Henri Tamm

Foucault’ kardiograafi haldurarvuti prototüüp

Bakalaureusetöö (12 EAP)

Arvutitehnika eriala

Juhendaja:

PhD Vahur Zadin

Tartu 2017
Resümee/Abstract

Foucault’ kardiograafi haldurarvuti prototüüp

Käesoleva bakalaureusetöö eesmärgiks oli valmistada uus Foucault’ kardiograafi haldurarvuti prototüüp, mis oleks kaasaegne ning mugav kasutada. Haldurarvuti peab suutma kuvada reaalajas graafikut ning salvestada mõõdetud mõõdetud andmeid.

Töös kirjeldatakse valitud riistvara, tarkvara arendamiseks kasutatud teeke, andmebaasi valikut ning programmi dokumenteerimist. Peale selle seletatakse lahti programmi töö.

CERCS: T120 Süsteemitehnoloogia, arvutitehnoloogia; P175 Informaatika, süsteemiteorioria;

Märksõnad: Foucault’ kardiograaf, haldurarvuti, prototüüp

Master computer prototype for Foucault Cardiograph

The aim of this thesis was to construct a new Foucault cardiograph master computer prototype, that would be modern and easy to use. The master computer has to be able to display real-time graphs and save the measured data.

The thesis describes the selected hardware, librarys used for the software development, database selection and documentation of the program. Furthermore, the programs work is explained.

CERCS: T120 Systems engineering, computer technology; P175 Informatics, systems theory;

Märksõnad: Foucault cardiography, master computer, prototype
Sisukord

Resüüm/Abstract .................................................................................................................. 2

Lühendid ja mõisted .............................................................................................................. 5

1 Sissejuhatus .................................................................................................................... 6

2 Valdkonna ülevaade ....................................................................................................... 7

2.1 Elektrokardiograafia (EKG) ......................................................................................... 8

2.2 Foucault’ kardiograafia (FouKG) ................................................................................ 8

2.3 Tartu Ülikoolis hetkel kasutatav Foucault’ kardiograafi haldurarvuti ....................... 9

2.4 Töö eesmärk .................................................................................................................. 9

3 Haldurarvuti prototüüp .................................................................................................. 11

3.1 Riistvara ....................................................................................................................... 11

3.1.1 Arendusplaat .......................................................................................................... 11

3.1.2 Raspberry Pi ........................................................................................................... 11

3.1.3 Raspberry Pi puutetundlik ekraan ........................................................................ 12

3.1.4 Adafruit Powerboost 1000C ................................................................................ 13

3.1.5 Aku ......................................................................................................................... 13

3.1.6 Adafruit 16-bit ADC ADS1115 ............................................................................. 14

3.2 Tarkvara ....................................................................................................................... 14

3.2.1 Python .................................................................................................................... 15

3.2.2 Pycharm .................................................................................................................. 15

3.2.3 Kivy teek ............................................................................................................... 15

3.2.4 Kivy Garden ja Kivy Graph ................................................................................... 16

3.2.5 Kivy language ...................................................................................................... 16

3.2.6 Kasutajaliides ....................................................................................................... 16

3.2.7 SQLite ....................................................................................................................... 17

3.2.8 Doxygen ................................................................................................................ 17

4 Tehtud töö ......................................................................................................................... 18
4.1 Riistvara struktuur ................................................................. 18
4.2 Koodi struktuur ja klassid ..................................................... 19
4.3 Programmis kasutatavad vidinad .......................................... 20
4.4 Peaakna disain ja nupud ....................................................... 21
4.5 Graafik .................................................................................. 22
4.6 Kõrvalakna disain .................................................................. 23
4.7 Andmebaas .......................................................................... 24
4.8 Doxygen kommenteerimine .................................................. 25
4.9 Olulisimad verstapostid ning nende ületamine ..................... 26
5  Arutelu ja järeldused .................................................................. 28
   5.1 Arutelu .................................................................................. 28
   5.2 Järeldused ............................................................................. 28
6  Kokkuvõte ................................................................................ 29

Viited .......................................................................................... 30

Lisa 1. Kardiograafi programm .................................................. 32
Lisa 2. Kardiograafi kasutajaliidese Kivy fail ............................. 43
Lihtlisents ................................................................................ 55
Lühendid ja mõisted

ADC (engl analog-to-digital converter) – analoog-digitaalmuundur.

DSI (engl Display Serial Interface) – kuvari jadaliides.

EhhoKG (engl echocardiography) – ehk hokardiograafia.

EKG (engl electrocardiography) – elektrokardiograafia.

FouKG (engl Foucault cardiography) – Foucault’ kardiograafia.

GPIO (engl general-purpose input/output) – üldotstarbelised sisend/väljund viigud.

GPU (engl graphics processing unit) – graafikaprotsessor.

GUI (engl graphical user interface) – graafiline kasutajaliides.

HDMI (engl High-Definition Multimedia Interface) - kõrglahutusega multimeediumiliides.

I2C (engl Inter-Integrated Circuit) – kahejuhtmeliides, kahesuunaline kahesooneline järjestiksiin.

IDE (engl integrated development environment) – integreeritud programmeerimiskeskkond.

KT (engl computed tomography) – kompuutertomograafia.

LED (engl light emitting diode) – valgusdiood.

MRT (engl magnetic resonance imaging) – magnetresonantstomograafia.

OS (engl operating system) – operatsioonisüsteem.

SDRAM (engl synchronous dynamic random-access memory) – sünkroonne dünaamiline muutmüalu.

USB (engl Universal Serial Bus) – universaalne järjestiksiin.
1 Sissejuhatus


Südame mehaaniliste protsesside jälkimiseks on võimalik kasutada ka Foucault’ kardiograafiat, mis põhineb mitteinvassiivsel impedantsmeetodil. Võrreldes elektrokardiogrammiga, on Foucault’ kardiogrammi mõõtmiseks võimalik arendada palju väiksemaid ning odavama hinnaga seadmeid. Taolised seadmed võib jagada saatjaplokiks, vastuvõtuplokiks ning haldurarvutiks. Käesolevas töös luuakse uus ja kaasaegne Foucault’ kardiograafi haldurarvuti prototüüp.
2 Valdkonna ülevaade


EKG ehk elektrokardiograafia on enimkäsin kasutatav südame jälgimise meetod. EKG on kiire meetod, mis võimaldab määrata südame löögisagedust ning erinevaid südamerütmihäireid, kuid vajab samuti arsti järelevalvet. [2]

Foucault’ kardiograafia ehk FouKG, mis põhineb elektrilisel bioimpedantsmeetodil, võiks aga kujutada varianti, kus patsient saab vajalikud mõõtmised ise ise. Kuna Foucault’ kardiograafia on mugav mõõtmismeetod, võimaldaks see ka pikaajalist südame monitoorimist. Seetõttu võiks FouKG-l põhinev seade leida rakendust mitte ainult meditsiini valdkonnas, vaid seda saaksid südame löögimahu ning löögisageduse jälgimiseks igapäevaselt kasutada ka näiteks sportlased ja sõjaväelased.
2.1 Elektrokardiograafia (EKG)

2.2 Foucault’ kardiograafia (FouKG)

Foucault’ kardiograafia nimi tuleb prantsuse füüsiku ning esimese pöörisvoolude uurija Léon Foucault nimest.

