzítőt a szívritmusok közt eltelt idők mérésére, és egy 8 bitest a folyamatos UART-üzenet küldés időzítésére.

2.3.6.1 Inicializálás

A 8 bites timer 20 ms-onként kér megszakítást, hogy elküldje az UART üzeneteket. Először áll, és csak akkor indul el, ha a megfelelő parancs érkezik. Elindítása a TCCR1B regiszter utolsó három bitjének TIM0_PRESCALER_MSK (0b101) definícióval történő állításával lehetséges. Compare match esetén nullázódik a számláló.

```
void TIMER0_init()//Timer0 inicializálása  (8 bites, ADC és UART vezérlése)
{              //CTC mód, 1024-es prescaler, de először áll.
   GTCCR=0;
   TCCR0A=0b00000010; // | COM0A1 | COM0A0 | COM0B1 | COM0B0 | -  | -  | WGM01 | WGM00
   TCCR0B=0;   // | FOC0A | FOC0B |  -  |  -  | WGM02 | CS02 | CS01 | CS00
   TCNT0=0;
   OCR0A=150;  //1024-es prescaler-nél ez kb. 20ms
   TIMSK0=0b00000010;//Csak compare match A
}
```

A 16 bites timer a szívverések közt eltelt idő mérésére szolgál. Kezelése szoftveresen történik, ezért nem kér interruptot, és nem nullázza magát.

```
void TIMER1_init()//Timer1 inicializálása (16 bites, idők mérése)
{              //normál mód, 1024-es prescaler
   TCCR1A=0b00000000;// | COM1A1 || COM1A0 | COM1B1 | COM1B0 |  -  |  -  | WGM11 | WGM10
   TCCR1B=0b00000101;// | ICNC1 | ICES1 |  -  | WGM13 | WGM12 | CS12 | CS11 | CS10
   TCNT1=0;
   TIMSK1=0b00000000;//nem kell IT, a főciklusban kezeljük
}
```

2.3.6.2 TIMER0 megszakítás kezelés

Itt egyszerűen kiküldjük az adatokat a megfelelő protokoll szerint:

```
ISR(TIMER0_COMPA_vect)  //TIMER 0 Compare Match A IT (UART adatküldés)
{
   transmit_char((char)0xAA);
   transmit_char((char)(UART_send));
   transmit_char((char)(UART_send>>8));
   transmit_char((char)(~0xAA));
}
```

2.3.7 Főprogram

A főprogramnak három feladata van: Inicializálja, beállítja az eszközt. Figyeli az UART-on beérkezett parancsokat, és eszerint vált a működési módok között. Kiszolgálja a működési módokat, vagyis MEASURE módban vizsgálja a beérkező adatokat, és elmenti a szívverések közt eltelt időt az EEPROM-ba, COLLECT_DATA módban pedig kiolvassa az EEPROM tartalmát, és elküldi UART-on a számítógépnek. A könnyebb megértés érdekében szétszedtem a kódot, de valójában az egész a main() függvényen belül található. A működéshez néhány új változót és típusdefiníciót kellett bevezetni:

```
int main(void)
{
   //Ezzel követjük, hogy az eszköz épp milyen módban van, mi a "feladata"
   typedef enum {MEASURE, COLLECT_DATA, STOPPED} mode_type;
   mode_type mode=STOPPED;
   unsigned long int EEPROM_ptr=0;      //Az EEPROM-ban való navigáláshoz
   char beat_ack=0;  //Azt jelzi, hogy az adott szívverést már észrevettük.
   long int tmp;  //Ideiglenes változó az adatok kezelésére
   unsigned int min=0xFFFF, max=0;  //A jelfeldolgozó algoritmushoz szükséges határok
   int mincnt=0, maxcnt=0; //A jelfeldolgozó algoritmushoz szükséges számlálók
   volatile unsigned int time;//Eltelt idő változója
   char times=0, hr=0;  //idő és szívritmus küldésének engedélyezése
```

