Tartu Ülikool

Loodus- ja täppisteaduste valdkond

Tehnoloogiainstituut

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 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, süsteemiteooria;

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

Lühendid ja mõisted

ADC (ingl *analoog-to-digital converter*) – analoog-digitaalmuundur.

DSI (ingl *Display Serial Interface*) – kuvari jadaliides.

EhhoKG (ingl *echocardiography*) – ehhokardiograafia.

EKG (ingl *electrocardiography*) – elektrokardiograafia.

FouKG (ingl *Foucault cardiography*) – Foucault' kardiograafia.

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

GPU (ingl *graphics processing unit*) – graafikaprotsessor.

GUI (ingl *graphical user interface*) – graafiline kasutajaliides.

HDMI (ingl *High-Definition Multimedia Interface*) - kõrglahutusega multimeediumiliides.

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

IDE (ingl *integrated development environment*) – integreeritud programmeerimiskeskkond.

KT (ingl *computed tomography*) – kompuutertomograafia.

LED (ingl *light emitting diode*) – valgusdiood.

MRT (ingl *magnetic resonance imaging*) – magnetresonantstomograafia.

OS (ingl *operating system*) – operatsioonisüsteem.

SDRAM (ingl *synchronous dynamic random-access memory*) – sünkroonne dünaamiline muutmälu.

USB (ingl *Universal Serial Bus*) – universaalne järjestiksiin.

1 Sissejuhatus

Tänapäeval on inimeste üheks peamiseks surma põhjuseks südame tööga seotud haigused nagu südame veresoonte lupjumine ja infarkt. [1] Südame tervist mõjutavad mitmed faktorid nagu toitumine, suitsetamine, alkoholi tarbimine, kui ka vaimne tervis. Ootamatute haiguste vältimiseks soovitavad arstid käia regulaarselt kontrollides, kus mõõdetakse patsiendi vererõhku, kolesterooli taset, veresuhkrut ning südame elektrilist aktiivsust EKG näol. Teades inimeste loomusest tulenevat laiskust oleks mugavam, kui patsiendid saaksid südame tööd kontrollida otse kodust. Elektrokardiogrammi mõõtmiseks kasutatav tehnika on aga tavakasutaja jaoks kallis ning võtaks kodus palju ruumi.

Südame mehaaniliste protsesside jälgimiseks on võimalik kasutada ka Foucault' kardiograafiat, mis põhineb mitteinvasiivsel 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

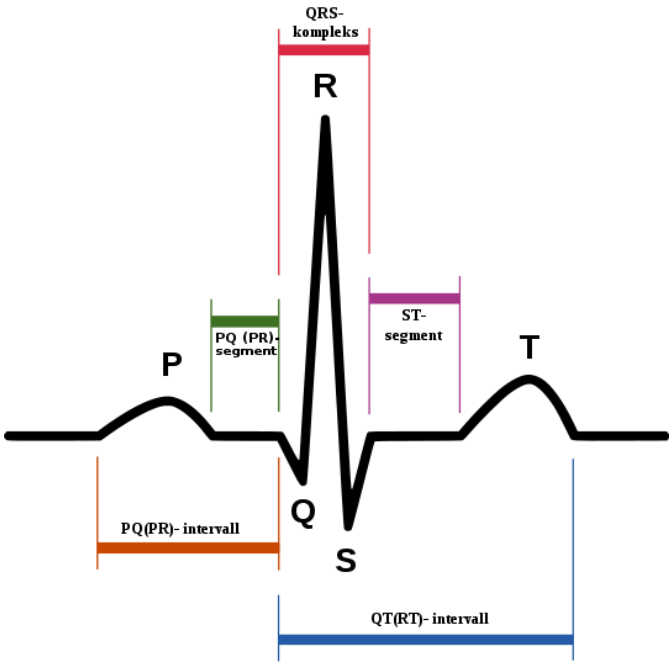
Südame töö jälgimiseks on mitmeid mitteinvasiivseid (veretuid) meetodeid, kuid paljud neist on kohmakad või vajavad pidevat arsti järelevalvet. Näiteks pakub Ida-Tallinna Keskhaigla Funktsionaaldiagnostika osakond südametöö jälgimiseks järgmisi meetodeid: EhhoKG, MRT, KT ning EKG. EhhoKG ehk Ehhokardiograafia on südame ultraheli uuring, mis võimaldab määrata südame klappide seisundit, südame suurust ning verevoolu kiirust. MRT ehk südame magnetresonantstomograafia on meetod, mis kasutab südamest detailse kujutise saamiseks tuumamagnetresonantsi. MRT võimaldab hinnata südame erinevate osade ning veresoonte anatoomiat. KT ehk südame kompuutertomograafia puhul tehakse südamest erinevate nurkade all röntgenpildid. Kompuutertomograafia võimaldab määrata muutusi südame veresoontes. Kõik eelnevalt nimetatud meetodid nõuavad suurt ja kallist tehnikat ning arsti sekkumist mõõtmistel. [2]

EKG ehk elektrokardiograafia on enimkasutatav 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 teha 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)

Elektrokardiograafia on südame tegevuse jälgimise meetod, mis põhineb elektriliste potentsiaalide muutuse registreerimisel keha pinnalt. Inimese jäsemetele ning rindkerele kinnitatakse elektroodid, mis mõõdavad lühikese aja jooksul südamelöögi tagajärjel tekkinud erutuslainete teket, kulgu ning vaibumist. Selle tulemusena tekib iseloomuliku kujuga joon, mille segmentide analüüsimisel on võimalik hinnata südame tervist ning sellega seonduvaid haiguseid. [3] Joonisel 1 on kujutatud südamelöögi tagajärjel tekkiv iseloomulik kardiogrammi kuju.

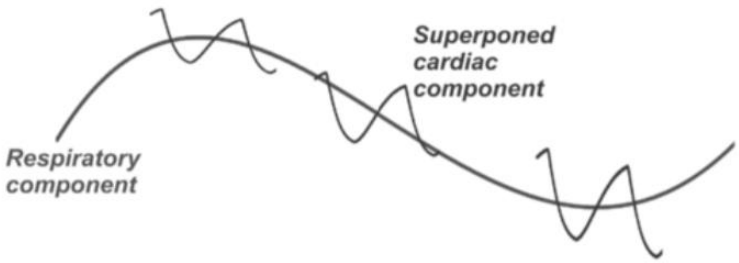


Joonis 1 Iseloomulik kardiogrammi kuju. [4]

2.2 Foucault' kardiograafia (FouKG)

Foucault' kardiograafia on südame tegevuse jälgimise meetod, mis põhineb südamepiirkonna sondeerimisel raadiosageduslike pööris-elektrivooludega ehk Foucault' vooludega. Erinevalt EKG-st ei kinnitata patsiendi kehale elektroode, vaid rindkere pinnale südame piirkonda asetatakse induktiivpool, mis indutseerib magnetvälja abil patsiendi kehas nõrgad pöörisvoolud. Mõõtes pöörisvoolude energiakadu kehas, saadakse signaal, mis on sarnane südame löögimahule ajas. [5] Joonisel 2 on kujutatud Foucault' kardiograafile iseloomulik joon, kus eraldi on välja toodud graafiku südamekomponent (ingl *Superponed cardiac component*) ning hingamiskomponent (ingl *Respiratory component*).

Foucault' kardiograafia nimi tuleb prantsuse füüsiku ning esimese pöörisvoolude uurija Lèon Foucault nimest.