Joonis 1 Iseloomulik kardiogrammi kuju. [4]

Joonis 2 Iseloomulik Foucault’ kardiogrammi kuju. [5]
2.3 Tartu Ülikoolis hetkel kasutatav Foucault’ kardiograafi haldurarvuti

Tartu Ülikoolis kasutatakse hetkel Foucault’ kardiograafi haldurarvutina IBM ThinkPad 500, mis pärineb aastast 1997. Haldurarvuti on vana ning ei vasta tänapäevastele riistvara standarditele (Tabel 1).

Tabel 1. IBM ThinkPad 500 tehnilised andmed. [6]

<table>
<thead>
<tr>
<th>IBM ThinkPad 500</th>
<th>Protsessor</th>
<th>Mälu</th>
<th>Püsimälu</th>
<th>OS</th>
<th>Ekraan</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 486 SLC2 50 MHz CPU</td>
<td>4MB RAM</td>
<td>80 – 170 MB</td>
<td>Windows 3.1</td>
<td>7,24 tolli, eraldusvõime 640x480</td>
<td></td>
</tr>
</tbody>
</table>

Ka haldurarvuti programm on loodud aastal 1997. Programmi kasutajaliides kuvab kõik vajalikud andmed ning laseb muuta graafiku sätteid, kuid paljude nuppude ja tekstiväljade tõttu näeb vana kasutajaliides välja keeruline ning ei sobi tavakasutajale (Joonis 3).

Joonis 3 Foto vana haldurarvuti kasutajaliidesest.

2.4 Töö eesmärk

Käesoleva bakalaureusetöö eesmärgiks on valmistada Foucault’ kardiograafi uus ja kaasaegne haldurarvuti prototüüp. Vörreldes vana haldurarvuti versiooniga, rõhutakse uue haldurarvuti
tegemisel kasutajamugavusele ning kuna tegemist on potentsiaalse tootega, peaks haldurarvuti modernne välja nägema.

Püstitatud eesmärgi saavutamiseks oli tarvis lahendada järgmised ülesanded:

1. leida sobiv riistvara haldurarvuti jaoks;
2. disainida lihtne ja konkreetne kasutajaliides;
3. luua kasutajaliidesele reaalajas töötav graafik mõõdetud andmete kuvamiseks;
4. luua mõõdetud info salvestamiseks tarkvaraga suhtlev andmebaas.
3 Halduravuti prototüüp

3.1 Riistvara
Foucault’ kardiografi halduravuti riistvara valimisel lähtutis asjaolust, et halduravuti oleks mobiilne ning mugav kasutada. Järgnevalt on kirjeldatud kõik halduravuti riistvara osad.

3.1.1 Arendusplaat


<table>
<thead>
<tr>
<th></th>
<th>Raspberry Pi 3 Model B</th>
<th>Arduino Mega</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protsessor</td>
<td>1.2 GHz 64-bit nelja-tuupaline ARM Cortex-A53</td>
<td>16 MHz ATmega1280</td>
</tr>
<tr>
<td>Mälu (SDRAM)</td>
<td>1 GB (jagatud GPU-ga)</td>
<td>8 KB</td>
</tr>
<tr>
<td>Välkmälu</td>
<td>MicroSD kaart</td>
<td>128 KB</td>
</tr>
<tr>
<td>USB 2.0 portide arv</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>Võrguühendused</td>
<td>802.11n Wireless LAN, Bluetooth 4.1</td>
<td>-</td>
</tr>
</tbody>
</table>


3.1.2 Raspberry Pi
Raspberry Pi on võike krediitkaardi-suurune arvuti, mille arendaja on Raspberry Pi Foundation. [7] Kässoleva seadme jaoks valiti Raspberry Pi 3 Model B (Joonis 4), mis kasutab Broadcom’i süsteemikiipi BCM2837, milles on nelja-tuupaline 1,2 GHz 64-bit ARM Cortex-A53
protsessori. Raspberry Pi’l on 1 GB SDRAM-i ( ingl synchronous dynamic random-access memory), mis on jagatud GPU-ga ( ingl graphics processing unit). [8]


![Joonis 4 Raspberry Pi 3 Model B koos 16 GB MicroSD kaardiga.](image)

### 3.1.3 Raspberry Pi puutetundlik ekraan

3.1.4 Adafruit Powerboost 1000C


3.1.5 Aku

Seadme mobiilseemaks kasutamiseks on Powerboost 1000c külge ühendatud aku (Joonis 7), mis võimaldab halduravuti toimist, kui läheduses puudub vooluvõrk. Aku on mahtuvusega 4400 mAh ning väljundpingega 3,7 V.
Aku mahtuvusega 4400 mAh.

3.1.6 Adafruit 16-bit ADC ADS1115
Välismaailmast sisendi saamiseks on Raspberry Pi külge ühendatud Adafruit’i analoog-digitaalmuundur (Joonis 8). Adafruit’i ADS1115 ADC tagab 16-bitise täpsusega 860 diskreeti sekundis kasutades I2C (ingl Inter-Integrated Circuit) tehnoloogiat. Kiibi saab konfigureerida neljaks taavaliseks (single-ended) sisendiks või kaheks diferentsiaalseks sisendiks. Lisaks on kiibi peal ka võimendi, mis võimaldab signaali võimendada kuni 16 korda. [12]

3.2 Tarkvara
Kuna tegemist on seadme prototüübiga, valiti tarkvara programmeerimiseks Pythoni keel, millega saaks suuremas osas aru ka inimene, kes näeb koodi esimest korda. Kogu kood on kirjutatud objektorienteeritult, sest nii on programm loogiliselt tükeldatud (klassideks ja meetoditeks) ning hiljem on koodi lihtne parandada ja edasi arendada. Koodi arendati programmeerimiskeskkonnas Pycharm. Graafiline kasutajaliides on programmeeritud kasutades Kivy teeki.
3.2.1 Python


Käesolevas projektis kasutati Pythoni versiooni 2.7.12, sest seda toetab Kivy teek ning vörreldes Python 3.5 on võimalik tõrgete tekkimine väiksem.

3.2.2 Pycharm


3.2.3 Kivy teek


Kivy on võimas avatud lähtekoodiga teek, mis on spetsiaalselt mõeldud puutetundlike ekraanide GUI (ingl graphical user interface) programmeerimiseks. [16] Kuna kasutajaliidese kuvamiseks kasutatakse Raspberry Pi puutetundliku ekraani, sobib just Kivy teek kõige paremini, sest sellega saab kasutada multitouch funktsioone ning joonistada reaalajas graafikuid.

Kivy rakenduse loomiseks on vaja teha kolme asja:

- Luua App klassi alamklass;
- Defineerida selle klassi build() meetod nii, et see tagastaks vidinapuu juure;
- Kutsuda välja selle klassi run() meetod. [17]
Käesolevas programmis loodi *App* klassi alamklass nimena *Cardiograph_App* ja defineeriti selle `build()` meetod nii, et see laeb Kivy klee faili „cardiograph_app_gui.kv“ ning tagastab selle juurvidina. Programmi käivitamisel kutsutakse välja meetod *Cardiograph_App.run()*. Joonisel 9 on kujutatud koodilõik Kivy rakenduse käivitumise kohta.

```python
class Cardiograph_App(App):
    def build(self):
        return Builder.load_file("gui.kv")

if __name__ == "__main__":
    Cardiograph_App().run()
```

Joonis 9 Lõik koodist kardiograafi programmite käivitumise kohta.