2.3.7.1 Eszköz elindítása

A program inicializálja az UART, I2C, External Interrupt és ADC perifériákat, majd engedélyezi a megszakításokat. Ezután beállítja a PORT-ok kezdőértékeit és irányát, majd a mérő-chip 1.8V-os tápjának megjelenése után bekapcsolja a 3.3V-os tápot. Most már inicializálhatja a mérő-chipet.

```
UART_init();
TWI_init();
INT0_init();
ADC_init();

sei(); //Globális IT engedélyezés
DDRB=1<<PORTB5;   //User LED Output
DDRC=0b00001110; //csak a három írásvédelem output
PORTD |= (1<<PORTD3); //A mérőchip tápja még ne legyen bekapcsolva
DDRD=(1<<PORTD3); //A mérőchip tápjának bekapcsolása miatt output
PORTC=0; //Az írásvédelem biteknek 0-nak kell lenni.

while(ADCH<132)   //várunk, amíg a mérő chip tápfeszültsége meghaladja a 1.7V-ot
{
   ADCSRA |= (1<<ADSC); //Új mérés indítása
   while(!(ADCSRA & (1<<ADIF))); //megvárjuk, míg kész az adat.
}
PORTD &= ~(1<<PORTD3); //ezután bekapcsolhatjuk a mérőchip tápját.

MAX30100_init();
```

2.3.7.2 UART parancs figyelése

Ezután egy végtelen while ciklus kezdődik. Az alábbi kód mind ezen belül van. Először megvizsgáljuk, hogy érkezett-e parancs az UART-on keresztül, és ha igen, teljesítjük. Mindig vissza kell állítani a sequenceRDY változót, hogy az UART fogadó függvény tudja, hogy kiszolgáltuk a parancsot, jöhet új parancs.