Joonis 2 Iseloomulik Foucault' kardiogrammi kuju. [5]

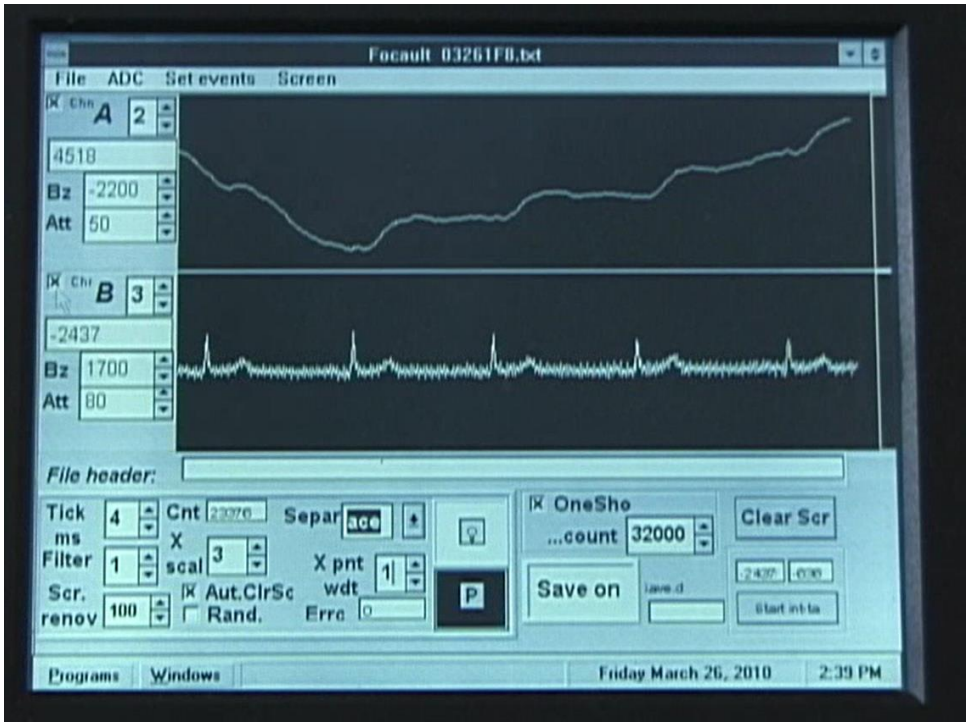
2.3 Tartu Ülikoolis hetkel kasutatav Foucault' kardiograafi haldurarvuti

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]

	Protsessor	**Mälu**	**Püsimälu**	**OS**	**Ekraan**
IBM ThinkPad 500	IBM 486 SLC2 50 MHz CPU	4MB RAM	80 – 170 MB flopiketas	Windows 3.1	7,24 tolli, eraldusvõime 640x480

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. Võrreldes vana haldurarvuti versiooniga, rõhutakse uue haldurarvuti

tegemisel kasutajamugavusele ning kuna tegemist on potentsiaalse tootega, peaks halduarvuti modernne välja nägema.

Püstitatud eesmärgi saavutamiseks oli tarvis lahendada järgmised ülesanded:

1. leida sobiv riistvara halduarvuti 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 Haldurarvuti prototüüp

3.1 Riistvara

Foucault' kardiograafi haldurarvuti riistvara valimisel lähtuti asjaolust, et haldurarvuti oleks mobiilne ning mugav kasutada. Järgnevalt on kirjeldatud kõik haldurarvuti riistvara osad.

3.1.1 Arendusplaat

Elektrooniliste projektide tegemiseks on internetis mitmeid arendusplaate. Neist kaheks kõige populaarsemaks võib pidada Raspberry Pi'd ja Arduinot. Kahe arendusplaadi suurimaks erinevuseks on see, et Raspberry Pi'd võib pidada üldotstarbeliseks arvutiks, Arduino on aga mikrokontroller. Arendusplaatide erinevused võimsuses on suured (Tabel 2.). Kui Arduino sobib hästi täitma ühte kindlat ülesannet, siis Raspberry Pi on võimeline kuvama kasutajale reaalajas graafikut, töötlema läbi puutetundliku ekraani saadud sisendit ning salvestama mõõdetud andmeid andmebaasi.

Tabel 2. Raspberry Pi 3 Model B ja Arduino Mega riistvara võrdlus.

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

Eelnevate kaalutluste põhjal valiti käesoleva seadme arendusplaadiks Raspberry Pi 3 Model B, mis on väikse seadme jaoks hea suurusega ning piisavalt võimas. Kasutajalt sisendi saamiseks ning andmete kuvamiseks valiti Raspberry Pi puutetundlik ekraan, mille eraldusvõime on 800x480 pikslit. Vastavalt sellele on disainitud seadme tarkvara.

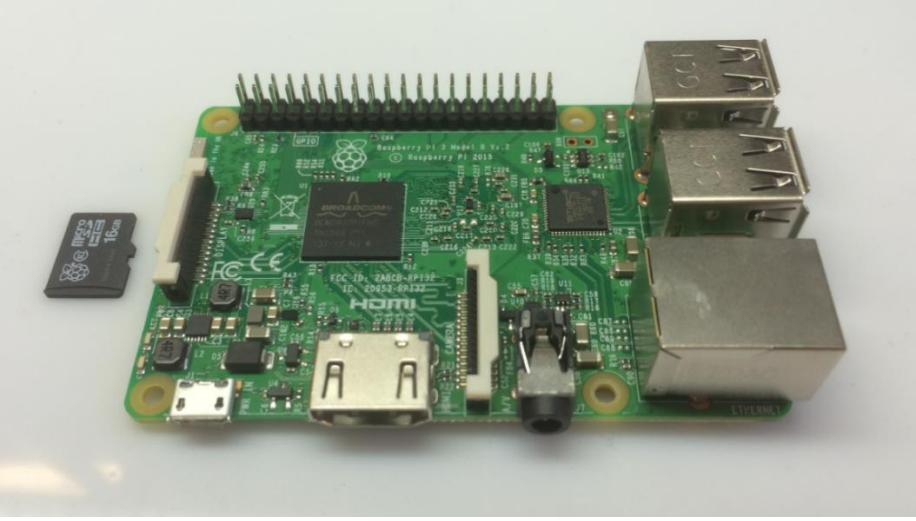
3.1.2 Raspberry Pi

Raspberry Pi on väike krediitkaardi-suurune arvuti, mille arendaja on Raspberry Pi Foundation. [7] Käesoleva seadme jaoks valiti Raspberry Pi 3 Model B (Joonis 4), mis kasutab Broadcom'i süsteemikiipi BCM2837, milles on nelja-tuumaline 1,2 GHz 64-bit ARM Cortex-A53

protsessor. Raspberry Pi'l on 1 GB SDRAM-i (ingl *synchronous dynamic random-access memory*), mis on jagatud GPU-ga (ingl *graphics processing unit*). [8]

Raspberry Pi kasutamiseks saab selle külge ühendada USB (ingl *Universal Serial Bus*) liidesega klaviatuuri, hiire ning HDMI (ingl *High-Definition Multimedia Interface*) kaabliga monitori. Kuna käesolevas projektis kasutati Raspberry Pi puutetundliku ekraani, ei olnud ühtegi eelnevalt nimetatud seadmetest otseselt vaja. Küll aga ühendati arendamisel seadme külge klaviatuur ja hiir.