### 3.2.4 Kivy Garden ja Kivy Graph
Üheks Kivy alamprojekti on Kivy Garden, mis on kogum erinevate kasutajate poolt tehtud teekitest ja vidinatest (ingl *widget*). Kõik Kivy Garden’i asuvad vidinad on vabalt kasutatavad.
Käesolevas projektis kasutati Kivy Garden’i vidinat Kivy Graph, mis võimaldab reaalajas graafikute kuvamist. [18] [19]

### 3.2.5 Kivy language
Aja jooksul muutuvad suured programmid aina keerulisemaks ning kasutajaliidese ja programmi loogika vaheline koostöö võib muutuda väga segaseks. Selle jaoks on välja töötatud Kivy keel, mis võimaldab programmeerijal lihtsalt ja loogiliselt kujundada graafilist kasutajaliidest hoides seda samal ajal programmeerijale täielikult eraldatuna. See tagab arusaadava koodi struktuuri ning lubab programmeerijale kasutajaliidest lihtsalt muuta. [20]


### 3.2.6 Kasutajaliides
Kogu kasutajaliides on eraldiseisvalt defineeritud Kivy (Kv) failis „cardiograph_app_gui.kv“. Kivy fail on konstrukteeritud vidinapuust (ingl *widget tree*), mis koosneb juurvidinast (ingl *root widget*) ning selle alamvidinatest. Teisisõnu on failis kirjeldatud kõik kasutajaliidese osad üksteisele kuuluvuse kaudu. Kardiograafi juurvidinaks on ise loodud klass *Logic*, mis asub peameetodis ning juhib kogu programmloogikat.

BoxLayout seab enda alamvidinad horisontaalselt või vertikaalselt üksteise otsa. Kuna BoxLayout täidab oma alamvidinatega terve talle määratud ala, ei ole vidinate asukohta tarvis määrata.

FloatLayout võimaldab programmeerijal alamvidina asukoha ning suuruse määrata ilma piiranguteta (n-ö ujuvalt). Seetõttu on FloatLayoutile alamvidinate määruseks oluline defineerida vidina täpne asukoht (vöi suhteline paiknmine akna raamistikus) ning suurus.

3.2.7 SQLite

3.2.8 Doxygen
4 Tehtud töö

4.1 Riistvara struktuur


Joonis 10 Foucault’ kardiograafi haldurarvuti riistvara.
4.2 Koodi struktuur ja klassid


Näiteks, kui kasutaja vajutab nuppu „STOP“, kutsub failis „cardiograph_app_gui.kv“ defineeritud vidin välja klassi Logic meetodi stop(), mis peatab klassi Cardiograph isendi.
kaudu graafiku kuvamise ning salvestab mõõdetud andmed läbi Profile_Database klassi isendi andmebaaseni.

```
if __name__ == "main":
    Cardiograph_App().run()
```

![Diagram](diagram.png)

**Joonis 12** Koodi töö ning klasside struktuur.

### 4.3 Programmis kasutatavad vidinad


Mõlemas aknas on kasutusel mitmeid nuppe (ingl Button), silte (ingl Label) ning lüliteid (ingl ToggleButton). Peale selle on kõrvalaknas kasutusel mitu hürikakent (ingl Popup), tekstiväli...
(engl TextInput), raadionupp (engl Checkbox) ning liigur (engl Slider), millega saab määrama puhvri suurust. Joonisel 13 on kujutatud lõik kasutajaliidest defineerivast vidinapuust.

```python
Logic:
    ScreenManager:
        id: sm
    Screen:
        name: 'screen1'
        BoxLayout:
            id: bl
            orientation: "vertical"
            FloatLayout:
                size_hint: [1, .8]
                Graph:
                    pos_hint: {"center_x": 0.5, 'center_y': 0.5}
                    id: graph
                    xlabel: "Time (ms)"
                    x_ticks_major: 1000
                    x_ticks_minor: 5
                    x_grid: True
                    x_grid_label: True
```

Joonis 13 Näide Kivy vidinapuust koos juurvidinaga Logic ja selle esimene alamvidinaga ScreenManager.

### 4.4 Peaakna disain ja nupud


„START/STOP“ nupu esmakordsel vajutamisel käivitab seade mõõtmised ning hakkab neid kuvama graafikul. Teisel vajutusel peatab seade mõõtmised ja jätab graafiku seisvasse olekusse.

„CALIBRATE“ nupuga ühendatud funktsioon `calibrate()` leiab hetkel kuvatava graafiku maksimaalse ning minimaalse väärtuse, suurendab maksimaalset ja vähendab minimaalset väärtust 800 võrra ning saab saadud väärtused graafiku ordinaattelje maksimaalseks ning minimaalseks väärtuseks. Nii jääb nuppu vajutades graafik ekraani keskelle ning on täies ulatuses kasutajale näha. „CALIBRATE“ nupu kõrval asub väiksem automaatse kalibreerimise
nupp „AUTO: ON/OFF“, mille sisse lülitamisel kalibreerib programm graafikut automaatselt määratud intervalli tagant.

„MORE“ nuppu vajutades kuvatakse kasutajale teine aken, kus on võimalik hallata profiile ning muuta graafiku seadeid.

4.5 Graafik
Graafiku kuvamiseks kasutati Kivy Garden’ist pärit Graph klassi. Klassi kasutamiseks on projekti kausta lisatud fail Graph.py ning seejärel imporditud see peaklassis. Graafiku tegemiseks loodi kõigepealt Kv failis Graph objekt, millele määrati ID, x- ja y-telje nimi, suurte ja väikeste graafikusegmentide suurused ning x- ja y-telje esialgset maksimum ja miinimum väärtsed. Graph objekt töötab antud juhul justkui lõuendina, kuhu peale on võimalik joonistada mitmeid graafikuid. Kuigi väliselt tundub, et käesolevas programmis kuvatakse ainult ühte graafikut, on kuvatavaid graafikuid tegelikult kaks.

Peale lõuendi defineerimist loodi peaklassis MeshLinePlot objekt ning lisati see Graph objektile. MeshLinePlot on objekt, mis joonistab lõuendile rea punkte ning ühendab need punktid joontega. Punktid antakse ette paaride listina, kus iga paar sisaldab endas punkti x- ja y-koordinaati. [19] Näiteks lisatakse kardiograafi programmis uus koordinaadipaar iga kord kui soovitakse graafikut uusi andmeid kuvada. Koordinaadipaari x-koordinaadiks on mõõdetud andmete ajahetk ning y-koordinaadiks andmete väärtsus.
Kuvatav graafik liigub pidevalt ajas vasakult paremale ning kardiograafile omase efekti saavutamiseks ei kustutata graafiku paremale serva jõudes kogu graafikujoont, vaid jäetakse n-ö vana joon alles ja joonistatakse see vasakult paremale liikudes uuesti üle. Üle joonistamine ei toimu kohe, vaid vana ja uue graafikjuone vahel on tühij ala. Selle tõttu on graafikul alati kaks joont, üks vasakul ja teine paremal pool tühja ala.