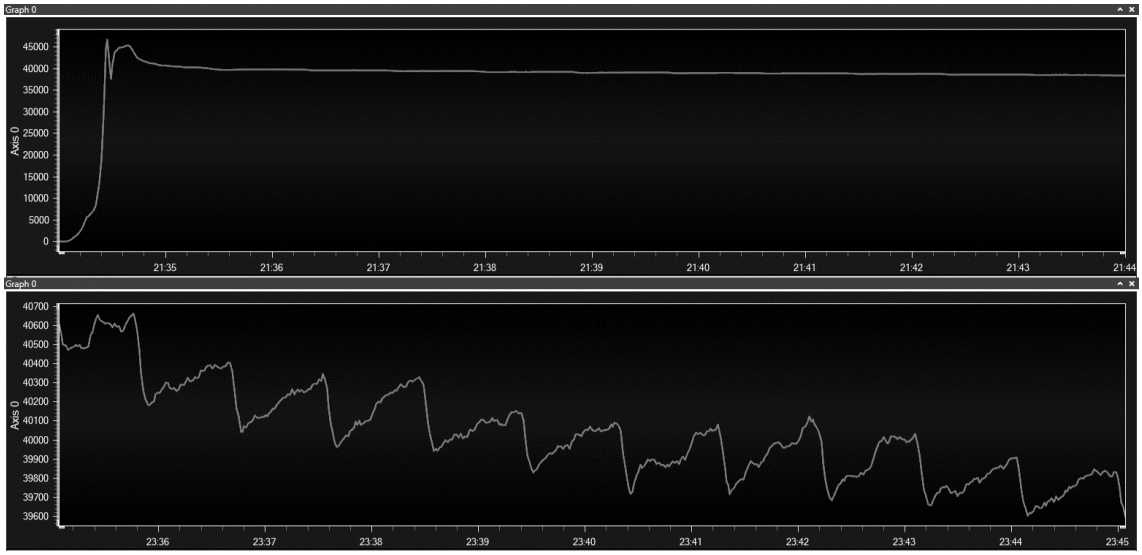
```
while(1)
   {
   //Ha kész egy vezérlő parancs az UART-on,
   //az elsőként beérkezett bájt alapján döntünk
   if(sequenceRDY)
   {
      switch (UART_sequence[0])
      {
         case 'w': { //Ha 0, beírjuk a megadott címre a megadott adatot
            TWI_start(WRITE, MEM_ADDRESS, UART_sequence[1], 1, UART_sequence[2]);
            sequenceRDY = 0;
            break; }

         case 'r': { //Ha 1, kiolvasunk megadott számú adatot a megadott címről
            TWI_start(READ, MEM_ADDRESS, UART_sequence[1], UART_sequence[2], 0);
            sequenceRDY = 0;
            break;}

         case 'c': { //Ha 2, átlépünk adat-lekérdezés módba
            mode = COLLECT_DATA;
            EIMSK=0; //Ezalatt nem érdekelnek a külső interruptok
            sequenceRDY = 0;
            break;}
```

```
      case 'm': { //Mode
         switch(UART_sequence[1])
         {
            case 'm':{  //Measure
               mode = MEASURE;
               EIMSK=1;
               sequenceRDY = 0;
            break;}

            case 'p': { //ha p, megállítjuk a mérést
               mode = STOPPED;
               EIMSK=0; //Ezalatt nem érdekelnek a külső interruptok
               sequenceRDY = 0;
            break;}

            case 's': { //Ha s, alaphelyzetbe állunk, és megállunk.
               mode = STOPPED;
               EEPROM_ptr =DataToRead =IT_State =FIFO_wr_ptr =FIFO_rd_ptr =OVF_cnt =0;
               MAX30100_init();
               EIMSK=0; //Ezalatt nem érdekelnek a külső interruptok
               while(buff_rd(&tmp)); //kiürítjük a puffert
               sequenceRDY = 0;
            break;}

            default: { sequenceRDY = 0; break;}
         }
         break;}

      case 'i': {//info
         switch(UART_sequence[1])
         {
            case 'l': { //Élő idejű megjelenítés kapcsolása
               if(UART_sequence[2])
                  TCCR0B |= TIM0_PRESCALER_MSK;
               else
                  TCCR0B &= ~TIM0_PRESCALER_MSK;
               sequenceRDY = 0;
            break;}

            case 't': { //idő megjelenítés kapcsolása
               times=UART_sequence[2];
               sequenceRDY = 0;
            break;}

            case 'h': { //szívritmus megjelenítés kapcsolása
               hr=UART_sequence[2];
               sequenceRDY = 0;
            break;}

            default: { sequenceRDY = 0; break;}
         }
      break;}

      //Ha nem ismert parancsot kaptunk, nem csinálunk semmit,
      //csak várunk a következő parancsra.
      default: { sequenceRDY = 0; break;}
   }
}
```

2.3.7.3 Mérés mód

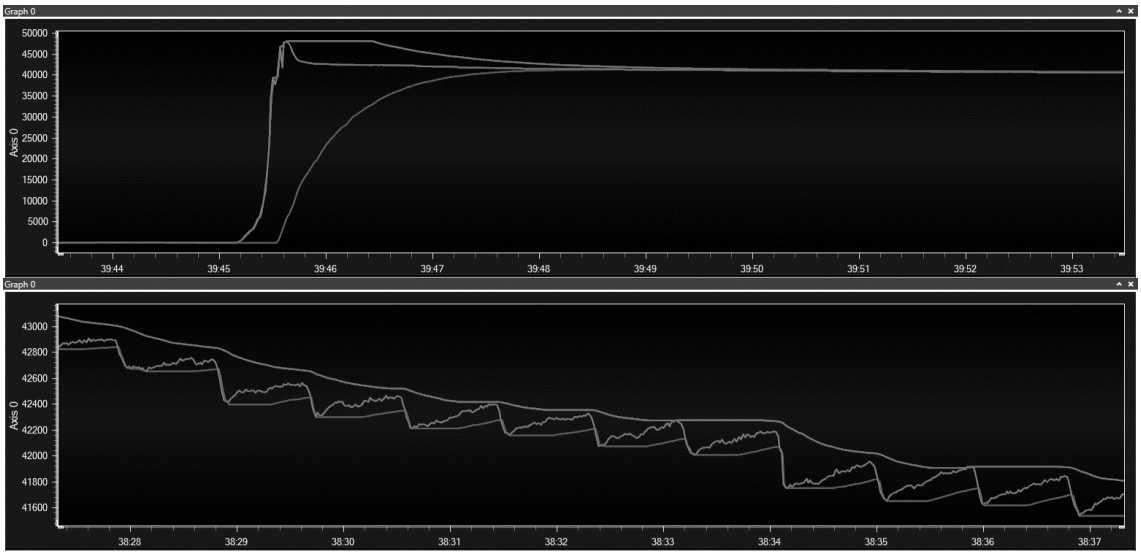
MEASURE módban a program feladata, hogy vizsgálja és feldolgozza a mintákat, és eldöntse, hogy átlépik-e a beállított küszöböt, amitől szívverésnek értelmezhető. Méri a szívverések közt eltelt időt, és beírja a memóriába.