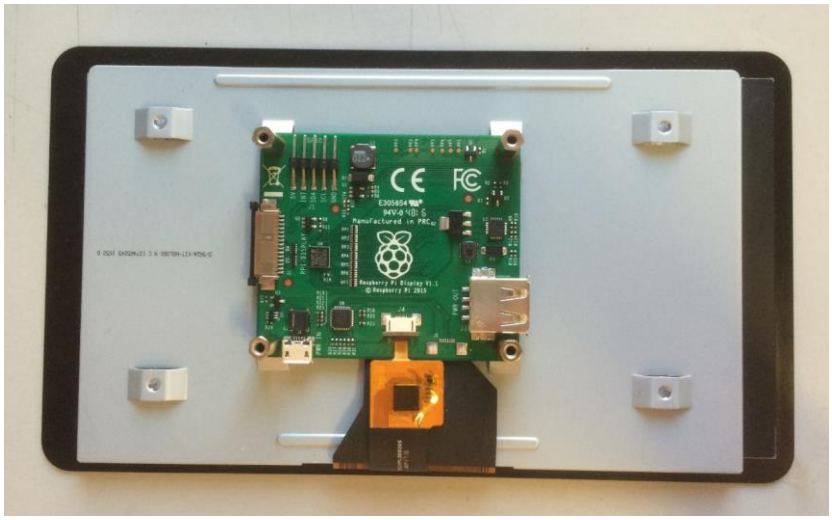
Raspberry Pi'l ei ole ühtegi püsimälu andmekandjat. Selleks on aga Pi'l SD-kaardi pesa, kuhu saab arvuti püsimäluna ühendada MicroSD kaardi. [7] Käesolevas töös paigaldati SD-kaardile Raspbian Jessie operatsioonisüsteemi, mis on Debian Linuxi distributsioonil põhinev UNIXi-laadne OS (ingl *operating system*). [9]



Joonis 4 Raspberry Pi 3 Model B koos 16 GB MicroSD kaardiga.

3.1.3 Raspberry Pi puutetundlik ekraan

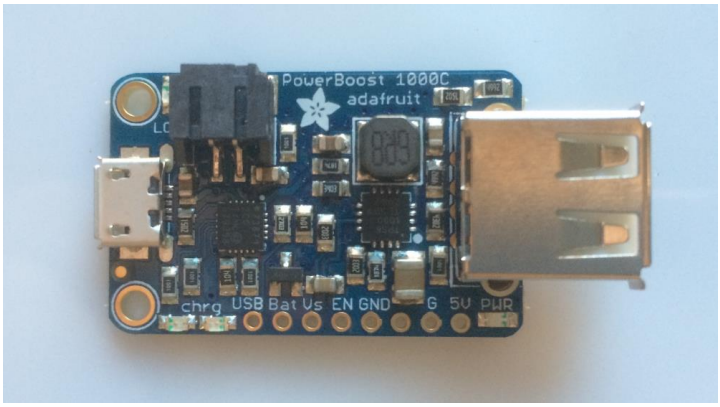
Kasutajaga suhtlemiseks ühendati Raspberry Pi külge Raspberry Pi ametlik 7-tolline puutetundlik ekraan (Joonis 5). Ekraan on eraldusvõimega 800x480 pikslit ning bitisügavusega 24. Raspberry Pi'ga on ekraan ühendatud DSI (ingl *Display Serial Interface)* ribakaabliga ning voolu antakse GPIO (ingl *General-purpose input/output*) viikudest. [10]



Joonis 5 Raspberry Pi puutetundliku ekraani tagumine pool.

3.1.4 Adafruit Powerboost 1000C

Süsteemi voolutarbimist juhib Adafruit Powerboost 1000c toiteskeem (Joonis 6). Powerboost 1000c väljundiks on 5.2V alalisvool, mis sobib hästi Raspberry Pi ja ekraani voolutarbeks. Toiteskeemi külge saab ühendada Micro-USB adapteri, kui ka 3.7V liitiumaku. Powerboost 1000c kasutab TPS61090 boost-muundurit, mis suudab tuvastada aku tühjenemist. Kui aku pinge langeb alla 3,2 voldi, hakkab põlema vastav LED (ingl *light emitting diode*). Peale selle on Powerboostil tark koormust jagav skeem, millega on võimalik samaaegselt tagada seadme töö ning laadida akut. Ka selle jaoks on kiibil eraldi LED-id tähistamaks aku laadimist ning täitumist. [11]



Joonis 6 Adafruit Powerboost 1000C.

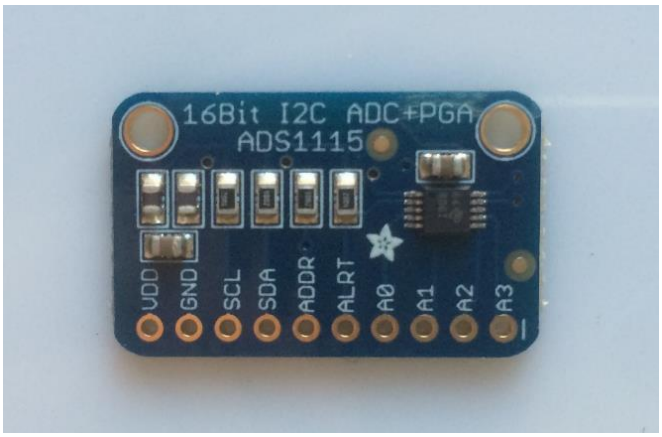
3.1.5 Aku

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

Joonis 7 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 tavaliseks (*single-ended*) sisendiks või kaheks diferentsiaalseks sisendiks. Lisaks on kiibi peal ka võimendi, mis võimaldab signaali võimendada kuni 16 korda. [12]



Joonis 8 Adafruit 16-bit ADC ADS1115.

3.2 Tarkvara

Kuna tegemist on seadme prototüübiga, valiti tarkvara programmeerimiseks Pythoni keel, millest 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

Python on laialt kasutatav üldotstarbeline programmeerimiskeel, mille lõi Guido von Rossum aastal 1991. Pythoni andmetüübid on dünaamilised ehk programmeerijal ei ole tarvis määratleda muutujate tüüpe. Keele disain rõhutab koodi loetavust. Näiteks kasutatakse koodi struktureerimiseks palju tühikuid, mitte kant- või loogelisi sulge. Kõik eelnevalt nimetatu teeb Pythoni kasutajale mugavaks keeleks, millega programmeerida tarkvara prototüüpi. [13]

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

3.2.2 Pycharm

Tarkvara arendamiseks kasutati JetBrains'i Pycharm programmeerimiskeskkonda, sest too pakub Pythoni programmeerimisel mitmekülgset tuge. Näiteks soovitab Pycharm eelnevalt deklareeritud muutujate ja meetodite kasutamisel võimalikke variante ise. Nii säästab programmeerija aega ning koodis esineb vähem vigu. Peale selle võimaldab IDE (ingl *integrated development environment*) kiiret koodi refaktoreerimist ning sisse on integreeritud ka veatuvasti (*debugger)*. [14]

3.2.3 Kivy teek

Pythonil on mitmeid erinevaid raamistikke graafilise kasutajaliidese tegemiseks. Tuntumad neist on näiteks PyQt, PyGUI, libavg ning Tkinter. [15] Raamistiku valimisel lähtuti arendatavast platvormist ning kasutajaliidese eesmärgist. Kardiograafi programmi arendatakse Linuxi operatsioonisüsteemile ning kasutajaliides peab sobima puutetundlikule ekraanile. Eelnevate kaalutluste põhjal valiti graafilise kasutajaliidese tegemiseks Pythoni 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 puutetundlikku 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 nimega *Cardiograph_App* ja defineeriti selle *build()* meetod nii, et see laeb Kivy keele 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.