Kuna mõõdetavatel andmetel võib alati esineda müra, on graafiku kuvamisel võimalik kasutada puhvrit, mis leib järjestikustega andmete keskmise väärtsuse. Puhvri suurus on võimalik muuta kõrvalaknas liuguriga. Mida suurem on puhver, seda rohkem andmeid silutakse ning seda lamedam on graafik.

4.6 Kõrvalakna disain


Nupp „Delete profile“ kustutab hetkel aktiivsena oleva profiili. Kui alles on jäänud ainult üks profiil ja kasutaja soovib ka seda kustutada, kuvatakse hüpiakas vastav veateade. See on vajalik, sest alati peab eksisteerima vähemalt üks profiil, millele mõõtmisel andmed salvestatakse. Kui
kasutaja soovib ikkagi valitud profiili kustutada, on võimalik luua uus profiil ja koheselt kustutada vana.


![Image](image.png)

**Joonis 15** Kärvalaken aktiivse profiili ja puhvri suurusega 3.

### 4.7 Andmebaas


Joonis 16 Andmebaasi struktuur.

Kolme tabeliga andmebaas loob loogilise andmete struktuuri. Profiilid, mõõtmised ning mõõdetud andmed on üksteisest eraldatud, tänu millele on andmebaasist lihtne pärida soovitud infot. Kasutajaliidesest endast ei ole võimalik vanu mõõtmisi kuvada, kuid eelnevatel teadmiste põhjal on andmebaasist käsite päringute tegemisel võimalik saada järgmist informatsiooni:

- Kõik andmebaasis olevad profiilit.
  ```sql
  SELECT * FROM Profiles;
  ```

- Kõik mõõtmised ühe profiili kohta, kus `profile_Id` on vaadeldava profiili ID.
  ```sql
  SELECT * FROM Measurements WHERE profile_id=(profiili ID);
  ```

- Kõik andmed ühe mõõtmise kohta, kus `measurement_id` on vaadeldava mõõtmise ID.
  ```sql
  SELECT * FROM Data WHERE measurement_id=(mõõtmise ID);
  ```

Juhul kui rakenduse käivitumisel andmebaasi ei eksisteeri, loob programm uue andmebaasi. Kuna alati peab eksisteerima vähemalt üks profiil, kellele andmeid salvestatakse, lisatakse pärast andmebaasi loomist tabelisse `Profiles` üks profiil vaikimisi.

4.8 Doxygen kommenteerimine

Joonis 17 Näide kommenteeritud meetodist ja Doxygeni genereeritud dokumentatsioonist.

4.9 Olulisimid verstapostid ning nende ületamine


Arvestades eelnevaid vigu, tuleks välisele seadmele programmi arendades tihedalt katsetada koodi ka seadme enda peal ning mitte arendada koodi ilma testimata liiga pikalt arenduskeskkonnas.

Profili nime muutmiseks ning kasutajale erinevate veateadete kuvamiseks kasutatakse programmis hüpikaknait (ingl Popup). Arendamise käigus selgus, et Kivy teegil eksisteerib

```python
def buffer_value(self):
    # Method that calculates the average value in the buffer.
    total_value = 0
    for value in self.cardiograph.buffer:
        total_value += value
    return int(total_value / len(self.cardiograph.buffer))

buffer_value()
```

```
def cardiograph.Cardiograph.buffer_value(self):
    # Method that calculates the average value in the buffer.
```
programmiviga, kus korduval hüpirakna avamisel ning sulgemisel viidatakse vanale, n-õ surmud hüpiraknale, mis põhjustab programmi sulgemise. Kuna antud viga ei suudetud lahendada, on kõik hüpirakna avamise käsud ümbritsetud veatuvastus plokiga.

Kuna Tartu Ülikoolis arendatava Foucault’ kardiograafi andur on vana ning puudub töökindlus, kasutati halduravutis graafiku kuvamiseks eelnevalt mõõdetud andmeid. Haldurarvuti arendamisel ei ole otse andurist signaalide mõõtmine vajalik, sest andmete mõõtmist on võimalik simuleerida.
5 Arutelu ja järelused

5.1 Arutelu

Haldurarvuti prototüüpi on võimalik edasi arendada mitmes aspektis. Salvestud infot saaks salvestada võrgus asuvasse andmebaasi. See tagaks andmete kindlama säilimise ning mitme seadme puhul saaks kasutaja mõõdetud andmeid salvestada ühe profiiliga. Antud töös kuvatakse graafikuid ainult eelnevalt mõõdetud andmete põhjal, kuid edasi võiks sisendiks hakata võtma reaalajas mõõdetavat signaali, milleks on tellitud ka analoog-digitaalmuundur. Haldurarvutile võiks disainida uue korpuse, mis ühendaks endas Raspberry Pi koos ekraaniga, Powerboost 1000c toiteskeemi, aku ning ADC.

5.2 Järelused

Käesoleva tööga saavutati kõik eesmärgiks seatud ülesanded:

1. haldurarvuti prototüüp töötab uue riistvara peal ning kasutajaga suhtlus toimub nüüd läbi puutetundliku ekraani;
2. seadmene on disainitud uus kasutajaliides;
3. tarkvara suudab reaalajas kuvada graafikut ning optimeerida ja siluda seda vastavalt vajadusele;
4. haldurarvuti prototüüp suudab profiilipõhiselt salvestada mõõdetud info andmebaasi.

Haldurarvuti prototüüp töötab ootuspärastelt ning on valmis edasiseks arenduseks Foucault’ kardiograafi üldises plaanis.
6 Kokkuvõte

Käesolev bakalaureusetöö kirjeldab Foucault’ kardiograafi uue halduravuti prototüübi disaini ning tarkvaralist lahendust.

Töö eesmärgiks oli valmistada halduravuti prototüüp, mida oleks mugav kasutada ning mis näeks modernne vältja. Eesmärgi saavutamiseks oli tarvis leida halduravuti jaoks sobiv riistvara, disainida kasutajaliides koos töötava graafikuga ning luua andmete salvestamiseks profiilipõhine andmebaas.


Tehtud halduravuti on osa arendamisel olevast Foucault’ kardiograafist.
Viited


Lisad

Lisa 1. Kardiograafi programm

This application shows data measured by the Foucault' cardiograph. All measured data is saved in a database according to the selected profile.