Az algoritmus kidolgozásához szükség volt a mérő-chipből érkező jel vizsgálatára. Ehhez az Atmel Data Visualizer nevű programot használtam. Az adatokat az Atmel Data Stream Protocol [24] szerint, beállítható keretben, a bájtok megfelelő sorrendjével küldve a program képes élő időben megjeleníteni a mintát. Az alábbi két ábrán látszik, hogy a mérő-chip által küldött jel változása nagyon kicsi (fölső ábra), és a nullához képest változó mértékben van eltolva (alsó ábra).



25. ábra - Feldolgozatlan jel

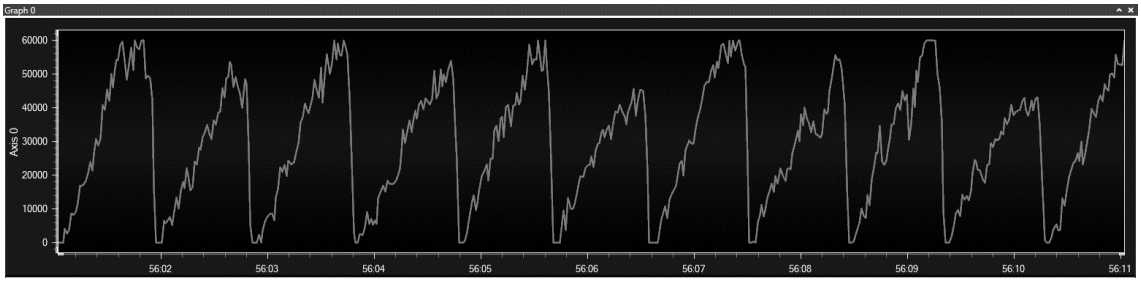
Ennek a problémának a kiküszöbölésére algoritmus követi a jel változását. Két változóval figyelem a jel minimumát és maximumát. Ha például a mért jel nagyobb a maximumnál, akkor a max változó új értéke a mért jel. Ezután a változó vár egy ideig, és ha ezalatt nem „ütközik” megint a jellel, akkor elkezd csökkenni, egészen addig, amíg nem lesz megint kisebb nála. Ugyanez igaz fordítva a minimumra is. A jel és a két határ alakulása az alábbi ábrán látható:



26. ábra - Minimum és maximum

A szívverés észlelés döntését alacsonyra állítottam, és akkor számít szívritmusnak, ha a jel az alatt van. Így biztosabb döntést tudtam elérni. Ehhez az a fontosabb, hogy a jel ez alá a határ alá mindenképp bemenjen minden szívverésnél, ezért követi a minimum kicsit gyorsabban a jelet.

Miután a program kiszámítja a két határt, a mért értéket átskálázza 0-tól 60000-ig a minimum és a maximum közti helyzete alapján: levonja a minimumot, hogy a jel 0-tól induljon, majd beszorozza $\frac{60000}{max-min}$-el, hogy a jel 60000-ig terjedjen. Az így kapott jel a következő:



27. ábra - Feldolgozott jel

Látható, hogy a jel sokkal szabályosabb, egy állandó döntési küszöböt, például 6000-ret beállítva jó biztonsággal detektálni tudjuk a szívverést.