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

## first line executed in the program
if __name__ == "__main__":
    Cardiograph_App().run()
```

Joonis 9 Lõik koodist kardiograafi programmi käivitumise kohta.

3.2.4 Kivy Garden ja Kivy Graph

Üheks Kivy alamprojektiks on Kivy Garden, mis on kogum erinevate kasutajate poolt tehtud teekidest ja vidinatest (ingl *widget*). Kõik Kivy Garden'is 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 programmi loogikast täielikult eraldatuna. See tagab arusaadava koodi struktuuri ning lubab programmeerijal kasutajaliidest lihtsalt muuta. [20]

Programmi koostamisel luuakse eraldi Kivy keele fail, kus defineeritakse kõik kasutajaliidese osad. Programmi loogikas on antud osadele võimalik määrata erinevaid funktsioone. Näiteks loodi käesoleva projekti Kivy keele failis nupp „New Profile“ ning määrati talle asukoht, suurus, tekst ning ID. Teades objekti ID-d seoti programmi loogikas nupp funktsiooniga, mis loob nupu alla vajutamisel andmebaasi uue profiili.

3.2.6 Kasutajaliides

Kogu kasutajaliides on eraldiseisvalt defineeritud Kivy (Kv) failis „cardiograph_app_gui.kv“. Kivy fail on konstrueeritud 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 programmi loogikat.

Igale kasutajaliidese vidinale on võimalik määrata parameetreid nagu asukoht, suurus ja ID. Vidinate paiknemine on määratud erinevate asetustega (ingl *Layouts*). Kivy võimaldab järgnevaid asetusi: *Anchor-*, *Box-*, *Float-*, *Relative-*, *Grid-*, *Page-*, *Scatter-*, ning *StackLayout*. Kardiograafi programmis kasutati asetusi *BoxLayout* ja *FloatLayout*.

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 *FloatLayout*ile alamvidinate määramisel oluline defineerida vidina täpne asukoht (või suhteline paiknemine akna raamistikus) ning suurus.

3.2.7 SQLite

Mõõdetud andmete salvestamiseks tuli süsteemile valida andmebaas. Kaheks populaarsemaks andmebaasi süsteemiks on hetkel Oracle ning MySQL. [21] Mõlemad andmebaasid on võimsad klient-server tüüpi süsteemid. Kuna käesolevas rakenduses salvestatakse andmeid ainult ühes seadmes ning ühel ajahetkel, ei ole vajalik klient-server tüüpi andmebaas. Peale selle ei pea andmebaas olema võrgus, vaid andmed võivad olla salvestatud kohalikule andmekandjale. Seetõttu sobib käesoleva rakenduse jaoks hästi SQLite andmebaas, mis ei kasuta eraldiseisvat serverit, vaid kogu info kirjutatakse ja loetakse kohalikust failist. SQLite sobib hästi, kui andmebaasi kasutatakse ainult ühes lõimes korraga. Peale selle võtab andmebaas vähe ruumi ning kuna kogu info on salvestatud ühte faili, on andmebaasi lihtne erinevate masinate vahel liigutada. [22]

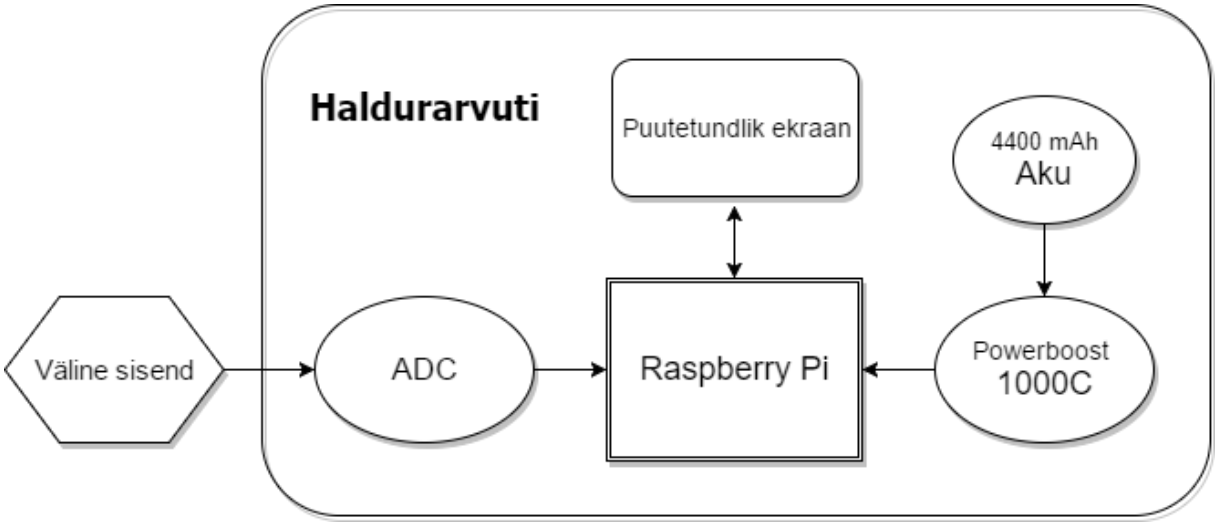
3.2.8 Doxygen

Doxygen on 1997. aastal loodud koodi dokumenteerimise tööriist. Kogu dokumentatsioon kirjutatakse kommentaaridena koodi sisse ja on tänu sellele kergesti jälgitav. Jälgides Doxygeni määratud kommenteerimise reegleid, tuleb kasutajal valida vaid õiged seaded ning tööriist genereerib kogu koodist HTML (ingl *HyperText Markup Language*) formaadis dokumentatsiooni. Doxygen on vabavaraline ning toetab enamus suuri programmeerimskeeli nagu C, C++, Java, Python, PHP jne. [23]

4 Tehtud töö

4.1 Riistvara struktuur

Uus Foucault' kardiograafi haldurarvuti koosneb peatükis 3.1 kirjeldatud riistvarast. Haldurarvuti riistvara struktuur on kujutatud Joonisel 10. Seadme tuumaks on Raspberry Pi 3 Model B, mille peal töötab Raspbian Jessie operatsioonisüsteem. Raspberry Pi külge on ühendatud Raspberry Pi puutetundlik ekraan, millel kuvatakse kasutajale kogu vajalik info. Kuna eesmärgiks on teha kasutajale mobiilne ning mugav seade, on Raspberry Pi ühendatud Adafruit Powerboost 1000c toiteskeemiga, mille küljes on 4400 mAh mahtuvusega taaslaetav liitiumaku, võimaldades haldurarvutit toita nii adapterist, kui ka akust. Välismaailmast sisendi saamiseks on Raspberry Pi külge ühendatud Adafruit'i 4-Channel ADC (ingl *analoog-to-digital converter*). Loodud süsteem töötab 16 GB MicroSD kaardil. Et haldurarvuti jätaks ühtse seadme mulje, on riistvara ümber 3D prinditud kest. Joonisel 11 on näha kõrvuti vana ja uus Foucault' kardiograafi haldurarvuti.



Joonis 10 Foucault' kardiograafi haldurarvuti riistvara.

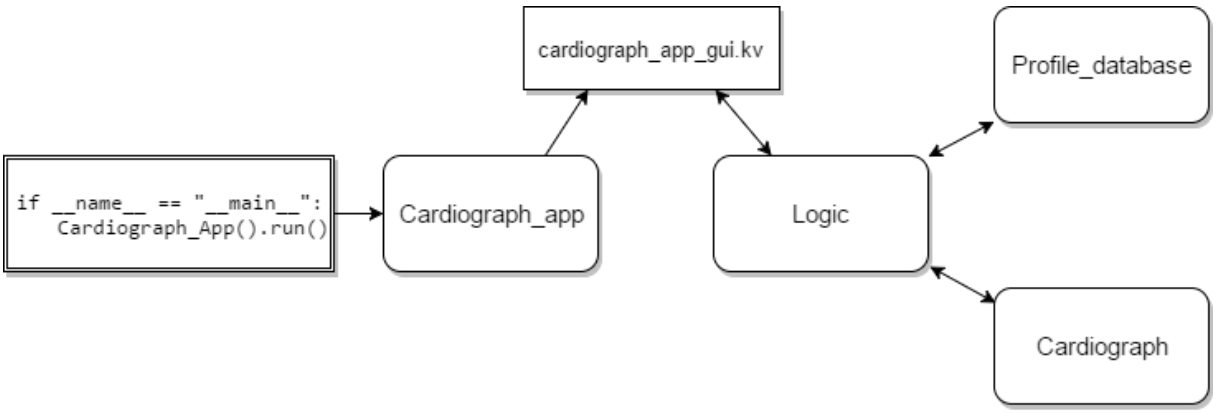
Joonis 11 Vana ja uus Foucault' kardiograafi haldurarvuti.

4.2 Koodi struktuur ja klassid

Kardiograafi programm koosneb neljast klassist: *Cardiograph_App* (vt. peatükk 3.2.3), *Logic*, *Cardiograph* ja *Profile_Database*. Nagu kõigi Kivy teeki kasutavate rakenduste puhul, hakkab ka käesoleva programmi töö *App* klassi alamklassi (ehk klassi *Cardiograph_app*) *run()* meetodi väljakutsumisega. *Cardiograph_App'i build()* meetod loob faili „cardiograph_app_gui.kv" alusel kogu programmi kasutajaliidese (vt. peatükk 3.2.6). Vidinapuust koosneva kasutajaliidese juurvidinaks on klass *Logic*, mis tegeleb kogu programmi loogikaga. Klass *Cardiograph* hoiab endas kõiki graafiku kuvamiseks vajalikke muutujaid ja meetodeid ning klass *Profile_Database* tegeleb andmebaasiga suhtlusega. Klassis *Logic* on loodud mõlema klassi isend, mille kaudu kuvatakse ekraanil graafikut ning salvestatakse mõõdetud andmeid. Eraldi defineeritud klassid loovad loogilise koodistruktuuri (Joonis 12), mida on lihtne jälgida ning muuta.

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 andmebaasi.



Joonis 12 Koodi töö ning klasside struktuur.

4.3 Programmis kasutatavad vidinad

Programmi kasutajaliides on defineeritud vidinapuuna. Käesolevas projektis on Juurvidina *Logic* esimeseks alamvidinaks *ScreenManager*, mis võimaldab kasutajaliideses kuvada erinevaid aknaid ning seda suure valiku üleminekutega. Kardiograafi programmis on kaks akent ehk kaks *Screen*i: peaaken, kus kuvatakse graafik ning kõrvalaken, kus saab hallata profiile ning muuta graafiku sätteid. Üleminekuna kasutatakse *SlideTransition*it, mis vahetab ekraane neid ühele ja teisele poole libistades.

Mõlemas aknas on kasutusel mitmeid nuppe (ingl *Button*), silte (ingl *Label*) ning lüliteid (ingl *ToggleButton*). Peale selle on kõrvalaknas kasutusel mitu hüpikakent (ingl *Popup*), tekstiväli

(ingl *TextInput*), raadionupp (ingl *Checkbox*) ning liugur (ingl *Slider*), millega saab määrata puhvri suurust. Joonisel 13 on kujutatud lõik kasutajaliidest defineerivast vidinapuust.