Created by Henri Tamm.

```
import kivy
kivy.require('1.9.1')
from kivy.core.window import Window
from kivy.lang import Builder
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
import Graph
from kivy.clock import Clock
from kivy.uix.label import Label
from kivy.uix.checkbox import CheckBox
from math import ceil, floor
import sqlite3

## Raspberry Pi screen size.
# Necessary for running the application in other platforms than Raspberry Pi.
Window.size = (800, 480)

## Previously measured data file location.
data_file_location = "C:\Users\SUPER\Documents\Bakatoo\Vedru_failid\M021625A_MaertB.txt"

class Cardiograph:
    
    # The constructor.
def __init__(self):
        self.present_data = 0
        self.measured_data = []
        self.data_dict = {}
        self.end_time = 0
        self.begin_time = 0
        self.current_time = 0
        self.passed_time = 0
## Graph type selection. 0 - FouKG; 1 - EKG
self.graph_type = 1
self.read_file_to_list()

## Buffer for smoothening the graph.
self.buffer = []
## Pointer for buffer list.
self.buffer_pointer = 0
## Variable for checking buffer fulfillment.
# if this is equals to buffer size -> buffer is full
self.buffer_fulfillment = 0
for i in range(3):  # populate buffer with 0-s, default buffer size is 3
    self.buffer.append(0)

## Graph points that make up the left side of the graph
self.graph_points1 = []
## Graph points that make up the right side of the graph
self.graph_points2 = []
## Maximum value of the abscissa
self.max_abscissa = 4000

def reset(self):
    '''
    Method that resets the graph.
    '''
    self.begin_time = int(round(time.time() * 1000)) - 4  # 4 is subtracted for a small spare (for error prevention)
    self.graph_points1 = []
    self.graph_points2 = []
    self.passed_time = 0

def get_current_time(self):
    '''
    Method that evaluates the current measurement running time.
    '''
    self.current_time = int(round(time.time() * 1000)) - self.begin_time
    if self.current_time > self.end_time:
        self.reset()
        self.current_time = int(round(time.time() * 1000)) - self.begin_time
    while (self.current_time % 4) != 0:  # all measurement times in file (and data dict) are divisible by four
        self.current_time -= 1

def read_file_to_list(self):
    '''
    Method that reads previously measured data to the data dictionary.
    Dictionary key is time in milliseconds and value is a tuple of FouKG and EKG data.
    '''
    data_file = open(data_file_location, "r")
data_file.readline()  # no need for the first line
    for line in data_file.readlines():
        parse = line.split()
        # time in ms, FouKG, EKG
        self.data_dict[int(parse[0])] = (-int(parse[1]), int(parse[2]))
    self.end_time = int(parse[0])  # set measurements end time
    data_file.close()
def get_present_data(self):
    
    Method that gets present data and puts it into the buffer.
    Right now the data is acquired from the data_file, but in the
    future from the cardiograph itself.
    
    self.get_current_time()
    self.present_data = self.data_dict[self.current_time][self.graph_type]
    # put the present data into the buffer
    self.buffer[self.buffer_pointer] = self.present_data
    self.buffer_pointer += 1
    if self.buffer_pointer >= len(self.buffer):
        self.buffer_pointer = 0

def make_graph(self):
    
    Method that constructs the showable graph (list of points).
    
    self.get_present_data() # first get the present data
    
    # if buffer is full -> show data, otherwise wait for buffer to fill
    if self.buffer_fulfillment >= (len(self.buffer) - 1):
        buffer_value = self.buffer_value()
        self.measured_data.append((self.current_time, buffer_value))
        # save data as a tuple
        self.check_graph_reset()  # remove elements from graph2 to make room for graph1
        for elem in self.graph_points2:
            if elem[0] <= (self.current_time - self.passed_time):
                self.graph_points2.remove(elem)

        self.graph_points1.append((self.current_time -
            self.passed_time, buffer_value)) # add new value to graph
        self.graph_points1.sort(key=itemgetter(0)) # sort is necessary for correct graph line

    return self.graph_points1, self.graph_points2[10:]
    else:
        self.buffer_fulfillment += 1
        return [], []

def check_graph_reset(self):
    
    Method that checks if the graph line has reached the right side.
    
    if self.current_time > (self.passed_time + self.max_abscissa):
        self.passed_time += self.max_abscissa
        self.graph_points2 = self.graph_points1
        self.graph_points1 = []

def buffer_value(self):
    
    Method that calculates the average value in the buffer.
    
    total_value = 0
    for value in self.buffer:
        total_value += value
    return int(total_value / len(self.buffer))
```python
class Logic(BoxLayout):
    '''
    Class that handles all the program logic.
The core class.
    '''

    ## The constructor.
    def __init__(self,):
        super(Logic, self).__init__()
        # plot object used for showing left side of the graph.
        self.plot1 = Graph.MeshLinePlot(color=[0, 1, 0, 1])
        # plot object used for showing right side of the graph.
        self.plot2 = Graph.MeshLinePlot(color=[0, 1, 0, 1])
        # Trigger, that shows if there have been any measurements.
        self.started = False
        # Cardiograph object.
        self.cardiograph = Cardiograph()
        # Database object.
        self.database = Profile_Database()
        # List for all the profile radiobuttons.
        self.checkbox_list = []
        # List for all the profile labels.
        self.label_list = []
        # autocalibration frequency in seconds
        self.autocalibration_frequency = 1
        Clock.schedule_interval(self.update_time, 1)  # show time (update every second)
        Clock.schedule_once(self.load_profiles, 0.2)

    def show_data(self, dt):
        '''
        Method that shows measured data on the graph.
        '''
        self.plot1.points, self.plot2.points = self.cardiograph.make_graph()

    def start_stop(self):
        '''
        Method that starts or stops the measurements depending on the "START/STOP" button state.
        '''
        if self.started == False:  # change trigger to active, there have been measurements
            self.started = True
            if self.ids.button1.text == "START":  # "Start" has been pressed
                self.cardiograph.measured_data = []
                active_profile, _ = self.active_profile()
                self.database.new_measurement(str(time.asctime()), active_profile.id)
                Clock.unschedule(self.show_data)  # unschedule needed for correct Clock work
                self.ids.graph.add_plot(self.plot1)  # add_plot
                self.ids.graph.add_plot(self.plot2)
                self.ids.button1.text = "STOP"
                self.auto_calibrate()  # check if autocalibration is on
                Clock.schedule_once(self.calibrate, 0.2)
                self.cardiograph.reset()  # set the beginning time of the measuring
                Clock.schedule_interval(self.show_data, 0.005)
```

35
else:  # "Stop" has been pressed
    self.stop()

def stop(self):
    ""
    Method that stops the measuring.
    ""
    if self.ids.button1.text == "STOP":
        self.ids.button1.text = "START"
        Clock.unschedule(self.show_data)
        Clock.unschedule(self.calibrate)
        self.save_data()

def save_data(self):
    ""
    Method that saves all current measurement data to the database.
    ""
    current_measurement = self.database.current_measurement()
    for value in self.cardiograph.measured_data:
        self.database.add_value(value[0], value[1],
                                current_measurement)
    self.database.commit()

def calibrate(self, dt):
    ""
    Method that puts the graph in the middle of the screen.
    ""
    points = self.cardiograph.graph_points1 +
             self.cardiograph.graph_points2
    if points != []:
        max_value = max(points, key=itemgetter(1))[1]  # max value of
        max_value = int(ceil(max_value / 100.0)) * 100  # rounds up to
        hundreds (example: 2444 -> 2500)
        min_value = min(points, key=itemgetter(1))[1]  # min value of
        min_value = int(floor(min_value / 100.0)) * 100  # rounds down
        to hundreds
        self.ids.graph.ymax = max_value + 800  # 800 is added for
        bigger space
        self.ids.graph.ymin = min_value - 800

def auto_calibrate(self):
    ""
    Method that turns the auto calibration on and off.
    ""
    if self.ids.togglebutton1.state == 'down':
        Clock.unschedule(self.calibrate)
        Clock.schedule_interval(self.calibrate,
                                 self.autocalibration_frequency)
        self.ids.togglebutton1.text = "AUTO:
                                       \n                                       ON"
    else:
        Clock.unschedule(self.calibrate)
        self.ids.togglebutton1.text = "AUTO:
                                       \n                                       OFF"

def increase_abscissa(self):
    ""
    Method that increases the graph's abscissa.
    ""
    if self.started:
        current_abscissa = self.ids.graph.xmax
if current_abscissa < 5000:
    self.cardiograph.max_abscissa = current_abscissa + 1000
    self.cardiograph.check_graph_reset()
    self.ids.graph.xmax = current_abscissa + 1000

def decrease_abscissa(self):
    '''
    Method that decreases the graph's abscissa.
    '''
    if self.started:
        current_abscissa = self.ids.graph.xmax
        if current_abscissa > 2000:
            self.cardiograph.max_abscissa = current_abscissa - 1000
            self.cardiograph.check_graph_reset()
            self.ids.graph.xmax = current_abscissa - 1000

def increase_ordinate(self):
    '''
    Method that increases the graph's ordinate.
    '''
    if self.started:
        current_ordinate = self.ids.graph.ymax - self.ids.graph.ymin
        if self.ids.togglebutton_EKG.state == 'down':
            if current_ordinate < 4000:
                self.ids.graph.ymax += 500
                self.ids.graph.ymin -= 500
            else:
                if current_ordinate < 8000:
                    self.ids.graph.ymax += 500
                    self.ids.graph.ymin -= 500
        else:
            if current_ordinate > 2000:
                self.ids.graph.ymax -= 500
                self.ids.graph.ymin += 500
            else:
                if current_ordinate > 5000:
                    self.ids.graph.ymax -= 500
                    self.ids.graph.ymin += 500

def decrease_ordinate(self):
    '''
    Method that decreases the graph's ordinate.
    '''
    if self.started:
        current_ordinate = self.ids.graph.ymax - self.ids.graph.ymin
        if self.ids.togglebutton_EKG.state == 'down':
            if current_ordinate > 2000:
                self.ids.graph.ymax -= 500
                self.ids.graph.ymin += 500
            else:
                if current_ordinate > 5000:
                    self.ids.graph.ymax -= 500
                    self.ids.graph.ymin += 500

def update_time(self, dt):
    '''
    Method for updating time and date.
    '''
    self.ids.label_time.text = str(time.asctime())

def update_autocalibration_frequency(self):
    '''
    Method that updates the autocalibration frequency.
    '''
    self.autocalibration_frequency = self.ids.slider2.value
    self.auto_calibrate()

def update_buffer(self):
Method for updating buffer size.