Az algoritmus kódja a következő:

```
if(mode == MEASURE)  //Ha fut a mérés
      {
         if(buff_rd(&tmp)) //kiolvasunk egy adatot.
         {
            if(tmp<=min)   //Ha a jel kisebb, mint a minimum
            {
               min=tmp;    //akkor ez az új minimum
               mincnt=0;   //Elölről kezdjük a várakozást.
            }
            else if(mincnt>=40)  //Ha letelt a várakozási idő
            {
               {
                  min += (tmp-min)/60; //Csökkentjük a távolságot a jeltől
               }
            }
            else //Ha nagyobb a jel, de még nem vártunk eleget
            {
               mincnt++;   //Tovább várunk
            }

            if(tmp>=max)   //Ha a jel nagyobb, mint a maximum
            {
               max=tmp;    //Akkor ez lesz az új maximum
               maxcnt=0;   //Elölről kezdjük a várakozást.
            }
            else if(maxcnt>=80)  //Ha letelt a várakozási idő
            {
               {
                  max -= (max-tmp)/80; //Csökkentjük a távolságot a jeltől
               }
            }
            else //Ha kisebb a jel, de még nem vártunk eleget
            {
               maxcnt++;   //Tovább várunk
            }

            //Átszámoljuk a jelet az új minimummal és maximummal:
            tmp = (tmp-min)*(60000/(max-min));

            UART_send=tmp; //live stream küldéshez
```

Ezután figyelni kell a minta értékét, és ha 6000 alatt van, detektálni. Mivel a jel nem teljesen szabályosan változik, lehetséges, hogy a detektálás után még egyszer visszalép, és újra átlépi a küszöböt. Hogy ilyenkor ne számoljunk két szívritmust, a program figyeli, hogy elég idő telt-e el az előző szívritmus detektálás óta. Az idő mérésére a mikrokontroller TIMER1 16 bites időzítő áramköre szolgál. Ha megvan az idő adat, beírjuk a memóriába, és, ha a szívritmus, vagy idő kijelzés funkció be van kapcsolva, kiírjuk a megfelelő sztringet. A LED_ON és a LED_OFF két makró a fejlesztői kártyán található felhasználói LED állítására.

```
      if(tmp < 6000 )//Itt döntjük el, hogy átlépte-e a szintet,
            {        //ami alatt szívverésnek számít
               time = TCNT1L + (TCNT1H << 8);
               if(time>2343)  //Csak egy bizonyos idő elteltével tekintjük új szívverésnek
               {              //ez a szám 200-as pulzusnak felel meg.
                  if(!beat_ack ) //Ha ezt a szívverést még nem detektáltuk
                  {
                     LED_ON;  //Egy LED is jelzi a szívritmust
                     TCNT1=0; //Az időmérést elölről kezdhetjük
```

```
                        time = (((long int)time*128)/10000); //átszámítás 10 millisecundum-ra,
                        //és beírás a memóriába
                        TWI_start(WRITE, MEM_ADDRESS, EEPROM_ptr, 2, time);
                        EEPROM_ptr +=2;//A pointer léptetése a memóriában
                        if(times)   //Ha aktív ez a mód, kiírjuk az időt
                        {
                           transmit_char(time/100 + '0');
                           transmit_char('.');
                           transmit_char((time%100)/10 + '0');
                           transmit_char(time%10 + '0');
                           transmit_char('s');
                           transmit_char(' ');
                        }
                        if(hr)   //Ha aktív ez a mód, kiírjuk a szívritmust
                        {
                           unsigned int temp=6000/time;
                           transmit_char(temp/100 + '0');
                           transmit_char((temp%100)/10 + '0');
                           transmit_char(temp%10 + '0');
                           transmit_char('b');
                           transmit_char('p');
                           transmit_char('m');
                           transmit_char(' ');
                        }
                        //Ha elértük a memória tetejét, elölről kezdjük, de megállunk
                        if(EEPROM_ptr>=EEPROM_ADDRESS_LIMIT)
                           {
                              EEPROM_ptr=0;
                              mode=STOPPED;
                           }
                        beat_ack=1;    //Jelezzük, hogy ezt a szívverést detektáltuk.
                     }
                  }
               }
               else     //Ha az adat kisebb, mint a döntési küszöb
               {
                  beat_ack=0;    //Jelezzük, hogy új szívverést várunk.
                  LED_OFF; //Kialszik a LED
               }
            }
         }
```