```
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 esimese alamvidinaga ScreenManager.

4.4 Peaakna disain ja nupud

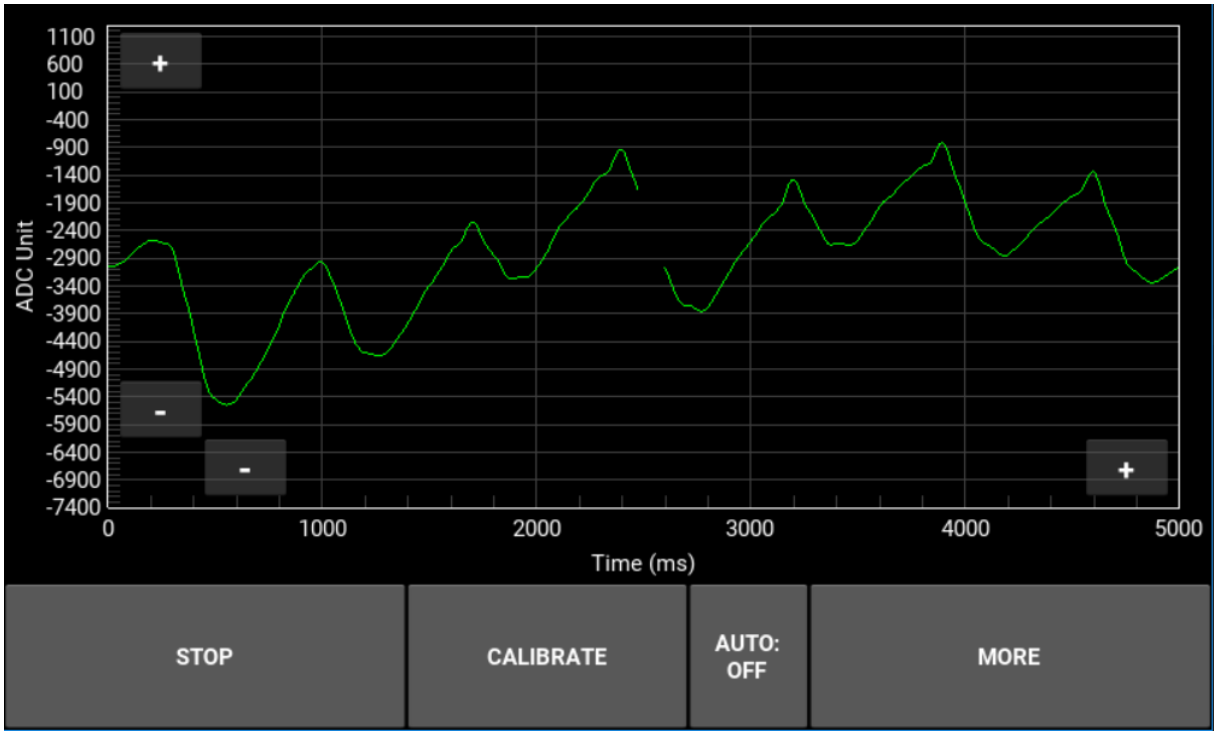
Tarkvara disainimisel võeti eesmärgiks, et kasutajaliides näeks välja lihtne ja konkreetne. Seadet esimest korda kasutav inimene peaks programmi tööst aru saama ka ilma kõrvaliste juhisteta. Seetõttu võtab 80% kogu esipaneelist enda alla kasutajale silmapaistev graafik. Graafiku all paiknevad kolm suurt nuppu ja üks väike nupp. Suured nupud on vastavalt „START/STOP“ nupp, „CALIBRATE“ nupp ja „MORE“ nupp ning väike nupp „AUTO: ON/OFF“. Peale selle on graafiku peal väiksed plussi ja miinusega tähistatud nupud, mis võimaldavad kasutajal käsitsi muuta graafiku x- ja y-telje suurust. Peaakna vaade on kujutatud Joonisel 14.