```python
self.cardiograph.buffer = []
self.cardiograph.buffer_pointer = 0
self.cardiograph.buffer_fulfillment = 0
# populate buffer with 0-s depending on the buffer size
for i in range(int(self.ids.slider1.value)):
    self.cardiograph.buffer.append(0)
```

```python
def checkbox_check(self, checkbox):
    # Method for checking that one radiobutton is always active.
    # If the pressed radiobutton is already active, don't change it to inactive.
    if checkbox.active == False:
        checkbox.active = True
```

```python
def graph_type_check(self, button):
    # Method for checking that one graph type is always active (FouKG or EKG) and changing the shown graph type.
    if button.state == 'normal':
        button.state = 'down'
    else:
        if button.text == 'EKG':
            self.cardiograph.graph_type = 1
        else:
            self.cardiograph.graph_type = 0
```

```python
def quit_application(self):
    # Method that stops the application.
    App.get_running_app().stop()
```

```python
def load_profiles(self, dt):
    # Method that loads all the profiles from the database and adds them to the GUI.
    profile_ids, profile_names = self.database.load_profiles()
    center_y = 0.8
    for i in range(len(profile_ids)):
        self.make_profile(center_y, str(profile_ids[i]), str(profile_names[i]))
        center_y -= 0.1
    self.checkbox_list[0].active = True
```

```python
def new_profile(self):
    # Method that creates a new profile to the database and GUI.
    If there are already 8 profiles, a popup is presented instead.
    if len(self.label_list) >= 8:
        try:
            self.ids.popup_new_profile.open()
        except:
            print("Delete Profile popup error."))```
else:
    center_y = self.checkbox_list[-1].pos_hint['center_y'] - 0.1
    index = len(self.label_list) + 1
    name = 'Profile ' + str(index)
    profile_id = self.database.new_profile(name)
    self.make_profile(center_y, str(profile_id), str(name))

def make_profile(self, center_y, profile_id, profile_name):
    '''
    Method that creates a new profile in the GUI.
    This is done by creating a new label with the profile name and a
    checkbox.
    '''
    l = Label(text=profile_name, id=profile_id, font_size=18,
              pos_hint={'center_x': 0.22, 'center_y': center_y},
    )
    c = CheckBox(id=profile_id, size_hint_y=None, size_hint_x=0.2,
                  pos_hint={'center_x': 0.15, 'center_y': center_y},
                  height='48dp',
                  group='group1'
    )
    c.bind(on_press=self.checkbox_check)
    self.ids.second_screen.add_widget(l)
    self.ids.second_screen.add_widget(c)
    self.checkbox_list.append(c)
    self.label_list.append(l)

def delete_profile(self):
    '''
    Method that deletes the active profile.
    If there is only one profile left, a popup is presented instead.
    '''
    if len(self.label_list) <= 1:
        try:
            self.ids.popup_delete_profile.open()
        except:
            print("Delete Profile popup error."")
    else:
        checkbox, label = self.active_profile()
        self.ids.second_screen.remove_widget(checkbox)
        self.ids.second_screen.remove_widget(label)
        self.checkbox_list.remove(checkbox)
        self.label_list.remove(label)
        self.update_profile_list()
        self.database.delete_profile(label.id)

def edit_profile(self):
    '''
    Method that changes the profile name.
    '''
    checkbox, label = self.active_profile()
    new_name = self.ids.edit_profile_textinput.text
    self.database.edit_profile(new_name, label.id)

def active_profile(self):
    '''
Method that returns the corresponding checkbox and label for the active profile.
```python
for checkbox in self.checkbox_list:
    if checkbox.active:
        for label in self.label_list:
            if label.id == checkbox.id:
                return checkbox, label
```

def update_profile_list(self):
    '''
    Method that updates the profile list in the GUI.
    Used when deleting a profile.
    '''
    center_y = 0.8
    for i in range(len(self.checkbox_list)):
        self.checkbox_list[i].pos_hint = {
            "center_x": 0.15, 'center_y': center_y
        }
        self.label_list[i].pos_hint = {
            "center_x": 0.22, 'center_y': center_y
        }
        center_y -= 0.1
    self.checkbox_list[0].active = True

def open_edit_profile_popup(self): # TODO: selle meetodi error korda
    '''
    Method that opens a popup for editing a profile.
    '''
    try:
        checkbox, label = self.active_profile()
        self.ids.popup_edit_profile.open()
        self.ids.edit_profile_textinput.text = label.text
    except:
        print('Edit Profile Popup error. ')

class Profile_Database:
    '''
    Class used for saving and retrieving data from the database.
    '''
    ## The constructor
    def __init__(self):
        ## Connection with the database
        self.connection = sqlite3.connect('database.db')
        print("Database opened successfully")
        ## Cursor object used for executing commands
        self.cursor = self.connection.cursor()
        self.check_database_existence()