COLLECT_DATA módban megállunk, és lekérdezzük a memóriát egyszer. Nem állítunk semmit alap állapotba, hátha megint le akarják kérdezni, vagy folytatni szeretnék a mérést. Az UART-on való küldés automatikusan zajlik, a DataHandler() függvényben, ami sikeres olvasás esetén hívódik meg. Mivel a mikrokontroller órajele sokkal gyorsabb, mint az I2C kommunikáció, az I2C kommunikációkat tároló puffer megtelne, és valahány adatot nem küldenénk el. Ezért minden küldés előtt megvárjuk, hogy legalább egy szabad hely legyen a pufferben.

```
      else if(mode == COLLECT_DATA) //Ha lekérdezték a memóriát
      {
         mode = STOPPED;   // Megállunk.
         //Végigmegyünk a memórián,
         for(long int i=0; i<EEPROM_ptr; i+=2)
         {
            while(TWI_wr==TWI_rd && TWI_isempty==0);  //Várunk a puffer üresedésére
            TWI_start(READ, MEM_ADDRESS, i, 2, 0);  //és kiolvassuk.
         }
      }
   }
}
```

3 Összefoglalás

Munkám során megismerkedtem a szívritmus mérésére alkalmazott modern módszerekkel, elsősorban a fényelnyelésen alapuló, pletizmografikus módszerrel. Betekintést nyertem pletizmográfiához használt legmodernebb cél-chipek működésébe, és alkalmazásába. Fejlesztettem a beágyazott programozással, és mikrokontrollerek kezelésével kapcsolatos gyakorlatomat. A munka hasznomra vált abban, hogy elsajátítsak bizonyos áramkör tervezési ismereteket, és rávilágított arra, milyen szempontokat érdemes figyelembe venni a mikrokontroller alapú rendszerek tervezésénél.

Az irodalomkutatás után megterveztem egy szívritmus mérésére, tárolására, és PC-nek való továbbítására alkalmas hardvert, és a nyomtatott áramkör tervezés alapjainak elsajátítása után nyomtatott áramköri tervet készítettem, mely alapján az eszköz legyártható. Megírtam egy szoftvert, amelyet futtatva egy mikrokontroller képes irányítani az említett folyamatot, és felhasználói parancsokra is reagál. Az elkészült eszközzel méréseket végeztem, és algoritmust dolgoztam ki a mért jel hatékony feldolgozására, és a szívritmus detektálására.

A projektnek több lehetséges továbbfejlesztési lehetősége van. Az eszköz hordozhatósága fontos szempont. Ehhez szükség van az áramfogyasztás minimalizálására. A jelfeldolgozó algoritmus továbbfejleszthető, pontosítható a megbízhatóbb működés érdekében. Esetleg a számítógéppel vagy okostelefonnal történő kényelmes, vezeték nélküli kommunikáció is megvalósításra kerülhetne, amely valamilyen felhasználói interfész elkészítésével még kényelmesebbé tehető.

Irodalomjegyzék

[1] Atmel: *Atmel 8-bit microcontroller with 4/8/16/32 kBytes in-system programmable flash – datasheet,* *http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf*

[2] Atmel: *ATmega328P Xplained Mini – user guide*, http://www.atmel.com/Images/Atmel-42287-ATmega328P-Xplained-Mini-User-Guide_UserGuide.pdf

[3] Atmel: *ATmega328P Xplained Mini – design documentation*, http://www.atmel.com/tools/mega328p-xmini.aspx?tab=documents

[4] Maxim Integrated Products, Inc.: *MAX30100 Pulse Oximeter and Heart-Rate Sensor IC for Wearable Health*, *https://datasheets.maximintegrated.com/en/ds/MAX30100.pdf*