„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 seab saadud väärtused graafiku ordinaattelje maksimaalseks ning minimaalseks väärtuseks. Nii jääb nuppu vajutades graafik ekraani keskele 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.



Joonis 14 Peaaken andmete mõõtmise hetkel.

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 esialgsed maksimum ja miinimum väärtused. *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 graafikul uusi andmeid kuvada. Koordinaadipaari x-koordinaadiks on mõõdetud andmete ajahetk ning y-koordinaadiks andmete väärtus.

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 graafikujoone vahel on tühi 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 leiab järjestikuste andmete keskmise väärtuse. Puhvri suurust on võimalik muuta kõrvalaknas liuguriga. Mida suurem on puhver, seda rohkem andmeid silutakse ning seda lamedam on graafik.

4.6 Kõrvalakna disain

Nagu peaakna puhul, on ka kõrvalakna disainimisel rõhutud lihtsusele. Kõrvalakna vaade on näha Joonisel 15. Kõrvalakna üleval paremas nurgas asub silt, mis kuvab kasutajale reaalajas kuupäeva ning kellaaega. Selleks on genereeritud peafailis kell, mis uuendab sildi nime üks kord sekundis. Sildi all asub kaks liugurit, millest üks muudab graafiku puhvri suurus ning teine automaatse kalibreerimise sagedust. Liugurite all on kaks nuppu, mis võimaldavad kasutajal valida kuvatava graafiku tüüpi (EKG või FouKG).

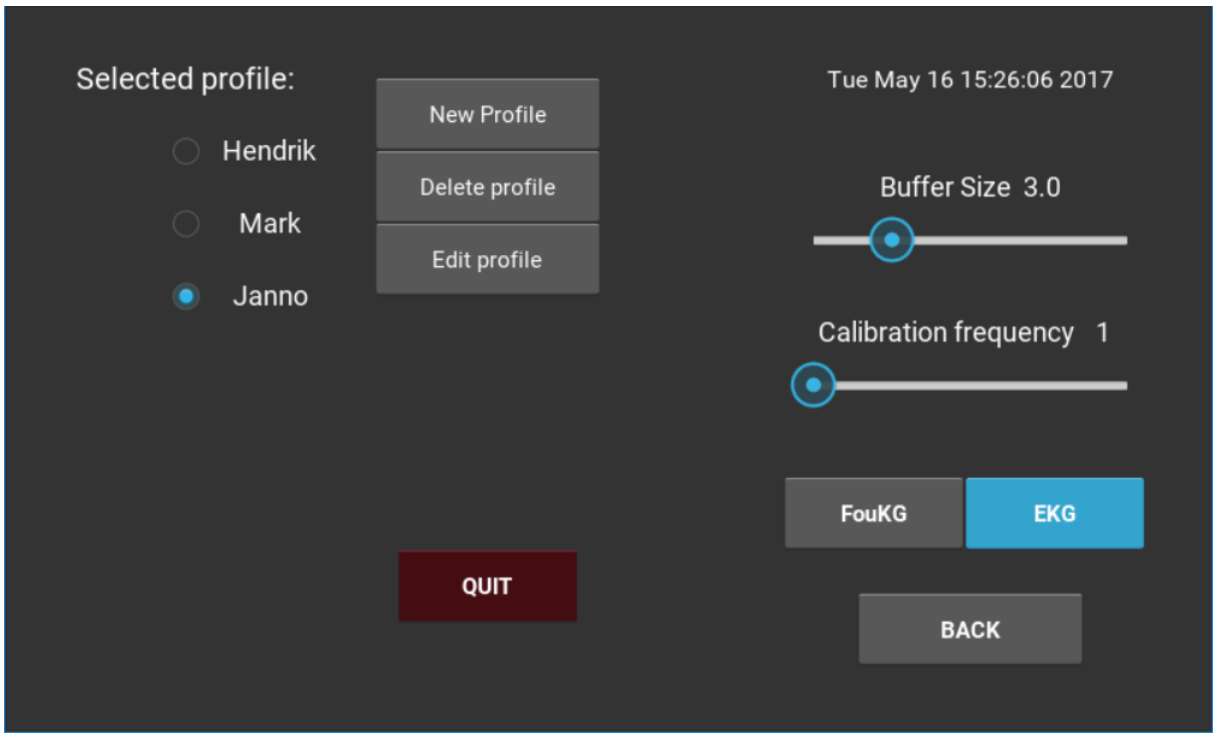
Akna vasakus küljes asub profiilide loetelu. Programmi käivitumisel loeb meetod *load_profiles()* andmebaasist kõik saadaolevad profiilid ning loob igaühe jaoks profiili nimelise sildi ja selle kõrvale raadionupu. Kasutajal on võimalik valida erinevaid profiile vajutades raadionuppude või nimesiltide peale. Raadionupud tagavad, et korraga oleks aktiivne ainult üks profiil.

Profiilide loetelu kõrval asuvad kolm nuppu „New Profile“, „Delete Profile“ ja „Edit Profile“. Nagu nimigi ütleb, vajutades nuppu „New Profile“, luuakse andmebaasi uus profiil vaikimisi määratud nimega ning koheselt luuakse ka uus nimesilt koos raadionupuga. Igale uuele profiilile luuakse ka talle unikaalne ID, mille kaudu tehakse andmebaasi päringuid. Nii võib programmis olla ka kaks või enam samanimelist profiili. Korraga saab programmis olla maksimaalselt kaheksa profiili. Kui maksimaalne profiilide arv on saavutatud ja kasutaja soovib luua veel ühte profiili, kuvatakse hüpikaknas vastav veateade.

Nupp „Delete profile“ kustutab hetkel aktiivsena oleva profiili. Kui alles on jäänud ainult üks profiil ja kasutaja soovib ka seda kustutada, kuvatakse hüpikaknas veateade. See on vajalik, sest alati peab eksisteerima vähemalt üks profiil, millele mõõtmistel andmed salvestatakse. Kui

kasutaja soovib ikkagi valitud profiili kustutada, on võimalik luua uus profiil ja koheselt kustutada vana.

Nupp „Edit profile“ kuvab kasutajale hüpikakna, kus on võimalik muuta hetkel aktiivsena oleva profiili nime. Kõigele lisaks on akna keskel nupp „QUIT“, mis sulgeb rakenduse.