    def check_database_existence(self):
        '''
        Method that checks for and existing database (checking every table separately).
        If a table is found to be missing, a new one is created.
        '''
        self.cursor.execute("""SELECT name FROM sqlite_master WHERE type='table' AND name='Profiles' "")
        data = self.cursor.fetchone()
        if data is None:
            self.cursor.execute("CREATE TABLE Profiles(profile_id integer..."
primary key, Name TEXT)"
    self.cursor.execute("INSERT INTO Profiles (name)
VALUES('Profile 1')")

    self.cursor.execute("SELECT name FROM sqlite_master WHERE
    type='table' AND name='Measurements' ")
    data = self.cursor.fetchone()
    if data is None:
        self.cursor.execute("CREATE TABLE Measurements(measurement_id
    integer primary key, timestamp TEXT, profile_id INT)"
    self.commit()

    self.cursor.execute("SELECT name FROM sqlite_master WHERE
    type='table' AND name='Data' ")
    data = self.cursor.fetchone()
    if data is None:
        self.cursor.execute("CREATE TABLE Data(measure_time INT, data
    INT, measurement_id INT)"
    self.commit()

def load_profiles(self):
    ""
    Method that returns all the profile IDs and names from the
database.
    ""
    profile_ids = []
    profile_names = []
    for row in self.cursor.execute('SELECT profile_id, name FROM
Profiles'):
        profile_ids.append(row[0])
        profile_names.append(row[1])
    return profile_ids, profile_names

def new_profile(self, name):
    ""
    Method that creates a new profile with the name "name" in the
Profiles table.
The profiles new generated ID is returned.
    ""
    t = (name,)
    self.cursor.execute("INSERT INTO Profiles (name) VALUES (?)", t)
    self.cursor.execute("SELECT profile_id FROM Profiles ORDER BY
    profile_id DESC LIMIT 1")
    id = self.cursor.fetchone()[0]
    self.commit()
    return id

def delete_profile(self, id):
    ""
    Method that deletes a profile with the ID "id" from the Profiles
table.
    ""
    # a tuple used for making safe data requests
    t = (id,)
    self.cursor.execute("DELETE FROM Profiles WHERE profile_id = ?", t)
    self.commit()

def edit_profile(self, new_name, id):
    ""
    Method that changes a profile name with ID "id" with a name
"new_name" in the Profiles table.
t = (new_name, id,)
self.cursor.execute("UPDATE Profiles set name = ? where profile_id = ?", t)
self.commit()

def new_measurement(self, timestamp, profile_id):
    
    Method that creates a new measurement to the Measurements table.

    t = (timestamp, profile_id,)
    self.cursor.execute("INSERT INTO Measurements (timestamp, profile_id) VALUES (?,?)", t)
    self.commit()

    
def current_measurement(self):
        
        Method that returns the ID of the current measurement.

        self.cursor.execute("SELECT measurement_id FROM Measurements ORDER BY measurement_id DESC LIMIT 1;")
        id = self.cursor.fetchone()[0]
        self.commit()
        return int(id)

    
def add_value(self, measure_time, data, measurement_id):
        
        Method that adds new data to the table Data.

        t = (measure_time, data, measurement_id,)
        self.cursor.execute("INSERT INTO Data VALUES (?,?,?)", t)

    
def commit(self):
        
        Method that saves (commits) the changes to the database.

        self.connection.commit()

    
def close(self):
        
        Method that closes the database.

        self.connection.close()

    

class Cardiograph_App(App):
    
    
    Class Cardiograph_App is a subclass of the App class. This is necessary for running a Kivy application.

    
    def build(self):
        
        Method build returns a Widget instance (the root of a widget tree) by loading the Kv language file.

        return Builder.load_file("cardiograph_app_gui.kv")

    

## first line executed in the program

if __name__ == "__main__":
Cardiograph_App().run()
Lisa 2. Kardiograafi kasutajaliidese Kivy fail

#import MeshLinePlot kivy.garden.graph.MeshLinePlot

Logic:

ScreenManager:
    id: sm

Screen:
    name: 'screen1'

BoxLayout:
    id: bl
    orientation: "vertical"

FloatLayout:
    size_hint: [1, .8]

Graph:
    pos_hint: {"center_x": 0.5, 'center_y': 0.5}
    id: graph
    xlabel: "Time (ms)"
    x_ticks_major: 1000
    x_ticks_minor: 5
    x_grid: True
    x_grid_label: True
    xmin: 0
    xmax: 4000
    ylabel: "ADC Unit"
    y_ticks_major: 500
    y_ticks_minor: 5
    y_grid: True
    y_grid_label: True
    ymin: 0
    ymax: 7000

Button:
    id: inc_abscissa_button
    size_hint: [0.07, 0.1]
    pos_hint: {"center_x": 0.93, 'center_y': 0.2}
    text: "+
    font_size: 20
    bold: True
    background_color: [1, 1, 1, 0.5]
    on_press:
        root.increase_abscissa()

Button:
    id: dec_abscissa_button
    size_hint: [0.07, 0.1]
    pos_hint: {"center_x": 0.2, 'center_y': 0.2}
    text: "-
    font_size: 24
    bold: True
    background_color: [1, 1, 1, 0.5]
    on_press:
        root.decrease_abscissa()

Button:
    id: inc_ordinate_button
    size_hint: [0.07, 0.1]
Screen:
  name: 'screen2'
  canvas.before:
    Color:
rgb: .2, .2, .2
Rectangle:
    size: self.size

FloatLayout:
    id: second_screen
Label:
    font_size: 20
    pos_hint: {'center_x': 0.15, 'center_y': 0.9}
    text: 'Selected profile:

# Profile related buttons
Button:
    id: button_np
    size_hint: None, None
    size: '150dp', '48dp'
    pos_hint: {'center_x': 0.4, 'center_y': 0.85}
    text: 'New Profile'
    on_release:
        root.new_profile()

Button:
    id: button_dp
    size_hint: None, None
    size: '150dp', '48dp'
    pos_hint: {'center_x': 0.4, 'center_y': 0.75}
    text: 'Delete profile'
    on_release:
        root.delete_profile()

Button:
    id: button_ep
    size_hint: None, None
    size: '150dp', '48dp'
    pos_hint: {'center_x': 0.4, 'center_y': 0.65}
    text: 'Edit profile'
    on_press:
        root.open_edit_profile_popup()

# Popups
Popup:
    id: popup_delete_profile
    title: "Can't delete profile"
    on_parent:
        if self.parent == second_screen:
            self.parent.remove_widget(self)
    size_hint: 0.7, 0.7
    FloatLayout:
        Label:
            text: 'There has to be at least one profile'
            font_size: 24
            pos_hint: {'center_x': 0.5, 'center_y': 0.7}
        Button:
            text: 'OK'
            size_hint: 0.3, 0.2
            pos_hint: {'center_x': 0.5, 'center_y': 0.3}
            on_release: popup_delete_profile.dismiss()

Popup:
    id: popup_new_profile
title: "Can't make new profile"

on_parent:
    if self.parent == second_screen:
        self.parent.remove_widget(self)
    size_hint: 0.7, 0.7

FloatLayout:
    Label:
        text: 'Maximum number of profiles reached'
        font_size: 24
        pos_hint: {'center_x': 0.5, 'center_y': 0.7}
    Button:
        text: 'OK'
        size_hint: 0.3, 0.2
        pos_hint: {'center_x': 0.5, 'center_y': 0.3}
        on_release: popup_new_profile.dismiss()