[5] Diodes Incorporated: *AP1084 – 5A Low droput positive adjustable or fixed-mode regulator, http://www.diodes.com/_files/datasheets/AP1084.pdf*

[6] Microchip Technology Inc.: *24AA1025/24LC1025/24FC1025 – 1024K I2C™ Serial EEPROM*, *http://ww1.microchip.com/downloads/en/DeviceDoc/20001941L.pdf*

[7] Embedded-Lab (by R-B): *PC-based heart rate monitor using Arduino and Easy Pulse sensor*, http://embedded-lab.com/blog/pc-based-heart-rate-monitor-using-arduino-and-easy-pulse-sensor/

[8] Sanjeev Kumar, Cypress Semiconductor Corp.: *Efficient Heart Rate Monitoring*, *http://www.cypress.com/file/106976/download*

[9] Dr. Boros Mihály: *Orvostechnika és monitorozás – Gyakorlati orvosi alapismeretek*, http://web.szote.u-szeged.hu/expsur/rop/aktualis_tananyag/konyvek/Orvostechnika%20es%20monitorozas.pdf

[10] Wikipédia: *Elektrokardiográfia,* *https://hu.wikipedia.org/wiki/Elektrokardiogr%C3%A1fia*

[11] Wikipedia: *Photoplethysmogram*, *https://en.wikipedia.org/wiki/Photoplethysmogram*

[12] Texas Instruments: *AFE4400 Integrated Analog Front-End for Heart Rate Monitors and Low-Cost Pulse Oximeters*, http://www.ti.com/lit/ds/symlink/afe4400.pdf

[13] PixArt Imaging Inc.: *PAH8001EI-2G Optical Heart Rate Detection Sensor*, http://www.szgsensor.com/uploads/soft/141229/%D4%AD%CF%E0%B9%E2%

B8%D0%B4%F8%D0%C4%C2%CA%CB%E3%B7%A8%D2%E5%BC%CE.pdf

[14] Analog Devices Inc.: *Single-Lead, Heart Rate Monitor Front End – Data Sheet AD8232*, *http://www.analog.com/media/en/technical-documentation/data-sheets/AD8232.pdf*

[15] New Japan Radio Co., Ltd.: *NJL5501R – COBP PHOTO REFLECTOR with RED & IR LED*, http://www.njr.com/semicon/PDF/NJL5501R_E.pdf

[16] OSRAM Opto Semiconductors: *BioMon Sensor Datasheet*, http://www.osram-os.com/Graphics/XPic7/00187722_0.pdf/SFH%207050,%20Lead%20(Pb)%20Free%20Product%20-%20RoHS%20Compliant.pdf

[17] MikroElektronika: *Heart rate click manual*, http://download.mikroe.com/manuals/click/heart-rate/heart_rate_click_manual.pdf

[18] Texas Instruments: *TPS61097A-33 Low-Input Voltage Synchronous-Boost Converter With Low Quiescent Current*, http://www.ti.com/lit/ds/symlink/tps61097a-33.pdf

[19] Fitbit: *Charge HR Wireless Activity Tracker Wristband*, https://www.fitbit.com/eu/chargehr

[20] Mio: *Alpha 2 Heart Rate Sport Watch*, http://www.mioglobal.com/EN-UK/Mio-ALPHA-2-Heart-Rate-Watch/Product.aspx

[21] Garmin: *Forerunner 225 GPS Strapless Heart Rate Monitor*, https://buy.garmin.com/en-US/US/into-sports/running/forerunner-225/prod512478.html

[22] Zencro: *HRM-2105 Heart Rate Monitor*, http://zencro.en.alibaba.com/product/60014256559-220952431/Earlobe_Clip_Heart_Rate_Monitor_HRM_2105.html

[23] GP Batteries: *GP100AAAHC – Datasheet*, http://cellpacksolutions.co.uk/online-shop/Products/ProductFILES/ProductPDFs_TDS/GP100AAAHC-U4.pdf

[24] Atmel: *Data Visualizer User Guide - Data stream*, http://www.atmel.com/webdoc/dv/ch11s01.html