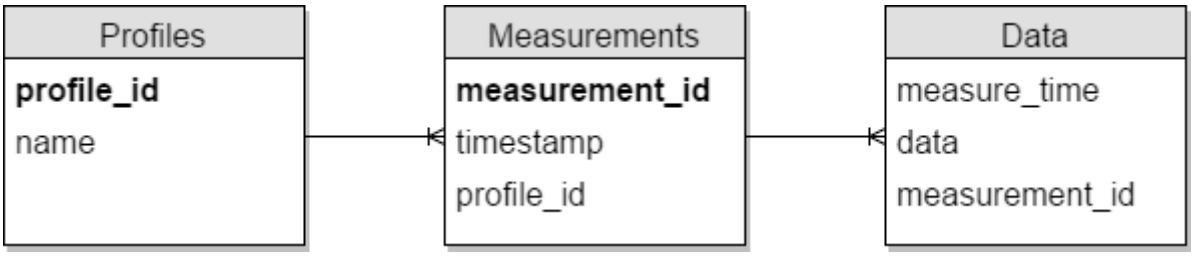


Joonis 15 Kõrvalaken aktiivse profiiliga Janno ja puhvri suurusega 3.

4.7 Andmebaas

Kardiograafi andmebaas koosneb kolmest tabelist. Tabel *Profiles* hoiab endas profiile. Igal profiilil on ID ja nimi. Tabel *Measurements* koosneb kõikidest tehtud mõõtmistest. Igal mõõtmisel on oma ID, mõõtmise ajahetk ning profiili ID. Tabel *Data* hoiab endas kõiki mõõdetud andmeid, kus igal mõõdetud väärtusel on mõõtmise aeg ning mõõtmise ID. Andmebaasi struktuur on kujutatud Joonisel 16.

Tabelisse *Profiles* tehakse uus sisestus, kui kasutaja vajutab nuppu „New Profile“ ning tabelist kustutatakse andmed „Delete Profile“ nupu vajutamisel. Tabelisse *Measurements* lisatakse andmed alati, kui kasutaja vajutab nuppu „START“ ehk alustab kardiograafiga mõõtmist. Läbi profiili ID välja jätab andmebaas meelde, mis kasutajal hetkel mõõtmisi sooritatakse. Mõõtmiste ajal salvestatakse kõik mõõdetud andmed listi ning alles mõõtmiste lõpus salvestatakse kogu info tabelisse *Data*. Mõõtmiste ajal ei salvestata andmebaasi andmeid selleks, et programmi töö oleks graafiku kuvamisel kiirem ning sujuvam. Kõik mõõdetud andmed on seotud hetkel töötava mõõtmise ID-ga.



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 eelnevate teadmiste põhjal on andmebaasist käsitsi päringute tegemisel võimalik saada järgmist informatsiooni:

- Kõik andmebaasis olevad profiilid.

  ```
  SELECT * FROM Profiles;
  ```

- Kõik mõõtmised ühe profiili kohta, kus *profile_Id* on vaadeldava profiili ID.

  ```
  SELECT * FROM Measurements WHERE profile_id=(profiili ID);
  ```

- Kõik andmed ühe mõõtmise kohta, kus *measurement_id* on vaadeldava mõõtmise ID.

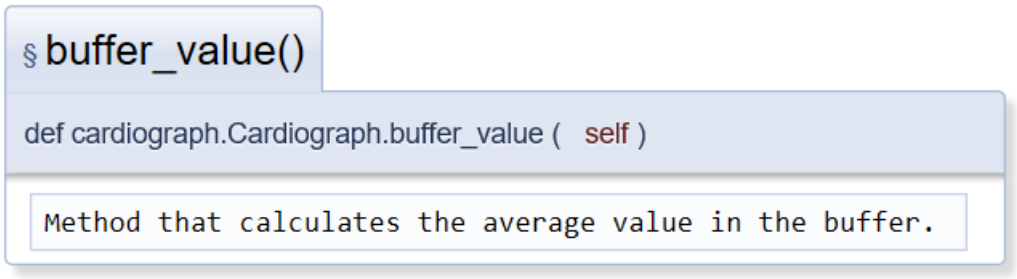
  ```
  SELECT * FROM Data WHERE measurement_id=(mõõtmise ID);
  ```

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

Et programmi töö oleks selgesti mõistetav, on kogu kood kommenteeritud vastavalt Doxygeni reeglitele. Kommenteeritud koodist on genereeritud Doxygeni HTML fail, mis seletab lahti programmis kasutatavate klasside, meetodite ning muutujate ülesanded. Nii on prototüübi tööga võimalik tutvuda ka ilma koodi ennast vaatamata. Joonisel 17 on näidatud üks programmis kasutatav meetod ning selle meetodi Doxygeni genereeritud dokumentatsioon.

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



Joonis 17 Näide kommenteeritud meetodist ja Doxygeni genereeritud dokumentatsioonist.

4.9 Olulisimad verstapostid ning nende ületamine

Käesolev töö õpetas, et sama kood võib töötada erinevate seadmete peal erinevalt. Näiteks disainiti programm esialgselt nii, et andmeid kuvav kood töötas eraldi lõimes (ingl *Thread*). Kood töötas Windowsis nii nagu oodatud, kuid Raspberry Pi peal hakkas graafik liikuma viivitustega. Nimelt ei suutnud Raspberry Pi piisavalt tihedalt graafikut uuendada ja samal ajal andmeid lugeda. Seetõttu on koodis genereeritud kell, mis kutsub iga 10 millisekundi tagant esile meetodi, mis kuvab graafikut. Nii uuendatakse graafiku andmeid 100 korda sekundis, mis on käesoleva programmi jaoks piisav.

Teiseks salvestati esialgselt andmeid pidevalt. Iga kord kui mõõdeti uued andmed, salvestati need kohe andmebaasi. Näiteks kui loetakse andmeid iga 4 millisekundi tagant, tuleks andmebaasi 250 korda sekundis sissekandeid teha. Nagu ka eelneva vea puhul, töötas Windowsis kood hästi, kuid pannes koodi Raspberry Pi peale, hakkas graafik end väga aeglaselt uuendama. Viga tuli sellest, et Raspberry Pi ei suutnud nii tihedalt andmebaasi infot salvestada. Vea vältimiseks salvestab programm esialgu mõõdetud andmed listi ja mõõtmise lõppedes kirjutab kõik andmed listist andmebaasi.

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

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

programmiviga, kus korduval hüpikakna avamisel ning sulgemisel viidatakse vanale, n-ö surnud hüpikaknale, mis põhjustab programmi sulgemise. Kuna antud viga ei suudetud lahendada, on kõik hüpikakna avamise käsud ümbritsetud veatuvastus plokiga.

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

5 Arutelu ja järeldused

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 Järeldused

Käesoleva tööga saavutati kõik eesmärgiks seatud ülesanded:

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

Haldurarvuti prototüüp töötab ootuspäraselt ning on valmis edasiseks arenduseks Foucault' kardiograafi üldises plaanis.

6 Kokkuvõte

Käesolev bakalaureusetöö kirjeldab Foucault' kardiograafi uue haldurarvuti prototüübi disaini ning tarkvaralist lahendust.

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

Kõik töös seatud eesmärgid said täidetud. Uus haldurarvuti töötab Raspberry Pi peal, mille külge on ühendatud puutetundlik ekraan. Ekraanilt on kasutajal võimalik näha graafikut mõõdetud andmetest ning muuta graafiku seadeid. Kõik mõõdetud andmed salvestatakse profiilipõhiselt andmebaasi.

Tehtud haldurarvuti on osa arendamisel olevast Foucault' kardiograafist.

Viited

[1] "WHO | The top 10 causes of death," *WHO*, 2017.