Popup:
    id: popup_new_profile
    title: "Edit profile"
    on_parent:
        if self.parent == second_screen:
            self.parent.remove_widget(self)
        size_hint: 0.7, 0.7
    FloatLayout:
        Label:
            text: 'Profile name:'
            font_size: 24
            pos_hint: {'center_x': 0.2, 'center_y': 0.8}
        TextInput
            id: edit_profile_textinput
            size_hint_y: None
            size_hint_x: 0.65
            pos_hint: {'center_x': 0.5, 'center_y': 0.6}
            height: '40dp'
            font_size: 22
            multiline: False
            text: 'New Profile'
        Button:
            text: 'OK'
            size_hint: 0.3, 0.2
            pos_hint: {'center_x': 0.5, 'center_y': 0.3}
            on_press: root.edit_profile()
            on_release: popup_edit_profile.dismiss()

# Time label
Label:
    id: lab
    el_time
    font_size: 16
    pos_hint: {'center_x': 0.8, 'center_y': 0.9}

Button:
    id: back_button
    size_hint: None, None
    pos_hint: {'center_x': 0.8, 'center_y': 0.14}
    size: '150dp', '48dp'
    text: 'BACK'
    bold: True
    on_release:
        root.update_buffer()
root.update_autocalibration_frequency()
sm.transition.direction = 'right'
sm.current = 'screen1'

# Slider for Buffer Size
Label:
    font_size: 18
    pos_hint: {'center_x': 0.78, 'center_y': 0.75}
    text: 'Buffer Size'

Slider:
    id: slider1
    size_hint_x: 0.3
    size_hint_y: 0.2
    pos_hint: {'center_x': 0.8, 'center_y': 0.68}
    range: (1, 9)
    step: 2
    value: 3.0

Label:
    font_size: 18
    pos_hint: {'center_x': 0.86, 'center_y': 0.75}
    text: '{}'.format(slider1.value)

# Slider for autocalibration frequency
Label:
    font_size: 18
    pos_hint: {'center_x': 0.78, 'center_y': 0.55}
    text: 'Calibration frequency'

Slider:
    id: slider2
    size_hint_x: 0.3
    size_hint_y: 0.2
    pos_hint: {'center_x': 0.8, 'center_y': 0.48}
    range: (1, 5)
    step: 1
    # value: 3.0

Label:
    font_size: 18
    pos_hint: {'center_x': 0.91, 'center_y': 0.55}
    text: '{}'.format(slider2.value)

# FouKG or EKG togglebuttons
ToggleButton:
    id: togglebutton_EKG
    size_hint: [0.15, 0.1]
    pos_hint: {'center_x': 0.87, 'center_y': 0.3}
    text: "EKG"
    bold: True
    group: 'togglegroup1'
    state: 'down'
    on_press: root.graph_type_check(togglebutton_EKG)

ToggleButton:
    id: togglebutton_FouKG
    size_hint: [0.15, 0.1]
    pos_hint: {'center_x': 0.72, 'center_y': 0.3}
    text: "FouKG"
    bold: True
group: 'togglegroup1'
on_press: root.graph_type_check(togglebutton_FouKG)

# Quit button
Button:
id: quit_button
size_hint: [0.15, 0.1]
pos_hint: {"center_x": 0.4, 'center_y': 0.2}
text: "QUIT"
bold: True
background_color: [.78, .15, .21, 1] # ruby red
on_release: root.quit_application()
Public Member Functions

def __init__(self)
    The constructor. More...

def reset(self)

def get_current_time(self)

def read_file_to_list(self)

def get_present_data(self)

def make_graph(self)

def check_graph_reset(self)

def buffer_value(self)

Public Attributes

present_data
    Currently measured data. More...

measured_data
    All measured data in one measurement. More...

data_dict
    Previously measured data, that will be showed on the graph (EKG or FouKG) end_time
Previously measured data end time in milliseconds. More...

**begin_time**
Begin time of the measuring. More...

**current_time**
Current time of current measurement. More...

**passed_time**
Passed time in chunks. More...

**graph_type**
Graph type selection. More...

**buffer**
Buffer for smoothening the graph. More...

**buffer_pointer**
Pointer for buffer list. More...

**buffer_fulfillment**
Variable for checking buffer fulfillment. More...

**graph_points1**
Graph points that make up the left side of the graph.

**graph_points2**
Graph points that make up the right side of the graph.

**max_abscissa**
Maximum value of the abscissa.
Detailed Description
Class that handles all the graph related things.

Constructor & Destructor Documentation

§.\_\_init\_\_(\)

def cardiograph.Cardiograph.\_\_init\_\_ ( self )

The constructor.

Member Function Documentation

§.buffer\_value()

def cardiograph.Cardiograph.buffer\_value ( self )

Method that calculates the average value in the buffer.

§.check\_graph\_reset()

def cardiograph.Cardiograph.check\_graph\_reset ( self )

Method that checks if the graph line has reached the right side.

§.get\_current\_time()

def cardiograph.Cardiograph.get\_current\_time ( self ) Method that evaluates
the current measurement running time.

§.get\_present\_data()

def cardiograph.Cardiograph.get\_present\_data ( self )

Method that gets present data and puts it into the buffer.
Right now the data is acquired from the data_file, but in the future from the cardiograph itself.


§_make_graph()

def cardiograph.Cardiograph.make_graph ( self )

Method that constructs the showable graph (list of points).

§_read_file_to_list()

def cardiograph.Cardiograph.read_file_to_list ( self )

Method that reads previously measured data to the data dictionary.
Dictionary key is time in milliseconds and value is a tuple of FouKG and EKG data.

§_reset()

def cardiograph.Cardiograph.reset ( self )

Method that resets the graph.

Member Data Documentation

§_begin_time

cardiograph.Cardiograph.begin_time Begin time of the measuring.

§_buffer

cardiograph.Cardiograph.buffer

Buffer for smoothening the graph.

§_buffer_fulfillment

cardiograph.Cardiograph.buffer_fulfillment

Variable for checking buffer fulfillment. if this is equals to buffer size -> buffer is full
§_buffer_pointer

cardiograph.Cardiograph.buffer_pointer Pointer for buffer list.

§_current_time

cardiograph.Cardiograph.current_time Current time of current measurement.

§_end_time

cardiograph.Cardiograph.end_time

Previously measured data end time in milliseconds.

§_graph_type

cardiograph.Cardiograph.graph_type

Graph type selection.

0 - FouKG; 1 - EKG

§_measured_data

cardiograph.Cardiograph.measured_data

All measured data in one measurement.

§_passed_time

cardiograph.Cardiograph.passed_time Passed time in chunks.

§_present_data
cardiograph.Cardiograph.present_data  Currently  measured
data.

The documentation for this class was generated from the following file:

- cardiograph.py

*Generated by* 1.8.12
Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, Henri Tamm

1. Annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose
   „Foucault’ kardiograafi haldurarvuti prototüüp“
   mille juhendaja on Vahur Zadin
   (a) reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, seal-
   hulgast digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja
   lõppemiseni;
   (b) üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgast
   digitaalarhiivi DSpace’i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile;
3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega
   isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, 17.05.2017