[2] "Funktsionaaldiagnostika osakond | Südamekeskus | Kliinikust | Sisekliinik | Kliinikud | Ida-Tallinna Keskhaigla." [Võrgumaterjal]. Saadaval: http://www.itk.ee/kliinikud/sisekliinik/sisekliinikust/sudamekeskus/funktsionaaldiagnostika-osakond. [Vaadatud: 15-May-2017].

[3] "Elektrokardiogramm." [Võrgumaterjal]. Saadaval: https://www.tlu.ee/opmat/ts/TST6025/elektrokardiogramm.html. [Vaadatud: 24-Apr-2017].

[4] "Kardiogrammi sakid segm interv.svg - Vikipeedia." [Võrgumaterjal]. Saadaval: https://et.wikipedia.org/wiki/Fail:Kardiogrammi_sakid_segm_interv.svg. [Vaadatud: 02-May-2017].

[5] O. Tarassova and J. Vedru, "Possibilities of Foucault Cardiography-Based Estimation of Heart Pumping Performance," in *13th International Conference on Electrical Bioimpedance and the 8th Conference on Electrical Impedance Tomography*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 586–589.

[6] "ThinkWiki - IBM ThinkPad 500." [Võrgumaterjal]. Saadaval: http://www.thinkwiki.org/wiki/Category:500. [Vaadatud: 15-May-2017].

[7] "Raspberry Pi Frequently Asked Questions." [Võrgumaterjal]. Saadaval: https://www.raspberrypi.org/help/faqs/. [Vaadatud: 22-Mar-2017].

[8] "Raspberry Pi 3 on sale." [Võrgumaterjal]. Saadaval: https://www.raspberrypi.org/blog/raspberry-pi-3-on-sale/. [Vaadatud: 23-Mar-2017].

[9] "RaspbianAbout - Raspbian." [Võrgumaterjal]. Saadaval: https://www.raspbian.org/RaspbianAbout/. [Vaadatud: 23-Mar-2017].

[10] "Raspberry Pi Touch Display." [Võrgumaterjal]. Saadaval: https://www.raspberrypi.org/products/raspberry-pi-touch-display/. [Vaadatud: 23-Mar-2017].

[11] "Adafruit Powerboost 1000C." [Võrgumaterjal]. Saadaval:

https://learn.adafruit.com/adafruit-powerboost-1000c-load-share-usb-charge-boost/overview. [Vaadatud: 24-Mar-2017].

[12] "ADS1115 16-Bit ADC." [Võrgumaterjal]. Saadaval: https://www.adafruit.com/product/1085. [Vaadatud: 24-Mar-2017].

[13] "General Python FAQ — Python 2.7.13 documentation." [Võrgumaterjal]. Saadaval: https://docs.python.org/2/faq/general.html. [Vaadatud: 03-May-2017].

[14] "PyCharm - Features." [Võrgumaterjal]. Saadaval: https://www.jetbrains.com/pycharm/features/. [Vaadatud: 21-Apr-2017].

[15] "GuiProgramming - Python Wiki." [Võrgumaterjal]. Saadaval: https://wiki.python.org/moin/GuiProgramming. [Vaadatud: 14-May-2017].

[16] "Kivy: Cross-platform Python Framework for NUI Development." [Võrgumaterjal]. Saadaval: https://kivy.org/#home. [Vaadatud: 15-Apr-2017].

[17] "Kivy Basics — Create an application." [Võrgumaterjal]. Saadaval: https://kivy.org/docs/guide/basic.html#create-an-application. [Vaadatud: 06-May-2017].

[18] "Garden — Kivy 1.9.2.dev0 documentation." [Võrgumaterjal]. Saadaval: https://kivy.org/docs/api-kivy.garden.html. [Vaadatud: 21-Apr-2017].

[19] "Kivy Garden - Graph." [Võrgumaterjal]. Saadaval: https://github.com/kivy-garden/garden.graph. [Vaadatud: 21-Apr-2017].

[20] "Kv language — Kivy 1.9.2.dev0 documentation." [Võrgumaterjal]. Saadaval: https://kivy.org/docs/guide/lang.html. [Vaadatud: 21-Apr-2017].

[21] "DB-Engines Ranking - popularity ranking of database management systems." [Võrgumaterjal]. Saadaval: https://db-engines.com/en/ranking. [Vaadatud: 14-May-2017].

[22] "About SQLite." [Võrgumaterjal]. Saadaval: https://www.sqlite.org/about.html. [Vaadatud: 07-May-2017].

[23] "Doxygen: Main Page." [Võrgumaterjal]. Saadaval: http://www.stack.nl/~dimitri/doxygen/index.html. [Vaadatud: 07-May-2017].

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 time
import sqlite3
from operator import itemgetter

## 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:
    '''
    Class that handles all the graph related things.
    '''

    ## The constructor.
    def __init__(self):
        ## Currently measured data.
        self.present_data = 0
        ## All measured data in one measurement.
        self.measured_data = []

        ## Previously measured data, that will be showed on the graph (EKG
or FouKG)
        self.data_dict = {}
        ## Previously measured data end time in milliseconds.
        self.end_time = 0
        ## Begin time of the measuring.
        self.begin_time = 0
        ## Current time of current measurement.
        self.current_time = 0
        ## Passed time in chuncks.
        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 cheks 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))
```

```
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)
            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)
```

```
        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
list
            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
list
            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

    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_frequecy(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.
        '''
        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)

    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

    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

    def quit_application(self):
        '''
        Method that stops the application.
        '''
        App.get_running_app().stop()

    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

    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 poupup 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 poupup 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()
        label.text = self.ids.edit_profile_textinput.text
        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.
        '''
        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: "-"
                        bold: True
                        font_size: 24
                        background_color: [1, 1, 1, 0.5]
                        on_press:
                            root.decrease_abscissa()

                    Button:
                        id: inc_ordinate_button
                        size_hint: [0.07, 0.1]
```

```
                pos_hint: {"center_x": 0.13, 'center_y': 0.9}
                text: "+"
                font_size: 20
                bold: True
                background_color: [1, 1, 1, 0.5]
                on_press:
                    root.increase_ordinate()

            Button:
                id: dec_ordinate_button
                size_hint: [0.07, 0.1]
                pos_hint: {"center_x": 0.13, 'center_y': 0.3}
                text: "-"
                bold: True
                font_size: 24
                background_color: [1, 1, 1, 0.5]
                on_press:
                    root.decrease_ordinate()

        BoxLayout:
            size_hint: [1, .2]

            Button:
                id: button1
                text: "START"
                bold: True
                on_press: root.start_stop()

            BoxLayout:
                id: boxlayout2
                orientation: "horizontal"

                Button:
                    size_hint: [.7, 1]
                    id: button2
                    text: "CALIBRATE"
                    bold: True
                    on_press: root.calibrate(1)
                    #on_press: root.increase_abscissa()

                ToggleButton:
                    id: togglebutton1
                    size_hint: [.3, 1]
                    text: "AUTO:\n  OFF"
                    bold: True
                    on_press: root.auto_calibrate()

            Button:
                id: button3
                text: "MORE"
                bold: True
                on_press: root.stop()
                on_release:
                    #root.decrease_abscissa()
                    sm.transition.direction = 'left'
                    sm.current = 'screen2'

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_edit_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: label_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_frequecy()
            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()
```

Lisa 3. Klassi *Cardiograph* Doxygeni dokumentatsioon

Public Member Functions | Public Attributes | List of all members cardiograph.Cardiograph Class Reference

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 chuncks. 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 cheks 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 chuncks.

§ 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, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

(b) üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas 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**